

SPRINGER
REFERENCE

David Padua
Editor-in-Chief

Encyclopedia of Parallel Computing

Encyclopedia of Parallel Computing

David Padua (Ed.)

Encyclopedia of Parallel Computing

With 880 Figures and 98 Tables

Editor-in-Chief
David Padua
University of Illinois at Urbana-Champaign
Urbana, IL
USA

ISBN 978-0-387-09765-7 e-ISBN 978-0-387-09766-4
DOI 10.1007/978-0-387-09766-4
Print and electronic bundle ISBN: 978-0-387-09844-9
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2011935063

© Springer Science+Business Media, LLC 2011

All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

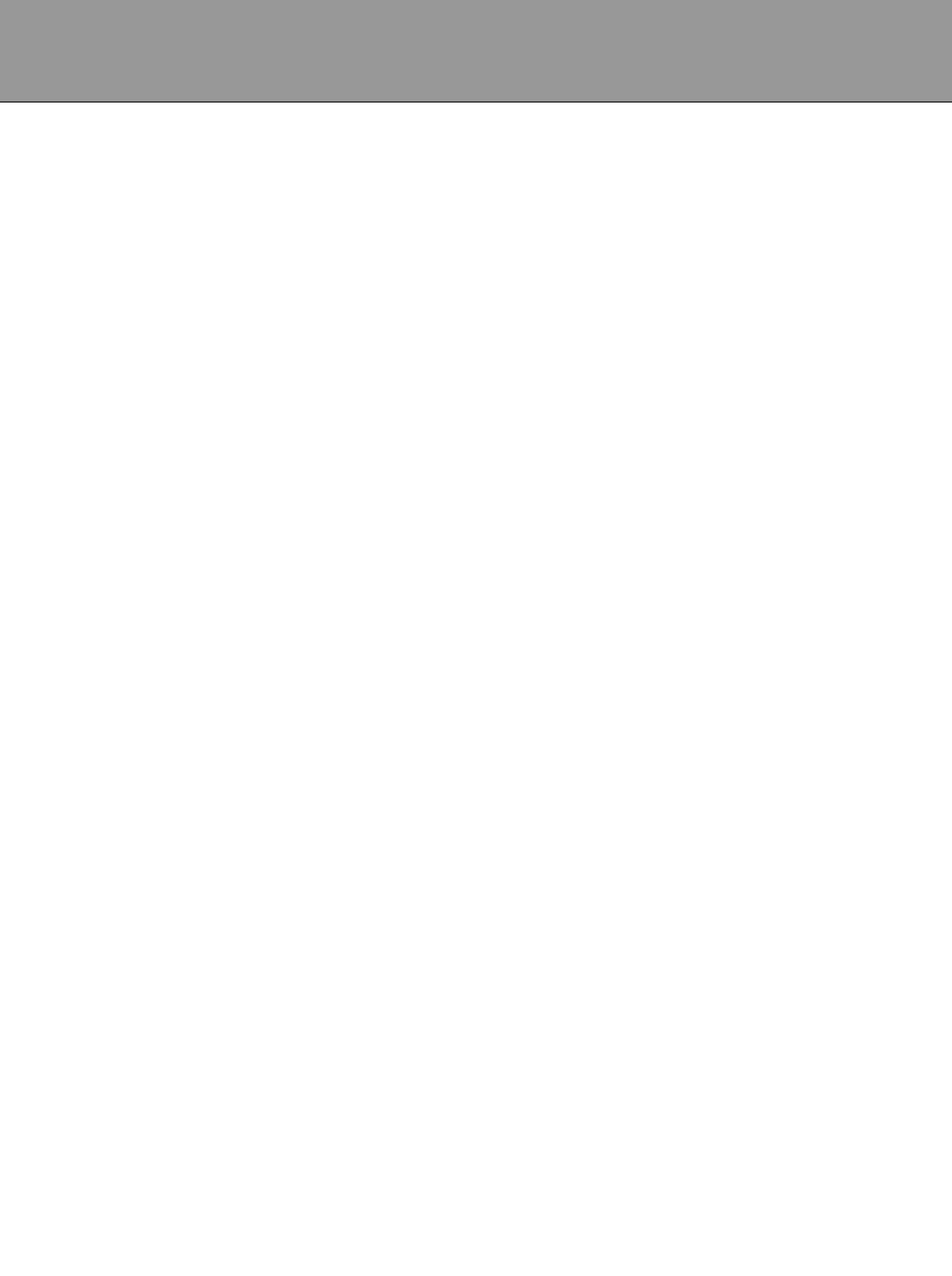
Preface

Parallelism, the capability of a computer to execute operations concurrently, has been a constant throughout the history of computing. It impacts hardware, software, theory, and applications. The fastest machines of the past few decades, the supercomputers, owe their performance advantage to parallelism. Today, physical limitations have forced the adoption of parallelism as the preeminent strategy of computer manufacturers for continued performance gains of all classes of machines, from embedded and mobile systems to the most powerful servers. Parallelism has been used to simplify the programming of certain applications which react to or simulate the parallelism of the natural world. At the same time, parallelism complicates programming when the objective is to take advantage of the existence of multiple hardware components to improve performance. Formal methods are necessary to study the correctness of parallel algorithms and implementations and to analyze their performance on different classes of real and theoretical systems. Finally, parallelism is crucial for many applications in the sciences, engineering, and interactive services such as search engines.

Because of its importance and the challenging problems it has engendered, there have been numerous research and development projects during the past half century. This Encyclopedia is our attempt to collect accurate and clear descriptions of the most important of those projects. Although not exhaustive, with over 300 entries the Encyclopedia covers most of the topics that we identified at the outset as important for a work of this nature. Entries include many of the best known projects and span all the important dimensions of parallel computing including machine design, software, programming languages, algorithms, theoretical issues, and applications.

This Encyclopedia is the result of the work of many, whose dedication made it possible. The 25 Editorial Board Members created the list of entries, did most of the reviewing, and suggested authors for the entries. Colin Robertson, the Managing Editor, Jennifer Carlson, Springer's Reference Development editor, and Editorial Assistants Julia Koerting and Simone Tavenrath, worked for 3 long years coordinating the recruiting of authors and the review and submission process. Melissa Fearon, Springer's Senior Editor, helped immensely with the coordination of authors and Editorial Board Members, especially during the difficult last few months. The nearly 400 authors wrote crisp, informative entries. They include experts in all major areas, come from many different nations, and span several generations. In many cases, the author is the lead designer or researcher responsible for the contribution reported in the entry. It was a great pleasure for me to be part of this project. The enthusiasm of everybody involved made this a joyful enterprise. I believe we have put together a meaningful snapshot of parallel computing in the last 50 years and presented a believable glimpse of the future. I hope the reader agrees with me on this and finds the entries, as I did, valuable contributions to the literature.

David Padua
Editor-in-Chief
University of Illinois at Urbana-Champaign
Urbana, IL
USA



Editors



David Padua
Editor-in-Chief
University of Illinois at Urbana-Champaign
Urbana, IL
USA



Sarita Adve
Editorial Board Member
University of Illinois at Urbana-Champaign
Urbana, IL
USA



Colin Robertson
Managing Editor
University of Illinois at Urbana-Champaign
Urbana, IL
USA



Gheorghe S. Almasi
Editorial Board Member
IBM T. J. Watson Research Center
Yorktown Heights, NY
USA



Srinivas Aluru
Editorial Board Member
Iowa State University
Ames, IA
USA



Gianfranco Bilardi
Editorial Board Member
University of Padova
Padova
Italy



David Bader
Editorial Board Member
Georgia Tech
Atlanta, GA
USA



Siddharta Chatterjee
Editorial Board Member
IBM Systems & Technology Group
Austin, TX
USA



Luiz DeRose
Editorial Board Member
Cray Inc.
St. Paul, MN
USA



José Duato
Editorial Board Member
Universitat Politècnica de València
València
Spain



Jack Dongarra
Editorial Board Member
University of Tennessee
Knoxville, TN
USA
Oak Ridge National Laboratory
Oak Ridge, TN
USA
University of Manchester
Manchester
UK



Paul Feautrier
Editorial Board Member
Ecole Normale Supérieure de Lyon
Lyon
France



María J. Garzarán
Editorial Board Member
University of Illinois at Urbana Champaign
Urbana, IL
USA



William Gropp
Editorial Board Member
University of Illinois Urbana-Champaign
Urbana, IL
USA



Michael Gerndt
Editorial Board Member
Technische Universitaet Muenchen
Garching
Germany



Thomas Gross
Editorial Board Member
ETH Zurich
Zurich
Switzerland



James C. Hoe
Editorial Board Member
Carnegie Mellon University
Pittsburgh, PA
USA



Hironori Kasahara
Editorial Board Member
Waseda University
Tokyo
Japan



Laxmikant Kale
Editorial Board Member
University of Illinois at Urbana Champaign
Urbana, IL
USA



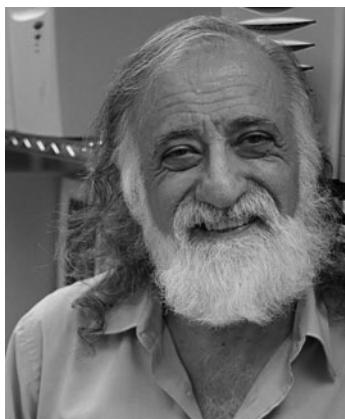
Christian Lengauer
Editorial Board Member
University of Passau
Passau
Germany



José E. Moreira
Editorial Board Member
IBM Thomas J. Watson Research Center
Yorktown Heights, NY
USA



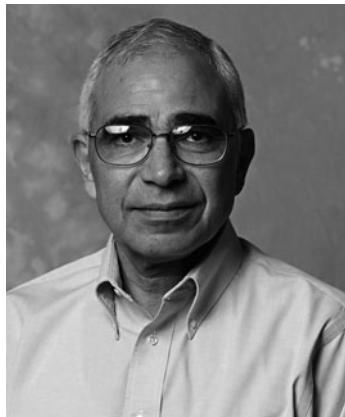
Keshav Pingali
Editorial Board Member
The University of Texas at Austin
Austin, TX
USA



Yale N. Patt
Editorial Board Member
The University of Texas at Austin
Austin, TX
USA



Markus Püschel
Editorial Board Member
ETH Zurich
Zurich
Switzerland



Ahmed H. Sameh
Editorial Board Member
Purdue University
West Lafayette, IN
USA



Pen-Chung Yew
Editorial Board Member
University of Minnesota at Twin Cities
Minneapolis, MN
USA



Vivek Sarkar
Editorial Board Member
Rice University
Houston, TX
USA



List of Contributors

DENNIS ABTS
Google Inc.
Madison, WI
USA

SARITA V. ADVE
University of Illinois at Urbana-Champaign
Urbana, IL
USA

GUL AGHA
University of Illinois at Urbana-Champaign
Urbana, IL
USA

JASMIN AJANOVIC
Intel Corporation
Portland, OR
USA

SEЛИM G. AKL
Queen's University
Kingston, ON
Canada

HASAN AKTULGA
Purdue University
West Lafayette, IN
USA

JOSÉ I. ALIAGA
TU Braunschweig Institute of Computational Mathematics
Braunschweig
Germany

ERIC ALLEN
Oracle Labs
Austin, TX
USA

GEORGE ALMASI
IBM
Yorktown Heights, NY
USA

SRINIVAS ALURU
Iowa State University
Ames, IA
USA
and
Indian Institute of Technology Bombay
Mumbai
India

PATRICK AMESTOY
Université de Toulouse ENSEEIHT-IRIT
Toulouse cedex 7
France

BABA ARIMILLI
IBM Systems and Technology Group
Austin, TX
USA

ROGER S. ARMEN
Thomas Jefferson University
Philadelphia, PA
USA

DOUGLAS ARMSTRONG
Intel Corporation
Champaign, IL
USA

DAVID I. AUGUST
Princeton University
Princeton, NJ
USA

CEVDET AYKANAT
Bilkent University
Ankara
Turkey

DAVID A. BADER
Georgia Institute of Technology
Atlanta, GA
USA

MICHAEL BADER
Universität Stuttgart
Stuttgart
Germany

DAVID H. BAILEY
Lawrence Berkeley National Laboratory
Berkeley, CA
USA

RAJEEV BALASUBRAMONIAN
University of Utah
Salt Lake City, UT
USA

UTPAL BANERJEE
University of California at Irvine
Irvine, CA
USA

ALESSANDRO BARDINE
Università di Pisa
Pisa
Italy

MUTHU MANIKANDAN BASKARAN
Reservoir Labs, Inc.
New York, NY
USA

CÉDRIC BASTOUL
University Paris-Sud 11 - INRIA Saclay Île-de-France
Orsay
France

AARON BECKER
University of Illinois at Urbana-Champaign
Urbana, IL
USA

MICHAEL W. BERRY
The University of Tennessee
Knoxville, TN
USA

ABHINAV BHATELE
University of Illinois at Urbana-Champaign
Urbana, IL
USA

SCOTT BIERSDORFF
University of Oregon
Eugene, OR
USA

GIANFRANCO BILARDI
University of Padova
Padova
Italy

ROBERT BJORNSEN
Yale University
New Haven, CT
USA

GUY BLELLOCH
Carnegie Mellon University
Pittsburgh, PA
USA

ROBERT BOCCINO
Carnegie Mellon University
Pittsburgh, PA
USA

HANS J. BOEHM
HP Labs
Palo Alto, CA
USA

ERIC J. BOHM
University of Illinois at Urbana-Champaign
Urbana, IL
USA

MATTHIAS BOLLHÖFER
Universitat Jaume I
Castellón
Spain

DAN BONACHEA
Lawrence Berkeley National Laboratory
Berkeley, CA
USA

PRADIP BOSE
IBM Corp. T.J. Watson Research Center
Yorktown Heights, NY
USA

MARIAN BREZINA

University of Colorado at Boulder
Boulder, CO
USA

JEFF BROOKS

Cray Inc.
St. Paul, MN
USA

HOLGER BRUNST

Technische Universität Dresden
Dresden
Germany

HANS-JOACHIM BUNGARTZ

Technische Universität München
Garching
Germany

MICHAEL G. BURKE

Rice University
Houston, TX
USA

ALFREDO BUTTARI

Université de Toulouse ENSEEIHT-IRIT
Toulouse cedex 7
France

ERIC J. BYLASKA

Pacific Northwest National Laboratory
Richland, WA
USA

ROY H. CAMPBELL

University of Illinois at Urbana-Champaign
Urbana, IL
USA

WILLIAM CARLSON

Institute for Defense Analyses
Bowie, MD
USA

MANUEL CARRO

Universidad Politécnica de Madrid
Madrid
Spain

ÜMIT V. ÇATALYÜREK

The Ohio State University
Columbus, OH
USA

LUIS H. CEZE

University of Washington
Seattle, WA
USA

BRADFORD L. CHAMBERLAIN

Cray Inc.
Seattle, WA
USA

ERNIE CHAN

NVIDIA Corporation
Santa Clara, CA
USA

RONG-GUEY CHANG

National Chung Cheng University
Chia-Yi
Taiwan

BARBARA CHAPMAN

University of Houston
Houston, TX
USA

DAVID CHASE

Oracle Labs
Burlington, MA
USA

DANIEL CHAVARRÍA-MIRANDA

Pacific Northwest National Laboratory
Richland, WA
USA

NORMAN H. CHRIST

Columbia University
New York, NY
USA

MURRAY COLE

University of Edinburgh
Edinburgh
UK

PHILLIP COLELLA
University of California
Berkeley, CA
USA

SALVADOR COLL
Universidad Politécnica de Valencia
Valencia
Spain

GUOJING CONG
IBM
Yorktown Heights, NY
USA

JAMES H. COWNIE
Intel Corporation (UK) Ltd.
Swindon
UK

ANTHONY P. CRAIG
National Center for Atmospheric Research
Boulder, CO
USA

ANTHONY CURTIS
University of Houston
Houston
TX

H. J. J. VAN DAM
Pacific Northwest National Laboratory
Richland, WA
USA

FREDERICA DAREMA
National Science Foundation
Arlington, VA
USA

ALAIN DARTE
École Normale Supérieure de Lyon
Lyon
France

RAJA DAS
IBM Corporation
Armonk, NY
USA

KAUSHIK DATTA
University of California
Berkeley, CA
USA

JIM DAVIES
Oxford University
UK

JAMES DEMMEL
University of California at Berkeley
Berkeley, CA
USA

MONTY DENNEAU
IBM Corp., T.J. Watson Research Center
Yorktown Heights, NY
USA

JACK B. DENNIS
Massachusetts Institute of Technology
Cambridge, MA
USA

MARK DEWING
Intel Corporation
Champaign, IL
USA

VOLKER DIEKERT
Universität Stuttgart FMI
Stuttgart
Germany

JACK DONGARRA
University of Tennessee
Knoxville, TN
USA

DAVID DONOFRIO
Lawrence Berkeley National Laboratory
Berkeley, CA
USA

RON O. DROR
D. E. Shaw Research
New York, NY
USA

IAIN DUFF
Science & Technology Facilities Council
Didcot, Oxfordshire
UK

MICHAEL DUNGWORTH

SANDHYA DWARKADAS
University of Rochester
Rochester, NY
USA

RUDOLF EIGENMANN
Purdue University
West Lafayette, IN
USA

E. N. (MOOTAZ) ELNOZAHY
IBM Research
Austin, TX
USA

JOEL EMER
Intel Corporation
Hudson, MA
USA

BABAK FALSAFI
Ecole Polytechnique Fédérale de Lausanne
Lausanne
Switzerland

PAOLO FARABOSCHI
Hewlett Packard
Sant Cugat del Valles
Spain

PAUL FEAUTRIER
Ecole Normale Supérieure de Lyon
Lyon
France

KARL FEIND
SGI
Eagan, MN
USA

WU-CHUN FENG
Virginia Tech
Blacksburg, VA
USA
and
Wake Forest University
Winston-Salem, NC
USA

JOHN FEO
Pacific Northwest National Laboratory
Richland, WA
USA

JEREMY T. FINEMAN
Carnegie Mellon University
Pittsburgh, PA
USA

JOSEPH A. FISHER
Miami Beach, FL
USA

CORMAC FLANAGAN
University of California at Santa Cruz
Santa Cruz, CA
USA

JOSÉ FLICH
Technical University of Valencia
Valencia
Spain

CHRISTINE FLOOD
Oracle Labs
Burlington, MA
USA

MICHAEL FLYNN
Stanford University
Stanford, CA
USA

JOSEPH FOGARTY
University of South Florida
Tampa, FL
USA

PIERFRANCESCO FOGLIA

Università di Pisa

Pisa

Italy

TRYGGVE FOSSUM

Intel Corporation

Hudson, MA

USA

GEOFFREY FOX

Indiana University

Bloomington, IN

USA

MARTIN FRÄNZLE

Carl von Ossietzky Universität

Oldenburg

Germany

FRANZ FRANCHETTI

Carnegie Mellon University

Pittsburgh, PA

USA

STEFAN M. FREUDENBERGER

Freudenberger Consulting

Zürich

Switzerland

HOLGER FRÖNING

University of Heidelberg

Heidelberg

Germany

KARL FÜRLINGER

Ludwig-Maximilians-Universität München

Munich

Germany

EFSTRATIOS GALLOPOULOS

University of Patras

Patras

Greece

ALAN GARA

IBM T.J. Watson Research Center

Yorktown Heights, NY

USA

PEDRO J. GARCIA

Universidad de Castilla-La Mancha

Albacete

Spain

MICHAEL GARLAND

NVIDIA Corporation

Santa Clara, CA

USA

KLAUS GÄRTNER

Weierstrass Institute for Applied Analysis and Stochastics

Berlin

Germany

ED GEHRINGER

North Carolina State University

Raleigh, NC

USA

ROBERT A. VAN DE GEIJN

The University of Texas at Austin

Austin, TX

USA

AL GEIST

Oak Ridge National Laboratory

Oak Ridge, TN

USA

THOMAS GEORGE

IBM Research

Delhi

India

MICHAEL GERNDT

Technische Universität München

München

Germany

AMOL GHOTING

IBM Thomas. J. Watson Research Center

Yorktown Heights, NY

USA

JOHN GILBERT

University of California

Santa Barbara, CA

USA

ROBERT J. VAN GLABBEK
NICTA
Sydney
Australia
and
The University of New South Wales
Sydney
Australia
and
Stanford University
Stanford, CA
USA

SERGEI GORLATCH
Westfälische Wilhelms-Universität Münster
Münster
Germany

KAZUSHIGE GOTO
The University of Texas at Austin
Austin, TX
USA

ALLAN GOTTLIEB
New York University
New York, NY
USA

STEVEN GOTTLIEB
Indiana University
Bloomington, IN
USA

N. GOVIND
Pacific Northwest National Laboratory
Richland, WA
USA

SUSAN L. GRAHAM
University of California
Berkeley, CA
USA

ANANTH Y. GRAMA
Purdue University
West Lafayette, IN
USA

DON GRICE
IBM Corporation
Poughkeepsie, NY
USA

LAURA GRIGORI
Laboratoire de Recherche en Informatique Université
Paris-Sud 11
Paris
France

WILLIAM GROPP
University of Illinois at Urbana-Champaign
Urbana, IL
USA

ABDOU GUERMOUCHE
Université de Bordeaux
Talence
France

JOHN A. GUNNELS
IBM Corp
Yorktown Heights, NY
USA

ANSHUL GUPTA
IBM T.J. Watson Research Center
Yorktown Heights, NY
USA

JOHN L. GUSTAFSON
Intel Corporation
Santa Clara, CA
USA

ROBERT H. HALSTEAD
Curl Inc.
Cambridge, MA
USA

KEVIN HAMMOND
University of St. Andrews
St. Andrews
UK

JAMES HARRELL

ROBERT HARRISON
Oak Ridge National Laboratory
Oak Ridge, TN
USA

JOHN C. HART
University of Illinois at Urbana-Champaign
Urbana, IL
USA

MICHAEL HEATH
University of Illinois at Urbana-Champaign
Urbana, IL
USA

HERMANN HELLWAGNER
Klagenfurt University
Klagenfurt
Austria

DANNY HENDLER
Ben-Gurion University of the Negev
Beer-Sheva
Israel

BRUCE HENDRICKSON
Sandia National Laboratories
Albuquerque, NM
USA

ROBERT HENSCHEL
Indiana University
Bloomington, IN
USA

KIERAN T. HERLEY
University College Cork
Cork
Ireland

MAURICE HERLIHY
Brown University
Providence, RI
USA

MANUEL HERMENEGILDO
Universidad Politécnica de Madrid
Madrid
Spain

IMDEA Software Institute
Madrid
Spain

OSCAR HERNANDEZ
Oak Ridge National Laboratory
Oak Ridge, TN
USA

PAUL HILFINGER
University of California
Berkeley, CA
USA

KEI HIRAKI
The University of Tokyo
Tokyo
Japan

H. PETER HOFSTEE
IBM Austin Research Laboratory
Austin, TX
USA

CHRIS HSIUNG
Hewlett Packard
Palo Alto, CA
USA

JONATHAN HU
Sandia National Laboratories
Livermore, CA
USA

THOMAS HUCKLE
Technische Universität München
Garching
Germany

WEN-MEI HWU
University of Illinois at Urbana-Champaign
Urbana, IL
USA

FRANÇOIS IRIGOIN
MINES ParisTech/CRI
Fontainebleau
France

KEN'ICHI ITAKURA
Japan Agency for Marine-Earth Science and Technology
(JAMSTEC)
Yokohama
Japan

JOSEPH F. JAJA
University of Maryland
College Park, MD
USA

JOEFON JANN
T. J. Watson Research Center, IBM Corp.
Yorktown Heights, NY
USA

KARL JANSEN
NIC, DESY Zeuthen
Zeuthen
Germany

PRITISH JETLEY
University of Illinois at Urbana-Champaign
Urbana, IL
USA

WIBE A. DE JONG
Pacific Northwest National Laboratory
Richland, WA
USA

LAXMIKANT V. KALÉ
University of Illinois at Urbana-Champaign
Urbana, IL
USA

ANANTH KALYANARAMAN
Washington State University
Pullman, WA
USA

AMIR KAMIL
University of California
Berkeley, CA
USA

KRISHNA KANDALLA
The Ohio State University
Columbus, OH
USA

LARRY KAPLAN
Cray Inc.
Seattle, WA
USA

TEJAS S. KARKHANIS
IBM T.J. Watson Research Center
Yorktown Heights, NY
USA

RAJESH K. KARMANI
University of Illinois at Urbana-Champaign
Urbana, IL
USA

GEORGE KARYPIS
University of Minnesota
Minneapolis, MN
USA

ARUN KEJARIWAL
Yahoo! Inc.
Sunnyvale, CA
USA

MALEQ KHAN
Virginia Tech
Blacksburg, VA
USA

THILO KIELMANN
Vrije Universiteit
Amsterdam
The Netherlands

GERRY KIRSCHNER
Cray Incorporated
St. Paul, MN
USA

CHRISTOF KLAUSECKER
Ludwig-Maximilians-Universität München
Munich
Germany

KATHLEEN KNOBE
Intel Corporation
Cambridge, MA
USA

ANDREAS KNÜPFER
Technische Universität Dresden
Dresden
Germany

GIORGOS KOLLIAS
Purdue University
West Lafayette, IN
USA

K. KOWALSKI
Pacific Northwest National Laboratory
Richland, WA
USA

QUINCEY KOZIOL
The HDF Group
Champaign, IL
USA

DIETER KRANZLMÜLLER
Ludwig-Maximilians-Universität München
Munich
Germany

MANOJKUMAR KRISHNAN
Pacific Northwest National Laboratory
Richland, WA
USA

CHI-BANG KUAN
National Tsing-Hua University
Hsin-Chu
Taiwan

DAVID J. KUCK
Intel Corporation
Champaign, IL
USA

JEFFERY A. KUEHN
Oak Ridge National Laboratory
Oak Ridge, TN
USA

V. S. ANIL KUMAR
Virginia Tech
Blacksburg, VA
USA

KALYAN KUMARAN
Argonne National Laboratory
Argonne, IL
USA

JAMES LA GRONE
University of Houston
Houston, TX
USA

ROBERT LATHAM
Argonne National Laboratory
Argonne, IL
USA

BRUCE LEASURE
Saint Paul, MN
USA

JENQ-KUEN LEE
National Tsing-Hua University
Hsin-Chu
Taiwan

CHARLES E. LEISERSON
Massachusetts Institute of Technology
Cambridge, MA
USA

CHRISTIAN LENGAUER
University of Passau
Passau
Germany

RICHARD LETHIN
Reservoir Labs, Inc.
New York, NY
USA

ALLEN LEUNG
Reservoir Labs, Inc.
New York, NY
USA

JOHN M. LEVESQUE
Cray Inc.
Knoxville, TN
USA

MICHAEL LEVINE

JEAN-YVES L'EXCELLENT
ENS Lyon
Lyon
France

JIAN LI
IBM Research
Austin, TX
USA

XIAOYE SHERRY LI
Lawrence Berkeley National Laboratory
Berkeley, CA
USA

ZHIYUAN LI
Purdue University
West Lafayette, IN
USA

CALVIN LIN
University of Texas at Austin
Austin, TX
USA

HESHAN LIN
Virginia Tech
Blacksburg, VA
USA

HANS-WOLFGANG LOIDL
Heriot-Watt University
Edinburgh
UK

RITA LOOPEN
Philipps-Universität Marburg
Marburg
Germany

PEDRO LÓPEZ
Universidad Politécnica de Valencia
Valencia
Spain

GEOFF LOWNEY
Intel Corporation
Hudson, MA
USA

VICTOR LUCHANGCO
Oracle Labs
Burlington, MA
USA

PIOTR LUSCZEK
University of Tennessee
Knoxville, TN
USA

OLAV LYSNE
The University of Oslo
Oslo
Norway

XIAOSONG MA
North Carolina State University
Raleigh, NC
USA
and
Oak Ridge National Laboratory
Raleigh, NC
USA

ARTHUR B. MACCABE
Oak Ridge National Laboratory
Oak Ridge, TN
USA

KAMESH MADDURI
Lawrence Berkeley National Laboratory
Berkeley, CA
USA

JAN-WILLEM MAESSEN
Google
Cambridge, MA
USA

KONSTANTIN MAKARYCHEV
IBM T.J. Watson Research Center
Yorktown Heights, NY
USA

JUNICHIRO MAKINO
National Astronomical Observatory of Japan
Tokyo
Japan

ALLEN D. MALONY
University of Oregon
Eugene, OR
USA

MADHA V. MARATHE
Virginia Tech
Blacksburg, VA
USA

ALBERTO F. MARTÍN
Universitat Jaume I
Castellón
Spain

GLENN MARTYNA
IBM Thomas J. Watson Research Center
Yorktown Heights, NY
USA

ERIC R. MAY
University of Michigan
Ann Arbor, MI
USA

SALLY A. MCKEE
Chalmers University of Technology
Goteborg
Sweden

MIRIAM MEHL
Technische Universität München
Garching
Germany

BENOIT MEISTER
Reservoir Labs, Inc.
New York, NY
USA

PHILLIP MERKEY
Michigan Technological University
Houghton, MI
USA

JOSÉ MESEGUER
University of Illinois at Urbana-Champaign
Urbana, IL
USA

MICHAEL METCALF
Berlin
Germany

SAMUEL MIDKIFF
Purdue University
West Lafayette, IN
USA

KENICHI MIURA
National Institute of Informatics
Tokyo
Japan

BERND MOHR
Forschungszentrum Jülich GmbH
Jülich
Germany

JOSÉ E. MOREIRA
IBM T.J. Watson Research Center
Yorktown Heights, NY
USA

ALAN MORRIS
University of Oregon
Eugene, OR
USA

J. ELIOT B. MOSS
University of Massachusetts
Amherst, MA
USA

MATTHIAS MÜLLER
Technische Universität Dresden
Dresden
Germany

PETER MÜLLER
ETH Zurich
Zurich
Switzerland

YOICHI MURAOKA
Waseda University
Tokyo
Japan

ANCA MUSCHOLL
Université Bordeaux 1
Talence
France

RAVI NAIR
IBM Thomas J. Watson Research Center
Yorktown Heights, NY
USA

STEPHEN NELSON

MARIO NEMIROVSKY
Barcelona Supercomputer Center
Barcelona
Spain

RYAN NEWTON
Intel Corporation
Hudson, MA
USA

Rocco De Nicola
Universita' di Firenze
Firenze
Italy

ALEXANDRU NICOLAU
University of California Irvine
Irvine, CA
USA

JAREK NIEPLOCHA[†]
Pacific Northwest National Laboratory
Richland, WA
USA

ALLEN NIKORA
California Institute of Technology
Pasadena, CA
USA

ROBERT W. NUMRICH
City University of New York
New York, NY
USA

STEVEN OBERLIN

LEONID OLIKER
Lawrence Berkeley National Laboratory
Berkeley, CA
USA

DAVID PADUA
University of Illinois at Urbana-Champaign
Urbana, IL
USA

SCOTT PAKIN
Los Alamos National Laboratory
Los Alamos, NM
USA

BRUCE PALMER
Pacific Northwest National Laboratory
Richland, WA
USA

DHABALESWAR K. PANDA
The Ohio State University
Columbus, OH
USA

SAGAR PANDIT
University of South Florida
Tampa, FL
USA

YALE N. PATT
The University of Texas at Austin
Austin, TX
USA

[†]deceased

OLIVIER PÈNE
University de Paris-Sud-XI
Orsay Cedex
France

PAUL PETERSEN
Intel Corporation
Champaign, IL
USA

BERNARD PHILIPPE
Campus de Beaulieu
Rennes
France

MICHAEL PHILIPPSEN
University of Erlangen-Nuremberg
Erlangen
Germany

JAMES C. PHILLIPS
University of Illinois at Urbana-Champaign
Urbana, IL
USA

ANDREA PIETRACAPRINA
Università di Padova
Padova
Italy

KESHAV PINGALI
The University of Texas at Austin
Austin, TX
USA

TIMOTHY M. PINKSTON
University of Southern California
Los Angeles, CA
USA

ERIC POLIZZI
University of Massachusetts
Amherst, MA
USA

STEPHEN W. POOLE
Oak Ridge National Laboratory
Oak Ridge, TN
USA

WILFRED POST
Oak Ridge National Laboratory
Oak Ridge, TN
USA

CHRISTOPH VON PRAUN
Georg-Simon-Ohm University of Applied Sciences
Nuremberg
Germany

FRANCO P. PREPARATA
Brown University
Providence, RI
USA

COSIMO ANTONIO PRETE
Università di Pisa
Pisa
Italy

TIMOTHY PRINCE
Intel Corporation
Santa Clara, CA
USA

JEAN-PIERRE PROST
Morteau
France

GEPPINO PUCCI
Università di Padova
Padova
Italy

MARKUS PÜSCHEL
ETH Zurich
Zurich
Switzerland

ENRIQUE S. QUINTANA-ORTÍ
Universitat Jaume I
Castellón
Spain

PATRICE QUINTON
ENS Cachan Bretagne
Bruz
France

RAM RAJAMONY IBM Research Austin, TX USA	YVES ROBERT Ecole Normale Supérieure de Lyon France
ARUN RAMAN Princeton University Princeton, NJ USA	ARCH D. ROBISON Intel Corporation Champaign, IL USA
LAWRENCE RAUCHWERGER Texas A&M University College Station, TX USA	A. W. ROSCOE Oxford University Oxford UK
JAMES R. REINDERS Intel Corporation Hillsboro, OR USA	ROBERT B. ROSS Argonne National Laboratory Argonne, IL USA
STEVEN P. REINHARDT	CHRIS ROWEN CEO, Tensilica Santa Clara, CA, USA
JOHN REPPY University of Chicago Chicago, IL USA	DUNCAN ROWETH Cray (UK) Ltd. UK
MARÍA ENGRACIA GÓMEZ REQUENA Universidad Politécnica de Valencia Valencia Spain	SUKYOUNG RYU Korea Advanced Institute of Science and Technology Daejeon Korea
DANIEL RICCIUTO Oak Ridge National Laboratory Oak Ridge, TN USA	VALENTINA SALAPURA IBM Research Yorktown Heights, NY USA
ROLF RIESEN IBM Research Dublin Ireland	JOEL H. SALTZ Emory University Atlanta, GA USA
TANGUY RISSET INSA Lyon Villeurbanne France	AHMED SAMEH Purdue University West Lafayette, IN USA
	MIGUEL SANCHEZ Universidad Politécnica de Valencia Valencia Spain

BENJAMIN SANDER
Advanced Micro Device Inc.
Austin, TX
USA

PETER SANDERS
Universitaet Karlsruhe
Karlsruhe
Germany

DAVIDE SANGIORGI
Universita'di Bologna
Bologna
Italy

VIVEK SARIN
Texas A&M University
College Station, TX
USA

VIVEK SARKAR
Rice University
Houston, TX
USA

OLAF SCHENK
University of Basel
Basel
Switzerland

MICHAEL SCHLANSKER
Hewlett-Packard Inc.
Palo-Alto, CA
USA

STEFAN SCHMID
Telekom Laboratories/TU Berlin
Berlin
Germany

MARTIN SCHULZ
Lawrence Livermore National Laboratory
Livermore, CA
USA

JAMES L. SCHWARZMEIER
Cray Inc.
Chippewa Falls, WI
USA

MICHAEL L. SCOTT
University of Rochester
Rochester, NY
USA

MATOUS SEDLACEK
Technische Universität München
Garching
Germany

JOEL SEIFERAS
University of Rochester
Rochester, NY
USA

FRANK OLAF SEM-JACOBSEN
The University of Oslo
Oslo
Norway

ANDRÉ SEZNEC
IRISA/INRIA, Rennes
Rennes
France

JOHN SHALF
Lawrence Berkeley National Laboratory
Berkeley, CA
USA

MEIYUE SHAO
Umeå University
Umeå
Sweden

DAVID E. SHAW
D. E. Shaw Research
New York, NY
USA
and
Columbia University
New York, NY
USA

XIAOWEI SHEN
IBM Research
Armonk, NY
USA

SAMEER SHENDE
University of Oregon
Eugene, OR
USA

GALEN M. SHIPMAN
Oak Ridge National Laboratory
Oak Ridge, TN
USA

HOWARD JAY SIEGEL
Colorado State University
Fort Collins, CO
USA

DANIEL P. SIEWIOREK
Carnegie Mellon University
Pittsburgh, PA
USA

FEDERICO SILLA
Universidad Politécnica de Valencia
Valencia
Spain

BARRY SMITH
Argonne National Laboratory
Argonne, IL
USA

BURTON SMITH
Microsoft Corporation
Redmond, WA
USA

MARC SNIR
University of Illinois at Urbana-Champaign
Urbana, IL
USA

LAWRENCE SNYDER
University of Washington
Seattle, WA
USA

MARCO SOLINAS
Università di Pisa
Pisa
Italy

EDGAR SOLOMONIK
University of California at Berkeley
Berkeley, CA
USA

MATTHEW SOTTILE
Galois, Inc.
Portland, OR
USA

M'HAMED SOULI
Université des Sciences et Technologies de Lille
Villeneuve d'Ascq cédex
France

WYATT SPEAR
University of Oregon
Eugene, OR
USA

EVAN W. SPEIGHT
IBM Research
Austin, TX
USA

MARK S. SQUILLANTE
IBM
Yorktown Heights, NY
USA

ALEXANDROS STAMATAKIS
Heidelberg Institute for Theoretical Studies
Heidelberg
Germany

GUY L. STEELE, JR.
Oracle Labs
Burlington, MA
USA

THOMAS L. STERLING
Louisiana State University
Baton Rouge, LA
USA

TJERK P. STRAATSMA
Pacific Northwest National Laboratory
Richland, WA
USA

PAULA E. STREZ
Virginia Tech
Blacksburg, VA
USA

THOMAS M. STRICKER
Zürich, CH
Switzerland

JIMMY SU
University of California
Berkeley, CA
USA

HARI SUBRAMONI
The Ohio State University
Columbus, OH
USA

SAYANTAN SUR
The Ohio State University
Columbus, OH
USA

JOHN SWENSEN
CPU Technology
Pleasanton, CA
USA

HIROSHI TAKAHARA
NEC Corporation
Tokyo
Japan

MICHELA TAUFER
University of Delaware
Newark, DE
USA

VINOD TIPPARAJU
Oak Ridge National Laboratory
Oak Ridge, TN
USA

ALEXANDER TISKIN
University of Warwick
Coventry
UK

JOSEP TORRELAS
University of Illinois at Urbana-Champaign
Urbana, IL
USA

JESPER LARSSON TRÄFF
University of Vienna
Vienna
Austria

PHILIP TRINDER
Heriot-Watt University
Edinburgh
UK

RAFFAELE TRIPICCIONE
Università di Ferrara and INFN Sezione di Ferrara
Ferrara
Italy

MARK TUCKERMAN
New York University
New York, NY
USA

RAY TUMINARO
Sandia National Laboratories
Livermore, CA
USA

BORA UÇAR
ENS Lyon
Lyon
France

MARAT VALIEV
Pacific Northwest National Laboratory
Richland, WA
USA

NICOLAS VASILACHE
Reservoir Labs, Inc.
New York, NY
USA

MARIANA VERTENSTEIN
National Center for Atmospheric Research
Boulder, CO
USA

JENS VOLKERT
Johannes Kepler University Linz
Linz
Austria

YEVGEN VORONENKO
Carnegie Mellon University
Pittsburgh, PA
USA

RICHARD W. VUDUC
Georgia Institute of Technology
Atlanta, GA
USA

GENE WAGENBRETH
University of Southern California
Topanga, CA
USA

DALI WANG
Oak Ridge National Laboratory
Oak Ridge, TN
USA

JASON WANG
LSTC
Livermore, CA
USA

GREGORY R. WATSON
IBM
Yorktown Heights, NY
USA

ROGER WATTENHOFER
ETH Zürich
Zurich
Switzerland

MICHAEL WEHNER
Lawrence Berkeley National Laboratory
Berkeley, CA
USA

JOSEF WEIDENDORFER
Technische Universität München
München
Germany

TONG WEN
University of California
Berkeley, CA
USA

R. CLINT WHALEY
University of Texas at San Antonio
San Antonio, TX
USA

ANDREW B. WHITE
Los Alamos National Laboratory
Los Alamos, NM
USA

BRIAN WHITNEY
Oracle Corporation
Hillsboro, OR
USA

ROLAND WISMÜLLER
University of Siegen
Siegen
Germany

ROBERT W. WISNIEWSKI
IBM
Yorktown Heights, NY
USA

DAVID WOHLFORD
Reservoir Labs, Inc.
New York, NY
USA

FELIX WOLF
Aachen University
Aachen
Germany

DAVID WONNACOTT
Haverford College
Haverford, PA
USA

PATRICK H. WORLEY
Oak Ridge National Laboratory
Oak Ridge, TN
USA

SUDHAKAR YALAMANCHILI
Georgia Institute of Technology
Atlanta, GA
USA

KATHERINE YELICK
University of California at Berkeley and Lawrence Berkeley
National Laboratory
Berkeley, CA
USA

PEN-CHUNG YEW
University of Minnesota at Twin-Cities
Minneapolis, MN
USA

BOBBY DALTON YOUNG
Colorado State University
Fort Collins, CO
USA

CLIFF YOUNG
D. E. Shaw Research
New York, NY
USA

GABRIEL ZACHMANN
Clausthal University
Clausthal-Zellerfeld
Germany

FIELD G. VAN ZEEF
The University of Texas at Austin
Austin, TX
USA

LIXIN ZHANG
IBM Research
Austin, TX
USA

GENGBIN ZHENG
University of Illinois at Urbana-Champaign
Urbana, IL
USA

HANS P. ZIMA
California Institute of Technology
Pasadena, CA
USA

JAROSLAW ZOLA
Iowa State University
Ames, IA
USA

A

Ab Initio Molecular Dynamics

- Car-Parrinello Method

Access Anomaly

- Race Conditions

Actors

RAJESH K. KARMANI, GUL AGHA

University of Illinois at Urbana-Champaign, Urbana, IL, USA

Definition

Actors is a model of concurrent computation for developing parallel, distributed, and mobile systems. Each actor is an autonomous object that operates concurrently and asynchronously, receiving and sending messages to other actors, creating new actors, and updating its own local state. An actor system consists of a collection of actors, some of whom may send messages to, or receive messages from, actors outside the system.

Preliminaries

An *actor* has a name that is globally unique and a behavior which determines its actions. In order to send an actor a message, the actor's name must be used; a name cannot be guessed but it may be communicated in a message. When an actor is idle, and it has a pending message, the actor accepts the message and does the computation defined by its behavior. As a result, the actor may take three types of actions: *send messages*, *create new actors*, and *update* its local state. An actor's behavior may change as it modifies its local state.

Actors do not share state: an actor must explicitly send a message to another actor in order to affect the latter's behavior. Each actor carries out its actions concurrently (and asynchronously) with other actors. Moreover, the path a message takes as well as network delays it may encounter are not specified. Thus, the arrival order of messages is indeterminate. The key semantic properties of the standard Actor model are *encapsulation* of state and *atomic* execution of a method in response to a message, *fairness* in scheduling actors and in the delivery of messages, and *location transparency* enabling distributed execution and mobility.

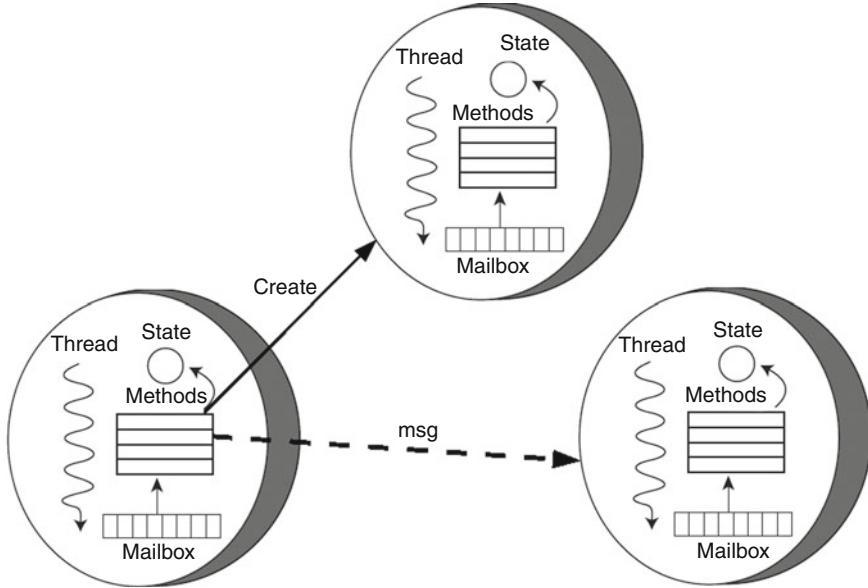
Advantages of the Actor Model

In the object-oriented programming paradigm, an object encapsulates data and behavior. This separates the interface of an object (what an object does) from its representation (how it does it). Such separation enables modular reasoning about object-based programs and facilitates their evolution. Actors extend the advantages of objects to concurrent computations by separating control (where and when) from the logic of a computation.

The Actor model of programming [3] allows programs to be decomposed into self-contained, autonomous, interactive, asynchronously operating components. Due to their asynchronous operation, actors provide a model for the nondeterminism inherent in distributed systems, reactive systems, mobile systems, and any form of interactive computing.

History

The concept of actors has developed over 3 decades. The earliest use of the term "actors" was in Carl Hewitt's *Planner* [23] where the term referred to rule-based active entities which search a knowledge base for patterns to match, and in response, trigger actions. For the next 2 decades, Hewitt's group worked on actors as



Actors. Fig. 1 Actors are concurrent objects that communicate through messages and may create new actors. An actor may be viewed as an object augmented with its own control, a *mailbox*, and a globally unique, immutable *name*

agents of computation, and it evolved as a model of concurrent computing. A brief history of actor research can be found in [7]. The commonly used definition of actors today follows the work of Agha (1985) which defines actors using a simple operational semantics [3].

In recent years, the Actor model has gained in popularity with the growth of parallel and distributed computing platforms such as multicore architectures, cloud computers, and sensor networks. A number of actor languages and frameworks have been developed. Some early actor languages include *ABCL*, *POOL*, *ConcurrentSmalltalk*, *ACT++*, *CEiffel* (see [11] for a review of these), and *HAL* [22]. Actor languages and frameworks in current use include *Erlang* (from Ericsson) [8], *E* (Erights.org), *Scala Actors* library (EPFL) [24], *Ptolemy* (UC Berkeley) [31], *ASP* (INRIA) [15], *JCoBox* (University of Kaiserslautern) [40], *SALSA* (UIUC and RPI) [42], *Charm++* [28] and *ActorFoundry* [10] (both from UIUC), the *Asynchronous Agents Library* [17] and *Axum* [18] (both from Microsoft), and *Orleans* framework for cloud computing from Microsoft Research [37]. Some well-known open source applications built using actors include Twitter's message queuing system and Lift Web Framework, and among commercial applications are Facebook Chat system and Vendetta's game engine.

Illustrative Actor Language

In order to show how actor programs work, consider a simple imperative actor language *ActorFoundry* that extends Java. A class defining an actor behavior extends `os1.manager.Actor`. Messages are handled by methods; such methods are annotated with `@message`. The `create(class,args)` method creates an actor instance of the specified actor class, where `args` correspond to the arguments of a constructor in the class. Each newly created actor has a unique name that is initially known only to the creator at the point where the creation occurs.

Execution Semantics

The semantics of *ActorFoundry* can be informally described as follows. Consider an *ActorFoundry* program *P* that consists of a set of actor definitions. An actor communicates with another actor in *P* by sending asynchronous (non-blocking) messages using the `send` statement: `send(a,msg)` has the effect of *eventually* appending the contents of `msg` to the mailbox of the actor *a*. However, the call to `send` returns immediately, that is, the sending actor does not wait for the message to arrive at its destination. Because actors operate asynchronously, and the network has indeterminate delays, the arrival order of messages is

nondeterministic. However, we assume that messages are *eventually* delivered (a form of *fairness*).

At the beginning of execution of P , the mailbox of each actor is empty and some actor in the program must receive a message from the environment. The Actor-Foundry runtime first creates an instance of a specified actor and then sends a specified message to it, which serves as P 's entry point.

Each actor can be viewed as executing a loop with the following steps: remove a message from its mailbox (often implemented as a queue), decode the message, and execute the corresponding method. If an actor's mailbox is empty, the actor blocks – waiting for the next message to arrive in the mailbox (such blocked actors are referred to as *idle actors*). The processing of a message may cause the actor's local state to be updated, new actors to be created, and messages to be sent. Because of the encapsulation property of actors, there is no interference between messages that are concurrently processed by different actors.

An actor program “terminates” when every actor created by the program is idle and the actors are not open to the environment (otherwise the environment could send new messages to their mailboxes in the future). Note that an actor program need not terminate – in particular, certain interactive programs and operating systems may continue to execute indefinitely.

Listing 1 shows the HelloWorld program in Actor-Foundry. The program comprises of two actor definitions: the HelloActor and the WorldActor. An instance of the HelloActor can receive one type of message, the greet message, which triggers the execution of greet method. The greet method serves as P 's entry point, in lieu of the traditional main method.

On receiving a greet message, the HelloActor sends a print message to the stdout actor (a built-in actor representing the standard output stream) along with the string “Hello.” As a result, “Hello” will eventually be printed on the standard output stream. Next, it creates an instance of the WorldActor. The HelloActor sends an audience message to the WorldActor, thus delegating the printing of “World” to it. Note that due to asynchrony in communication, it is possible for “World” to be printed before “Hello.”

Listing 1 Hello World! program in ActorFoundry

```
public class HelloActor extends Actor {
    @message
    public void greet() throws RemoteCodeException
    {
        ActorName other = null;
        send(stdout, "print", "Hello");
        other = create(WorldActor.class);
        send(other, "audience");
    }
}

public class WorldActor extends Actor {
    @message
    public void audience() throws RemoteCodeException
    {
        send(stdout, "print", "World");
    }
}
```

Synchronization

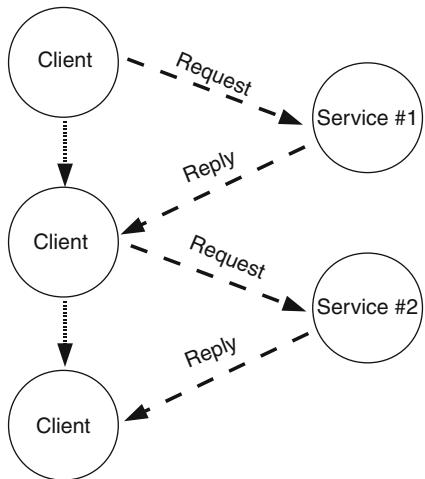
Synchronization in actors is achieved through communication. Two types of commonly used communication patterns are Remote Procedure Call (RPC)-like messaging and local synchronization constraints. Language constructs can enable actor programmers to specify such patterns. Such language constructs are definable in terms of primitive actor constructs, but providing them as first-class linguistic objects simplifies the task of writing parallel code.

RPC-Like Messaging

RPC-like communication is a common pattern of message-passing in actor programs. In RPC-like communication, the sender of a message waits for the reply to arrive before the sender proceeds with processing other messages. For example, consider the pattern shown in Fig. 2 for a client actor which requests a quote from a travel service. The client wishes to wait for the quote to arrive before it decides whether to buy the trip, or to request a quote from another service.

Without a high-level language abstraction to express RPC-like message pattern, a programmer has to explicitly implement the following steps in their program:

1. The client actor sends a request.
2. The client then checks incoming messages.



Actors. Fig. 2 A client actor requesting quotes from multiple competing services using RPC-like communication. The *dashed slanted arrows* denote messages and the *dashed vertical arrows* denote that the actor is waiting or is blocked during that period in its life

3. If the incoming message corresponds to the reply to its request, the client takes the appropriate action (accept the offer or keep searching).
4. If an incoming message does not correspond to the reply to its request, the message must be handled (e.g., by being buffered for later processing), and the client continues to check messages for the reply.

RPC-like messaging is almost universally supported in actor languages and libraries. RPC-like messages are particularly useful in two kinds of common scenarios. One scenario occurs when an actor wants to send an ordered sequence of messages to a particular recipient – in this case, it wants to ensure that a message has been received before it sends another. A variant of this scenario is where the sender wants to ensure that the target actor has received a message before it communicates this information to another actor. A second scenario is when the state of the requesting actor is dependent on the reply it receives. In this case, the requesting actor cannot meaningfully process unrelated messages until it receives a response.

Because RPC-like messages are similar to procedure calls in sequential languages, programmers have a tendency to overuse them. Unfortunately, inappropriate usage of RPC-like messages introduces unnecessary

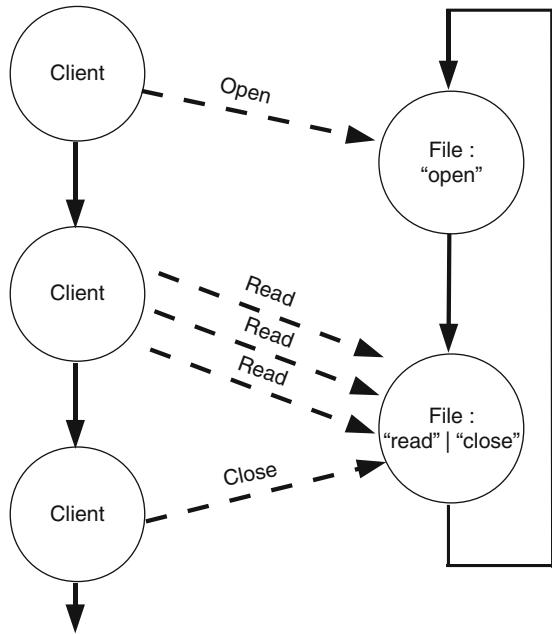
dependencies in the code. This may not only make the program’s execution more inefficient than it needs to be, it can lead to *deadlocks* and *livelocks* (where an actor ignores or postpones processing messages, waiting for an acknowledgment that never arrives).

Local Synchronization Constraints

Asynchrony is inherent in distributed systems and mobile systems. One implication of asynchrony is that the number of possible orderings in which messages may *arrive* is exponential in the number of messages that are “pending” at any time (i.e., messages that have been sent but have not been received). Because a sender may be unaware of what the state of the actor it is sending a message to will be when the latter receives the message, it is possible that the recipient may not be in a state where it can process the message it is receiving. For example, a spooler may not have a job when some printer requests one. As another example, messages to actors representing individual matrix elements (or groups of elements) asking them to process different iterations in a parallel Cholesky decomposition algorithm need to be monotonically ordered. The need for such orderings leads to considerable complexity in concurrent programs, often introducing bugs or inefficiencies due to suboptimal implementation strategies. For example, in the case of Cholesky decomposition, imposing a global ordering on the iterations leads to highly inefficient execution on multicomputers [6].

Consider the example of a print spooler. Suppose a ‘get’ message from an idle printer to its spooler may arrive when the spooler has no jobs to return the printer. One way to address this problem is for the spooler to refuse the request. Now the printer needs to repeatedly poll the spooler until the latter has a job. This technique is called *busy waiting*; busy waiting can be expensive – preventing the waiting actor from possibly doing other work while it “waits,” and it results in unnecessary message traffic. An alternate is to the spooler buffer the “get” message for deferred processing. The effect of such buffering is to change the order in which the messages are processed in a way that guarantees that the number of messages put messages to the spooler is always greater than the number of get messages processed by the spooler.

If pending messages are buffered explicitly inside the body of an actor, the code specifying the functionality



Actors. Fig. 3 A file actor communication with a client is constrained using local synchronization constraints. The *vertical arrows* depict the timeline of the life of an actor and the *slanted arrows* denote messages. The labels inside a circle denote the messages that the file actor can accept in that particular state

(the *how* or representation) of the actor is mixed with the logic determining the order in which the actor processes the messages (the *when*). Such mixing violates the software principle of *separation of concerns*. Researchers have proposed various constructs to enable programmers to specify the correct orderings in a modular and abstract way, specifically, as logical formulae (predicates) over the state of an actor and the type of messages. Many actor languages and frameworks provide such constructs; examples include local synchronization constraints in ActorFoundry, and pattern matching on sets of messages in Erlang and Scala Actors library.

Patterns of Actor Programming

Two common patterns of parallel programming are *pipeline* and *divide-and-conquer* [4]. These patterns are illustrated in Fig. 4a and 4b, respectively.

An example of the pipeline pattern is an image processing network (see Fig. 4a) in which a stream of

images is passed through a series of filtering and transforming stages. The output of the last stage is a stream of processed images. This pattern has been demonstrated by an image processing example, written using the *Asynchronous Agents Library* [17], which is part of the Microsoft Visual Studio 2010.

A *map-reduce* graph is an example of the divide-and-conquer pattern (see Fig. 4b). A master actor maps the computation onto a set of workers and the output from each of these workers is reduced in the “join continuation” behavior of the master actor (possibly modeled as a separate actor) (e.g., see [21]). Other examples of divide-and-conquer pattern are naïve parallel quicksort [38] and parallel mergesort. The synchronization idioms discussed above may be used in succinct encoding of these patterns in actor programs since these patterns essentially require ordering the processing of some messages.

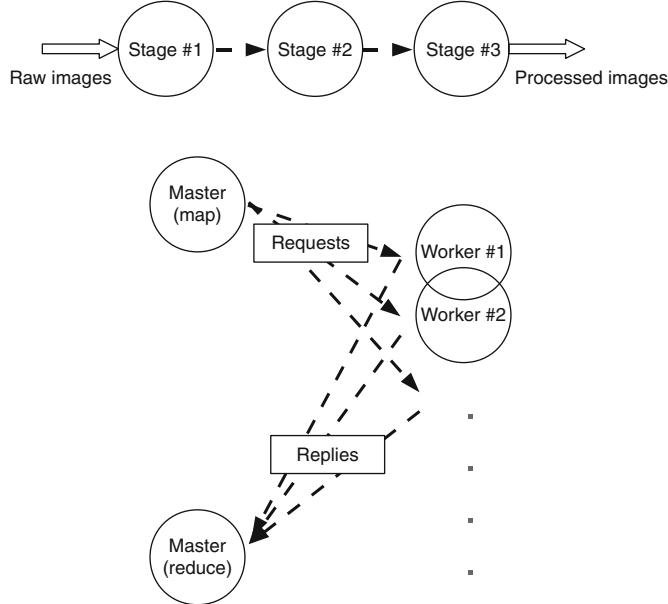
Semantic Properties

As mentioned earlier, some important semantic properties of the pure Actor model are encapsulation and atomic execution of methods (where a method represents computation in response to a message), fairness, and location transparency [29]. We discuss the implications of these properties.

Note that not all actor languages enforce all these properties. Often, the implementations compromise some actor properties, typically because it is simpler to achieve efficient implementations by doing so. However, it is possible by sophisticated program transformations, compilation, and runtime optimizations to regain almost all the efficiency that is lost in a naïve language implementation, although doing so is more challenging for library-like actor frameworks [29]. By failing to enforce some actor properties in an actor language or framework implementation, actor languages add to the burden of the programmers, who have to then ensure that they write programs in a way that does not violate the property.

Encapsulation and Atomicity

Encapsulation implies that no two actors share state. This is useful for enforcing an object-style decomposition in the code. In sequential object-based languages, this has led to the natural model of atomic change in objects: an object invokes (sends a message to) another



Actors. Fig. 4 Patterns of actor programming (from top): (a) pipeline pattern (b) divide-and-conquer pattern

object, which finishes processing the message before accepting another message from a different object. This allows us to reason about the behavior of the object in response to a message, given the state of the target object when it received the message. In a concurrent computation, it is possible for a message to arrive while an actor is busy processing another message. Now if the second message is allowed to interrupt the target actor and modify the target's state while the target is still processing the first message, it is no longer feasible to reason about the behavior of the target actor based on what the target's state was when it received the first message. This makes it difficult to reason about the behavior of the system as such interleaving of messages may lead to erroneous and inconsistent states.

Instead, the target actor processes messages one at a time, in a single *atomic step* consisting of all actions taken in response to a given message [7]. By dramatically reducing the nondeterminism that must be considered, such atomicity provides a *macro-step semantics* which simplifies reasoning about actor programs. Macro-step semantics is commonly used by correctness-checking tools; it significantly reduces the state-space exploration required to check a property against an actor program's potential executions (e.g., see [32]).

Fairness

The Actor model assumes a notion of *fairness* which states that every actor makes progress if it has some computation to do, and that every message is eventually delivered to the destination actor, unless the destination actor is permanently disabled. Fairness enables modular reasoning about the liveness properties of actor programs [7]. For example, if an actor system *A* is composed with an actor system *B* where *B* includes actors that are permanently busy, the composition does not affect the progress of the actors in *A*. A familiar example where fairness would be useful is in browsers. Problems are often caused by the composition of browser components with third-party plug-ins: in the absence of fairness, such plug-ins sometimes result in browser crashes and hang-ups.

Location Transparency

In the Actor model, the actual location of an actor does not affect its name. Actors communicate by exchanging messages with other actors, which could be on the same core, on the same CPU, or on another node in the network. Location transparent naming provides an abstraction for programmers, enabling them to program without worrying about the actual physical location of actors. Location transparent naming facilitates

automatic migration in the runtime, much as indirection in addressing facilitates compaction following garbage collection in sequential programming.

Mobility is defined as the ability of a computation to migrate across different nodes. Mobility is important for load-balancing, fault-tolerance, and reconfiguration. In particular, mobility is useful in achieving scalable performance, particularly for dynamic, irregular applications [26]. Moreover, employing different distributions in different stages of a computation may improve performance. In other cases, the optimal or correct performance depends on runtime conditions such as data and workload, or security characteristics of different platforms. For example, web applications may be migrated to servers or to mobile clients depending on the network conditions and capabilities of the client [12].

Mobility may also be useful in reducing the energy consumed by the execution of parallel applications. Different parts of an application often involve different parallel algorithms and the energy consumption of an algorithm depends on how many cores the algorithm is executed on and at what frequency these cores operate [27]. Mobility facilitates dynamic redistribution of a parallel computation to the appropriate number of cores (i.e., to the number of cores that minimize energy consumption for a given performance requirement and parallel algorithm) by migrating actors. Thus, mobility could be an important feature for energy-aware programming of multicore (*manycore*) architectures. Similarly, energy savings may be facilitated by being able to migrate actors in sensor networks and clouds.

Implementations

Erlang is arguably the best-known implementation of the Actor model. It was developed to program telecom switches at Ericsson about 20 years ago. Some recent actor implementations have been listed earlier. Many of these implementations have focused on a particular domain such as the Internet (SALSA), distributed applications (Erlang and E), sensor networks (ActorNet), and, more recently multicore processors (Scala Actors library, ActorFoundry, and many others in development).

It has been noted that a faithful but naïve implementation of the Actor model can be highly inefficient [29] (at least on the current generation of architectures). Consider three examples:

1. An implementation that maps each actor to a separate process may have a high cost for actor creation.
2. If the number of cores is less than the number of actors in the program (sometimes termed *CPU over-subscription*), an implementation mapping actors to separate processes may suffer from high context switching cost.
3. If two actors are located on the same sequential node, or on a shared-memory processor, it may be an order of magnitude more efficient to pass a reference to the message contents rather than to make a copy of the actual message contents.

These inefficiencies may be addressed by compilation and runtime techniques, or through a combination of the two. The implementation of the ABCI language [45] demonstrates some early ideas for optimizing both intra-node and internode execution and communication between actors. The Thal language project [26] shows that encapsulation, fairness, and universal naming in an actor language can be implemented efficiently on commodity hardware by using a combination of compiler and runtime. The Thal implementation also demonstrates that various communication abstractions such as RPC-like communication, local synchronization constraints, and join expressions can also be supported efficiently using various compile-time program transformations.

The Kilim framework develops a clever post-compilation continuation-passing style (CPS) transform (“weaving”) on Java-based actor programs for supporting lightweight actors that can pause and resume [39]. Kilim and Scala also add type systems to support safe but efficient messages among actors on a shared node [25, 39]. Recent work suggests that *ownership transfer* between actors, which enables safe and efficient messaging, can be statically inferred in most cases [33].

On distributed platforms such as cloud computers or grids, because of latency in sending messages to remote actors, an important technique for achieving good performance is communication–computation

overlap. Decomposition into actors and the placement of actors can significantly determine the extent of this overlap. Some of these issues have been effectively addressed in the Charm++ runtime [28]. Decomposition and placement issues are also expected to show up on scalable manycore architectures since these architecture cannot be expected to support constant time access to shared memory.

Finally, note that the notion of garbage in actors is somewhat complex. Because an actor name may be communicated in a message, it is not sufficient to mark the forward acquaintances (references) of *reachable* actors as reachable. The inverse acquaintances of reachable actors that may be potentially active need to be considered as well (these actors may send a message to a reachable actor). Efficient garbage collection of distributed actors remains an open research problem because of the problem of taking efficient distributed snapshots of the reachability graph in a running system [43].

Tools

Several tools are available to aid in writing, maintaining, debugging, model checking, and testing actor programs. Both Erlang and Scala have a plug-in for the popular, open source IDE (Integrated Development Environment) called Eclipse (<http://www.eclipse.org>). A commercial testing tool for Erlang programs called QuickCheck [5] is available. The tool enables programmers to specify program properties and input generators which are used to generate test inputs.

JCute [36] is a tool for automatic unit testing of programs written in a Java actor framework. Basset [30] works directly on executable (Java bytecode) actor programs and is easily retargetable to any actor language that compiles to bytecode. Basset understands the semantic structure of actor programs (such as the macro-step semantics), enabling efficient path exploration through the Java Pathfinder (JPF) – a popular tool for model checking programs [44]. The term rewriting system Maude provides an Actor module to specify program behavior; it has been used to model check actor programs [13]. There has also been work on runtime monitoring of actor programs [41].

Extensions and Abstractions

A programming language should facilitate the process of writing programs by being close to the conceptual level at which a programmer thinks about a problem rather than at the level at which it may be implemented. Higher level abstractions for concurrent programming may be defined in interaction languages which allow patterns to be captured as first-class objects [35]. Such abstractions can be implemented through an adaptive, reflective middleware [9]. Besides programming abstractions for concurrency in the pure (asynchronous) Actor model, there are variants of the Actor model, such as for real-time, which extend the model [31, 34]. Two interaction patterns are discussed to illustrate the ideas of interaction patterns.

Pattern-Directed Communication

Recall that a sending actor must know the name of a target actor before the sending actor can communicate with the target actor. This property, called *locality*, is useful for compositional reasoning about actor programs – if it is known that only some actors can send a message to an actor A, then it may be possible to figure out what types of messages A may get and perhaps specify some constraints on the order in which it may get them. However, real-world programs generally create an open system which interacts with their external environment. This means that having ways of discovering actors which provide certain services can be helpful. For example, if an actor migrates to some environment, discovering a printer in that environment may be useful.

Pattern-directed communication allows programmers to declare properties of a group of actors, enabling the use of the properties to discover actual recipients are chosen at runtime. In the *ActorSpace* model, an actor specifies recipients in terms of patterns over properties that must be satisfied by the recipients. The sender may send the message to all actors (in some group) that satisfy the property, or to a single representative actor [1]. There are other models for pattern-based communication. In *Linda*, potential recipients specify a pattern for messages they are interested in [14]. The sending actors simply inserts a message (called *tuple* in Linda) into

a *tuple-space*, from where the receiving actors may read or remove the tuples if the tuple matches the pattern of messages the receiving actor is interested in.

Coordination

Actors help simplify programming by increasing the granularity at which programmers need to reason about concurrency, namely, they may reason in terms of the potential interleavings of messages to actors, instead of in terms the interleavings of accesses to shared variables within actors. However, developing actor programs is still complicated and prone to errors. A key cause of complexity in actor programs is the large number of possible interleaving of messages to groups of actors: if these message orderings are not suitably constrained, some possible execution orders may fail to meet the desired specification.

Recall that local synchronization constraints postpone the dispatch of a message based on the contents of the messages and the local state of the receiving actor (see section “Local Synchronization Constraints”). *Synchronizers*, on the other hand, change the order in which messages are processed by a group of actors by defining constraints on ordering of messages processed at different actors in a group of actors. For example, if a withdrawal and deposit messages must be processed atomically by two different actors, a Synchronizer can specify that they must be scheduled together. Synchronizers are described in [2].

In the standard actor semantics, an actor that knows the name of a target actor may send the latter a message. An alternate semantics introduces the notion of a *channel*; a channel is used to establish communication between a given sender and a given recipient. Recent work on actor languages has introduced *stateful channel contracts* to constrain the order of messages between two actors. Channels are a central concept for communication between actors in both Microsoft’s Singularity platform [20] and Microsoft’s Axum language [18], while they can be optionally introduced between two actors in Erlang. Channel contracts specify a protocol that governs the communication between the two end points (actors) of the channel. The contracts are stated in terms of state transitions based on observing messages on the channel.

From the perspective of each end point (actor), the channel contract specifies the *interface* of the other end point (actor) in terms of not only the type of messages but also the ordering on messages. In Erlang, contracts are enforced at runtime, while in Singularity a more restrictive notion of typed contracts make it feasible to check the constraints at compile time.

Current Status and Perspective

Actor languages have been used for parallel and distributed computing in the real world for some time (e.g., Charm++ for scientific applications on supercomputers [28], Erlang for distributed applications [8]). In recent years, interest in actor-based languages has been growing, both among researchers and among practitioners. This interest is triggered by emerging programming platforms such as multicore computers and cloud computers. In some cases, such as cloud computing, web services, and sensor networks, the Actor model is a natural programming model because of the distributed nature of these platforms. Moreover, as multicore architectures are scaled, multicore computers will also look more and more like traditional multicomputer platforms. This is illustrated by the 48-core Single-Chip Cloud Computer (SCC) developed by Intel [16] and the 100-core TILE-Gx by Tilera [19]. However, the argument for using actor-based programming languages is not simply that they are a good match for distributed computing platforms; it is that Actors is a good model in which to think about concurrency. Actors simplify the task of programming by extending object-based design to concurrent (parallel, distributed, mobile) systems.

Bibliography

1. Agha G, Callsen CJ (1993) Actorspace: an open distributed programming paradigm. In: Proceedings of the fourth ACM SIGPLAN symposium on principles and practice of parallel programming (PPoPP), San Diego, CA. ACM, New York, pp 23–32
2. Agha G, Frolund S, Kim WY, Panwar R, Patterson A, Sturman D (1993) Abstraction and modularity mechanisms for concurrent computing. IEEE Trans Parallel Distr Syst 1(2):3–14
3. Agha G (1986) Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge, MA
4. Agha G (1990) Concurrent object-oriented programming. Commun ACM 33(9): 125–141

5. Arts T, Hughes J, Johansson J, Wiger U (2006) Testing telecoms software with quviq quickcheck. In: ERLANG '06: proceedings of the 2006 ACM SIGPLAN workshop on Erlang. ACM, New York, pp 2–10
6. Agha G, Kim WY (2002) Parallel programming and complexity analysis using actors. In: Proceedings: third working conference on massively parallel programming models, 1997, London. IEEE Computer Society Press, Los Alamitos, CA, pp 68–79
7. Agha G, Mason IA, Smith S, Talcott C (1997) A foundation for actor computation. *J Funct Program* 7(01):1–72
8. Armstrong J (2007) Programming Erlang: software for a concurrent World. Pragmatic Bookshelf, Raleigh, NC
9. Astley M, Sturman D, Agha G (2001) Customizable middleware for modular distributed software. *Commun ACM* 44(5):99–107
10. Astley M (1998–1999) The actor foundry: a java-based actor programming environment. Open Systems Laboratory, University of Illinois at Urbana-Champaign, Champaign, IL
11. Briot JP, Guerraoui R, Lohr KP (1998) Concurrency and distribution in object-oriented programming. *ACM Comput Surv* 30(3): 291–329
12. Chang PH, Agha G (2007) Towards context-aware web applications. In: 7th IFIP international conference on distributed applications and interoperable systems (DAIS), 2007, Paphos, Cyprus. LNCS 4531, Springer, Berlin
13. Clavel M, Durán F, Eker S, Lincoln P, Martí-Oliet N, Meseguer J, Talcott C (2007) All about maude – a high-performance logical framework: how to specify, program and verify systems in rewriting logic. Springer, Berlin
14. Carriero N, Gelertner D (1989) Linda in context. *Commun ACM* 32:444–458
15. Caromel D, Henrio L, Serpette BP (2009) Asynchronous sequential processes. *Inform Comput* 207(4):459–495
16. Intel Corporation. Single-chip cloud computer. <http://techresearch.intel.com/ProjectDetails.aspx?Id=1>
17. Microsoft Corporation. Asynchronous agents library. [http://msdn.microsoft.com/enus/library/dd492627\(VS.100\).aspx](http://msdn.microsoft.com/enus/library/dd492627(VS.100).aspx)
18. Microsoft Corporation. Axum programming language. <http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx>
19. Tilera Corporation. TILE-Gx processor family. <http://tilera.com/products/processors/TILE-Gxfamily>
20. Fähndrich M, Aiken M, Hawblitzel C, Hodson O, Hunt G, Larus JR, Levi S (2006) Language support for fast and reliable message-based communication in singularity OS. *SIGOPS Oper Syst Rev* 40(4):177–190
21. Feng TH, Lee EA (2008) Scalable models using model transformation. In: 1st international workshop on model based architecting and construction of embedded systems (ACESMB), Toulouse, France
22. Houck C, Agha G (1992) Hal: a high-level actor language and its distributed implementation. In: 21st international conference on parallel processing (ICPP), vol II, An Arbor, MI, pp 158–165
23. Hewitt C (1969) PLANNER: a language for proving theorems in robots. In: Proceedings of the 1st international joint conference on artificial intelligence, Morgan Kaufmann, San Francisco, CA, pp 295–301
24. Haller P, Odersky M (2007) Actors that unify threads and events. In: 9th International conference on coordination models and languages, vol 4467 of lecture notes in computer science, Springer, Berlin
25. Haller P, Odersky M (2010) Capabilities for uniqueness and borrowing. In: D'Hondt T (ed) ECOOP 2010 object-oriented programming, vol 6183 of lecture notes in computer science, Springer, Berlin, pp 354–378
26. Kim WY, Agha G (1995) Efficient support of location transparency in concurrent object-oriented programming languages. In: Supercomputing '95: proceedings of the 1995 ACM/IEEE conference on supercomputing, San Diego, CA. ACM, New York, p 39
27. Korthikanti VA, Agha G (2010) Towards optimizing energy costs of algorithms for shared memory architectures. In: SPAA '10: proceedings of the 22nd ACM symposium on parallelism in algorithms and architectures, Santorini, Greece. ACM, New York, pp 157–165
28. Kale LV, Krishnan S (1993) Charm++: a portable concurrent object oriented system based on c++. *ACM SIGPLAN Not* 28(10): 91–108
29. Karmani RK, Shali A, Agha G (2009) Actor frameworks for the JVM platform: a comparative analysis. In: PPP '09: proceedings of the 7th international conference on principles and practice of programming in java, Calgary, Alberta. ACM, New York, pp 11–20
30. Lauterburg S, Dotta M, Marinov D, Agha G (2009) A framework for state-space exploration of javabased actor programs. In: ASE '09: proceedings of the 2009 IEEE/ACM international conference on automated software engineering, Auckland, New Zealand. IEEE Computer Society, Washington, DC, pp 468–479
31. Lee EA (2003) Overview of the ptolemy project. Technical report UCB/ERL M03/25. University of California, Berkeley
32. Lauterburg S, Karmani RK, Marinov D, Agha G (2010) Evaluating ordering heuristics for dynamic partialorder reduction techniques. In: Fundamental approaches to software engineering (FASE) with ETAPS, 2010, LNCS 6013, Springer, Berlin
33. Negara S, Karmani RK, Agha G (2011) Inferring ownership transfer for efficient message passing. In: To appear in the 16th ACM SIGPLAN symposium on principles and practice of parallel programming (PPoPP). ACM, New York
34. Ren S, Agha GA (1995) Rtsynchronizer: language support for real-time specifications in distributed systems. *ACM SIGPLAN Not* 30(11):50–59
35. Sturman D, Agha G (1994) A protocol description language for customizing failure semantics. In: Proceedings of the thirteenth symposium on reliable distributed systems, Dana Point, CA. IEEE Computer Society Press, Los Alamitos, CA, pp 148–157
36. Sen K, Agha G (2006) Automated systematic testing of open distributed programs. In: Fundamental approaches to software engineering (FASE), volume 3922 of lecture notes in computer science, Springer, Berlin, pp 339–356
37. Kliot G, Larus J, Pandya R, Thelin J, Bykov S, Geller A (2010) Orleans: a framework for cloud computing. Technical report MSR-TR-2010-159, Microsoft Research

38. Singh V, Kumar V, Agha G, Tomlinson C (1991) Scalability of parallel sorting on mesh multicomputers. In: Parallel processing symposium, 1991. Proceedings, fifth international. Anaheim, CA, pp 92–101
39. Srinivasan S, Mycroft A (2008) Kilim: isolation typed actors for java. In: Proceedings of the European conference on object oriented programming (ECOOP), Springer, Berlin
40. Schäfer J, Poetzsch-Heffter A (2010) Jcobox: generalizing active objects to concurrent components. In: Proceedings of the 24th European conference on object-oriented programming, ECOOP'10, Maribor, Slovenia. Springer, Berlin/Heidelberg pp 275–299
41. Sen K, Vardhan A, Agha G, Rosu G (2004) Efficient decentralized monitoring of safety in distributed systems. In: ICSE '04: proceedings of the 26th international conference on software engineering, Edinburg, UK. IEEE Computer Society. Washington, DC, pp 418–427
42. Varela C, Agha G (2001) Programming dynamically reconfigurable open systems with SALSA. ACM SIGPLAN Notices 36(12): 20–34
43. Venkatasubramanian N, Agha G, Talcott C (1992) Scalable distributed garbage collection for systems of active objects. In: Bekkers Y, Cohen J (eds) International workshop on memory management, ACM SIGPLAN and INRIA, St. Malo, France. Lecture notes in computer science, vol 637, Springer, Berlin, pp 134–148
44. Visser W, Havelund K, Brat G, Park S (2000) Model checking programs. In: Proceedings of the 15th IEEE international conference on automated software engineering, ASE '00, Grenoble, France. IEEE Computer Society, Washington, DC, pp 3
45. Yonezawa A (ed) (1990) ABCL: an object-oriented concurrent system. MIT Press, Cambridge, MA

more efficient, and it is most often related to different speeds or overheads associated with the resources of the computing node that are required by the computing task.

Discussion

Introduction

In parallel computing environments, it may be more efficient to schedule a computing task on one computing node than on another. Such affinity of a specific task for a particular node can arise from many sources, where the goal of the scheduling policy is typically to optimize a functional of response time and/or throughput. For example, the affinity might be based on how fast the computing task can be executed on a computing node in an environment comprised of nodes with heterogeneous processing speeds. As another example, the affinity might concern the resources associated with the computing nodes such that each node has a collection of available resources, each task must execute on a node with a certain set of resources, and the scheduler attempts to maximize performance subject to these imposed system constraints.

Another form of affinity scheduling is based on the state of the memory system hierarchy. More specifically, it may be more efficient in parallel computing environments to schedule a computing task on a particular computing node than on any other if relevant data, code, or state information already resides in the caches or local memories associated with the node. This is the form most commonly referred to as affinity scheduling, and it is the primary focus of this entry. The use of such affinity information in general-purpose multiprocessor systems can often improve performance in terms of functionals of response time and throughput, particularly if this information is inexpensive to obtain and exploit. On the other hand, the performance benefits of this form of affinity scheduling often depend upon a number of factors and vary over time, and thus there is a fundamental scheduling trade-off between scheduling tasks where they execute most efficiently and keeping the workload shared among nodes.

Affinity scheduling in general-purpose multiprocessor systems will be next described within its historical

Affinity Scheduling

MARK S. SQUILLANTE
IBM, Yorktown Heights, NY, USA

Synonyms

Cache affinity scheduling; Resource affinity scheduling

Definition

Affinity scheduling is the allocation, or scheduling, of computing tasks on the computing nodes where they will be executed more efficiently. Such affinity of a task for a node can be based on any aspects of the computing node or computing task that make execution

context, followed by a brief discussion of the performance trade-off between affinity scheduling and load sharing.

General-Purpose Multiprocessor Systems

Many of the general-purpose parallel computing environments introduced in the 1980s were shared-memory multiprocessor systems of modest size in comparison with the parallel computers of today. Most of the general-purpose multiprocessor operating systems at the time implemented schedulers based on simple priority schemes that completely ignored the affinity a task might have for a specific processor due to the contents of processor caches. On the other hand, as processor cycle times improved at a much faster rate than main memory access times, researchers were observing the increasingly important relative performance impact of cache misses (refer to, e.g., [12]). The relative costs of a cache miss were also increasing due to other issues including cache coherency [2] and memory bus interference. As a matter of fact, the caches in one of the general-purpose multiprocessor systems at the time were not intended to reduce the memory access time at all, but rather to reduce memory bus interference and protect the bus from most processor-memory references (refer to, e.g., [23]).

In one of the first studies of such processor-cache affinity issues, Squillante and Lazowska [18, 19] considered the performance implications of scheduling based on the affinity of a task for the cache of a particular processor. The typical behaviors of tasks in general-purpose multiprocessor systems can include alternation between executing at a processor and releasing this processor either to perform I/O or synchronization operations (in which cases the task is not eligible for scheduling until completion of the operation) or because of quantum expiration or preemption (in which cases the task is suspended to allow execution of another task). Upon returning and being scheduled on a processor, the task may experience an initial burst of cache misses and the duration of this burst depends, in part, upon the number of blocks belonging to the task that are already resident in the cache of the processor. Continual increases in cache sizes at the time further suggested that a significant portion of the working set of a task may reside in the cache of a specific processor under these scenarios. Scheduling decisions that disregard such cache reload

times may cause significant increases in the execution times of individual tasks as well as reductions in the performance of the entire system. These cache-reload effects may be further compounded by an increase in the overhead of cache coherence protocols (due to a larger number of cache invalidations resulting from modification of a task's data still resident in another processor's cache) and an increase in bus traffic and interference (due to cache misses).

To this end, Squillante and Lazowska [18, 19] formulate and analyze mathematical models to investigate fundamental principles underlying the various performance trade-offs associated with processor-cache affinity scheduling. These mathematical models represent general abstractions of a wide variety of general-purpose multiprocessor systems and workloads, ranging from more traditional time-sharing parallel systems and workloads even up through more recent dynamic coscheduling parallel systems and workloads (see, e.g., [21]). Several different scheduling policies are considered, spanning the entire spectrum from ignoring processor-cache affinity to fixing tasks to execute on specific processors. The results of this modeling analysis illustrate and quantify the benefits and limitations of processor-cache affinity scheduling in general-purpose shared-memory multiprocessor systems with respect to improving and degrading the first two statistical moments of response time and throughput. In particular, the circumstances under which performance improvement and degradation can be realized and the importance of exploiting processor-cache affinity information depend upon many factors. Some of the most important of these factors include the size of processor caches, the locality of the task memory references, the size of the set of cache blocks in active use by the task (its cache footprint), the ratio of the cache footprint loading time to the execution time of the task per visit to a processor, the time spent non-schedulable by the task between processor visits, the processor activities in between such visits, the system architecture, the parallel computing workload, the system scheduling strategy, and the need to adapt scheduling decisions with changes in system load.

A large number of empirical research studies subsequently followed to further examine affinity scheduling across a broad range of general-purpose parallel computing systems, workloads, scheduling strategies,

and resource types. Gupta et al. [10] use a detailed multiprocessor simulator to evaluate various scheduling strategies, including gang scheduling (coscheduling), two-level scheduling (space-sharing) with process control, and processor-cache affinity scheduling, with a focus on the performance impact of scheduling on cache behavior. The benefits of processor-cache affinity scheduling are shown to exhibit a relatively small but noticeable performance improvement, which can be explained by the factors identified above under the multiprocessor system and workload studied in [10]. Two-level scheduling with process control and gang scheduling are shown to provide the highest levels of performance, with process control outperforming gang scheduling when applications have large working sets that fit within a cache. Vaswani and Zahorjan [27] then developed an implementation of a space-sharing strategy (under which processors are partitioned among applications) to examine the implications of cache affinity in shared-memory multiprocessor scheduling under various scientific application workloads. The results of this study illustrate that processor-cache affinity within such a space-sharing strategy provides negligible performance improvements for the three scientific applications considered. It is interesting to note that the models in [18, 19], parameterized by the system and application measurements and characteristics in [27], also show that affinity scheduling yields negligible performance benefits in these circumstances. Devarakonda and Mukherjee [7] consider various implementation issues involved in exploiting cache affinity to improve performance, arguing that affinity is most effective when implemented through a thread package which supports the multiplexing of user-level threads on operating system kernel-level threads. The results of this study show that a simple scheduling strategy can yield significant performance improvements under an appropriate application workload and a proper implementation approach, which can once again be explained by the factors identified above. Torrellas et al. [25, 26] study the performance benefits of cache-affinity scheduling in shared-memory multiprocessors under a wide variety of application workloads, including various scientific, software development, and database applications, and under a time-sharing strategy that was in widespread use by the vast majority of parallel computing environments at the time. The authors conclude

that affinity scheduling yields significant performance improvements for a few of the application workloads and moderate performance gains for most of the application workloads considered in their study, while not degrading the performance of the rest of these workloads. These results can once again be explained by the factors identified above.

In addition to affinity scheduling in general-purpose shared-memory multiprocessors, a number of related issues have arisen with respect to other types of resources in parallel systems. One particularly interesting application area concerns affinity-based scheduling in the context of parallel network protocol processing, as first considered by Salehi et al. (see [16] and the references cited therein), where parallel computation is used for protocol processing to support high-bandwidth, low-latency networks. In particular, Salehi et al. [16] investigate affinity-based scheduling issues with respect to: supporting a large number of streams concurrently; receive-side and send-side protocol processing (including data-touching protocol processing); stream burstiness and source locality of network traffic; and improving packet-level concurrency and caching behavior. The approach taken to evaluate various affinity-based scheduling policies is based on a combination of multiprocessor system measurements, analytic modeling, and simulation. This combination of methods is used to illustrate and quantify the potentially significant benefits and effectiveness of affinity-based scheduling in multiprocessor networking. Across all of the workloads considered, the authors find benefits in managing threads and free-memory by taking affinity issues into account, with different forms of affinity-based scheduling performing best under different workload conditions.

The trends of increasing cache and local memory sizes and of processor speeds decreasing at much faster rates than memory access times continued to grow over time. This includes generations of non-uniform memory-access (NUMA) shared-memory and distributed-memory multiprocessor systems in which the remoteness of memory accesses can have an even more significant impact on performance. The penalty for not adhering to processor affinities can be considerably more significant in such NUMA and distributed-memory multiprocessor systems, where the actual cause of the larger costs depends upon the memory

management policy but are typically due to remote access or demand paging of data stored in non-local memory modules. These trends in turn have caused the performance benefits of affinity scheduling in parallel computing environments to continue to grow. The vast majority of parallel computing systems available today, therefore, often exploit various forms of affinity scheduling throughout distinct aspects of the parallel computing environment. There continues to be important differences, however, among the various forms of affinity scheduling depending upon the system architecture, application workload, and scheduling strategy, in addition to new issues arising from some more recent trends such as power management.

Affinity Scheduling and Load Sharing Trade-off

In parallel computing environments that employ affinity scheduling, the system often allocates computing tasks on the computing nodes where they will be executed most efficiently. Conversely, underloaded nodes are often inevitable in parallel computing environments due to factors such as the transient nature of system load and the variability of task service times. If computing tasks are always executed on the computing nodes for which they have affinity, then the system may suffer from load sharing problems as tasks are waiting at overloaded nodes while other nodes are underloaded. On the one hand, if processor affinities are not followed, then the system may incur significant penalties as each computing task must establish its working set in close proximity to a computing node before it can proceed. On the other hand, scheduling decisions cannot be based solely on task-node affinity, else other scheduling criteria, such as fairness, may be sacrificed. Hence, there is a fundamental scheduling trade-off between keeping the workload shared among nodes and scheduling tasks where they execute most efficiently. An adaptive scheduling policy is needed that determines, as a function of system load, the appropriate balance between the extremes of strictly balancing the workload among all computing nodes and abiding by task-node affinities blindly.

One such form of load sharing has received considerable attention with respect to distributed system environments. Static policies, namely those that use

information about the average behavior of the system while ignoring the current state, have been proposed and studied by numerous researchers including Bokhari [4], and Tantawi and Towsley [22]. These policies have been shown, in many cases, to provide better performance than policies that do not attempt to share the system workload. Other studies have shown that more adaptive policies, namely those that make decisions based on the current state of the system, have the potential to greatly improve system performance over that obtained with static policies. Furthermore, Livny and Melman [14], and Eager, Lazowska, and Zahorjan [8] have shown that much of this potential can be realized with simple methods. This potential has prompted a number of studies of specific adaptive load sharing policies, including the research conducted by Barak and Shiloh [3], Wang and Morris [28], Eager et al. [9], and Mirchandaney et al. [15].

While there are many similarities between the affinity scheduling and load sharing trade-off in distributed and shared-memory (as well as some distributed-memory) parallel computing environments, there can be several important differences due to distinct characteristics of these diverse parallel system architectures. One of the most important differences concerns the direct costs of moving a computing task. In a distributed system, these costs are typically incurred by the computing node from which the task is being migrated, possibly with an additional network delay. Subsequent to this move, the processing requirements of the computing task are identical to those at the computing node where the task has affinity. On the other hand, the major costs of task migration in a shared-memory (as well as distributed-memory) system are the result of a larger service demand at the computing node to which the computing task is migrated, reflecting the time required to either establish the working set of the task in closer proximity to this node or remotely access the working set from this node. Hence, there is a shift of the direct costs of migration from the (overloaded) computing node for which the computing task has affinity to the (underloaded) node receiving the task.

Motivated by these and related differences between distributed and shared-memory systems, Squillante and Nelson [20] investigated the fundamental trade-off between affinity scheduling and load sharing in

shared-memory (as well as some distributed-memory) parallel computing environments. More specifically, the authors consider the question of how expensive task migration must become before it is not beneficial to have an underloaded computing node migrate a computing task waiting at another node, as it clearly would be beneficial to migrate such a task if the cost to do so was negligible. Squillante and Nelson [20], therefore, formulate and analyze mathematical models to investigate this and related questions concerning the conditions under which it becomes detrimental to migrate a task away from its affinity node with respect to the costs of not adhering to these task-node affinities. The results of this modeling analysis illustrate and quantify the potentially significant benefits of migrating waiting tasks to underloaded nodes in shared-memory multiprocessors even when migration costs are relatively large, particularly at moderate to heavy loads. By sharing the collection of computing tasks among all computing nodes, a combination of affinity scheduling and threshold-based task migration can yield performance that is better than a non-migratory policy even with a larger service demand for migrated tasks, provided proper threshold values are employed. These modeling analysis results also demonstrate the potential for unstable behavior under task migration policies when improper threshold settings are employed, where optimal policy thresholds avoiding such behavior are provided as a function of system load and the relative processing time of migrated tasks.

Related Entries

- [Load Balancing, Distributed Memory](#)
- [Operating System Strategies](#)
- [Scheduling Algorithms](#)

Bibliographic Notes and Further Reading

Affinity scheduling based on the different speeds of executing tasks in heterogeneous processor environments has received considerable attention in the literature, including both deterministic models and stochastic models for which, as examples, the interested reader is referred to [5, 11] and [13, 30], respectively. Affinity scheduling based on resource requirement constraints have also received considerable

attention in the literature (see, e.g., [5, 6, 29]). Examples of shared memory multiprocessor systems from the 1980s include the DEC Firefly [23] and the Sequent Symmetry [17]. Multiprocessor operating systems of this period that completely ignored processor-cache affinity include Mach [1], DYNIX [24], and Topaz [23].

This entry primarily focuses on affinity scheduling based on the state of the memory system hierarchy where it can be more efficient to schedule a computing task on a particular computing node than on another if any relevant information already resides in caches or local memories in close proximity to the node. This entry also considers the fundamental trade-off between affinity scheduling and load sharing. A number of important references have been provided throughout, to which the interested reader is referred together with the citations provided therein. Many more references on these subjects are widely available, in addition to the wide variety of strategies that have been proposed to address affinity scheduling and related performance trade-offs.

Bibliography

1. Accetta M, Baron R, Bolosky W, Golub D, Rashid R, Tevanian A, Young M (1986) Mach: a new kernel foundation for UNIX development. In: Proceedings of USENIX Association summer technical conference, Atlanta, GA, June 1986. USENIX Association, Berkeley, pp 93–112
2. Archibald J, Baer J-L (1986) Cache coherence protocols: evaluation using a multiprocessor simulation model. ACM Trans Comput Syst 4(4):273–298
3. Barak A, Shiloh A (1985) A distributed load-balancing policy for a multicomputer. Softw Pract Exper 15(9):901–913
4. Bokhari SH (1979) Dual processor scheduling with dynamic reassignment. IEEE Trans Softw Eng SE-5(4):341–349
5. Conway RW, Maxwell WL, Miller LW (1967) Theory of scheduling. Addison-Wesley, Reading
6. Craft DH (1983) Resource management in a decentralized system. In: Proceedings of symposium on operating systems principles, October 1983. ACM, New York, pp 11–19
7. Devarakonda M, Mukherjee A (1992) Issues in implementation of cache-affinity scheduling. In: Proceedings of winter USENIX conference, January 1992. USENIX Association, Berkeley, pp 345–357
8. Eager DL, Lazowska ED, Zahorjan J (1986) Adaptive load sharing in homogeneous distributed systems. IEEE Trans Softw Eng SE-12(5):662–675
9. Eager DL, Lazowska ED, Zahorjan J (1986) A comparison of receiver-initiated and sender-initiated adaptive load sharing. Perform Evaluation 6(1):53–68

10. Gupta A, Tucker A, Urushibara S (1991) The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In: Proceedings of ACM SIGMETRICS conference on measurement and modeling of computer systems, May 1991. ACM, New York, pp 120–132
11. Horowitz E, Sahni S (1976) Exact and approximate algorithms for scheduling nonidentical processors. *J ACM* 23(2):317–327
12. Jouppi NP, Wall DW (1989) Available instruction-level parallelism for superscalar and superpipelined machines. In: Proceedings of international symposium on computer architecture, April 1989. ACM Press, New York, pp 272–282
13. Lin W, Kumar PR (1984) Optimal control of a queueing system with two heterogeneous servers. *IEEE Trans Automatic Contr* 29(8):696–703
14. Livny M, Melman M (1982) Load balancing in homogeneous broadcast distributed systems. In: Proceedings of ACM computer network performance symposium. ACM Press, New York, pp 47–55
15. Mirchananey R, Towsley D, Stankovic JA (1990) Adaptive load sharing in heterogeneous systems. *J Parallel Distrib Comput* 9:331–346
16. Salehi JD, Kurose JF, Towsley D (1996) The effectiveness of affinity-based scheduling in multiprocessor networking (extended version). *IEEE/ACM Trans Netw* 4(4):516–530
17. Sequent Computer Systems (1988) Symmetry technical summary. Sequent Computer Systems Inc, Beaverton
18. Squillante MS, Lazowska ED (1990) Using processor-cache affinity information in shared-memory multiprocessor scheduling. Technical Report 89-06-01, Department of Computer Science, University of Washington, June 1989. Minor revision, Feb 1990
19. Squillante MS, Lazowska ED (1993) Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Trans Parallel Distrib Syst* 4(2):131–143
20. Squillante MS, Nelson RD (1991) Analysis of task migration in shared-memory multiprocessors. In: Proceedings of ACM SIGMETRICS conference on measurement and modeling of computer systems, May 1991. ACM, New York, pp 143–155
21. Squillante MS, Zhang Y, Sivasubramaniam A, Gautam N, Franke H, Moreira J (2002) Modeling and analysis of dynamic coscheduling in parallel and distributed environments. In: Proceedings of ACM SIGMETRICS conference on measurement and modeling of computer systems, June 2002. ACM, New York, pp 43–54
22. Tantawi AN, Towsley D (1985) Optimal static load balancing in distributed computer systems. *J ACM* 32(2):445–465
23. Thacker C, Stewart LC, Satterthwaite EH Jr (1988) Firefly: a multiprocessor workstation. *IEEE Trans Comput C-37(8)*:909–920
24. Thakkar SS, Gifford PR, Fieland GF (1988) The balance multiprocessor system. *IEEE Micro* 8(1):57–69
25. Torrellas J, Tucker A, Gupta A (1993) Benefits of cache-affinity scheduling in shared-memory multiprocessors: a summary. In: Proceedings of ACM SIGMETRICS conference on measurement and modeling of computer systems, May 1993. ACM, New York, pp 272–274
26. Torrellas J, Tucker A, Gupta A (1995) Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors. *J Parallel Distrib Comput* 24(2):139–151
27. Vaswani R, Zahorjan J (1991) The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In: Proceedings of symposium on operating systems principles, October 1991. ACM, New York, pp 26–40
28. Wang YT, Morris R (1985) Load sharing in distributed systems. *IEEE Trans Comput C-34(3)*:204–217
29. Weinrib A, Gopal G (1987) Decentralized resource allocation for distributed systems. In: Proceedings of IEEE INFOCOM '87, San Francisco, April 1987. IEEE, Washington, DC, pp 328–336
30. Yu OS (1974) Stochastic bounds for heterogeneous-server queues with Erlang service times. *J Appl Probab* 11:785–796

Ajtai–Komlós–Szemerédi Sorting Network

► AKS Network

AKS Network

JOEL SEIFERAS

University of Rochester, Rochester, NY, USA

Synonyms

[Ajtai–Komlós–Szemerédi sorting network](#); [AKS sorting network](#); [Logarithmic-depth sorting network](#)

Definition

AKS networks are $\mathcal{O}(\log n)$ -depth networks of 2-item sorters that sort their n input items by following the 1982 design by Miklós Ajtai, János Komlós, and Endre Szemerédi.

Discussion

Comparator Networks for Sorting

Following Knuth [9, Section 5.3.4] or Cormen et al. [8, Chapter 27], consider algorithms that reorder their input sequences I (of items, or “keys,” from some totally ordered universe) via “oblivious comparison-exchanges.” Each comparison-exchange is effected by a “comparator” from a data position i to another data

position j , which permutes $I[i]$ and $I[j]$ so that $I[i] \leq I[j]$. (So a comparator amounts to an application of a 2-item sorter.) The sequence of such comparison-exchanges depends only on the number n of input items, but not on whether items get switched. Independent comparison-exchanges, involving disjoint pairs of data positions, can be allowed to take place at the same time – up to $\lfloor n/2 \rfloor$ comparison-exchanges per parallel step.

Each such algorithm can be reformulated so that every comparison-exchange $[i : j]$ has $i < j$ (Knuth's standard form) [9, Exercise 5.3.4-16, or 8, Exercise 27.1-8]. Following Knuth, restrict attention to such algorithms and represent them by left-to-right parallel horizontal “time lines” for the n data positions (also referred to as *registers*), connecting pairs of them by vertical lines to indicate comparison-exchanges. A famous example is Batcher's family of networks for sorting “bitonic” sequences [5]. The network for $n = 16$ is depicted in Fig. 1. Since the maximum depth is 4, four steps suffice. In general, Batcher's networks sort bitonic input sequences of lengths n that are powers of 2, in $\log_2 n$ steps.

If such a network reorders *every* length- n input sequence into sorted order (i.e., so that $I[i] \leq I[j]$ holds whenever $i < j$ does), then call it a *sorting* network. Since the concatenation of a sorted sequence and the reverse of a sorted sequence is bitonic, Batcher's networks can be used to merge sorted sequences in $\Theta(\log n)$ parallel steps (using $\Theta(n \log n)$ comparators), and hence to implement merge sort in $\Theta(\log^2 n)$ parallel steps (using $\Theta(n \log^2 n)$ comparators).

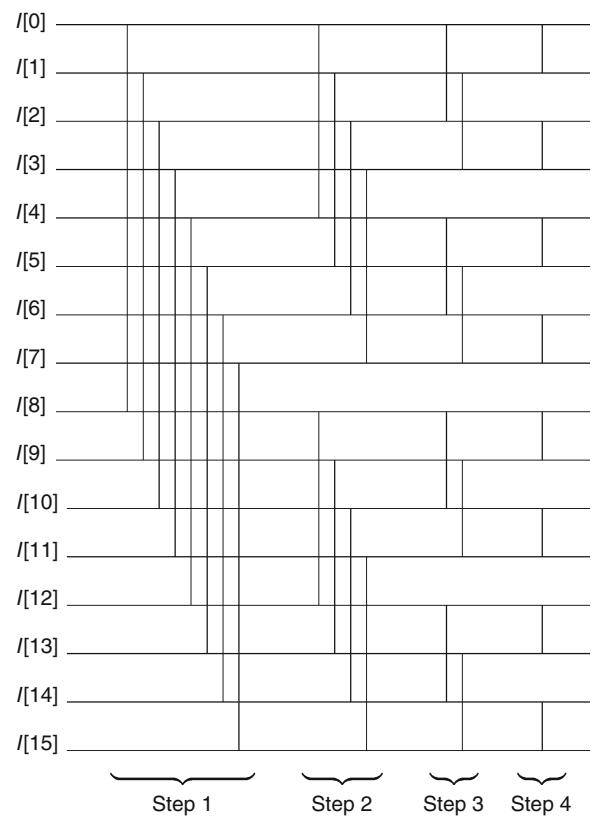
The total number $\Theta(n \log^2 n)$ of comparators used by Batcher's elegant sorting networks is worse by a factor of $\Theta(\log n)$ than the number $\Theta(n \log n) = \Theta(\log_2 n!)$ of two-outcome comparisons required for the best (nonoblivious) sorting algorithms, such as merge sort and heap sort [8, 9]. At the expense of simplicity and practical multiplicative constants, the “AKS” sorting networks of Ajtai, Komlós, and Szemerédi [1–2] close this gap, sorting in $\mathcal{O}(\log n)$ parallel steps, using $\mathcal{O}(n \log n)$ comparators.

The Sorting-by-Splitting Approach

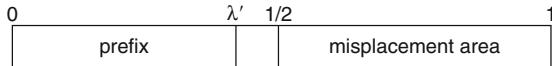
The AKS algorithm is based on a “sorting-by-splitting” or “sorting-by-classifying” approach that amounts to an ideal version of “quicksort” [8, 9]: Separate the items to be sorted into a smallest half and a largest half, and

continue recursively on each half. But the depth of just a halving network would already have to be $\Omega(\log n)$, again seeming to lead, through recursion, only to sorting networks of depth $\Omega(\log^2 n)$.

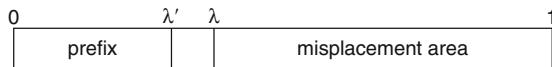
(To see that (all) the outputs of a perfect halver have to be at depth $\Omega(\log n)$, consider the computation on a sequence of n identical items, and consider any particular output item. If its depth is only d , then there are $n - 2^d$ input items that can have no effect on that output item. If this number exceeds $n/2$, then these uninvolved input items can all be made larger or smaller to make the unchanged output *wrong*. (In fact, a slightly more careful argument by Alekseev [4, 9] shows also that the total number of comparators has to be $\Omega(n \log n)$, again seeming to leave no room for a good recursive result.))



AKS Network. Fig. 1 Batcher's bitonic merger for $n = 16$ elements. Each successive comparator (vertical line) reorders its inputs (top and bottom endpoints) into sorted order



AKS Network. Fig. 2 Setting for approximate halving (prefix case)



AKS Network. Fig. 3 Setting for approximate λ -separation (prefix case)

The 1982, AKS breakthrough [1] was twofold: to notice that there *are* shallow *approximate* separators, and to find a way to tolerate their errors in sorting by classification.

Approximate Separation via Approximate Halving via Bipartite Expander Graphs

The approximate *separators* defined, designed, and used are based on the simpler definition and existence of approximate *halvers*. The criterion for approximate halving is a relative bound (ε) on the number of misplacements from each prefix or suffix (of relative length λ' up to $\lambda = 1/2$) of a completely sorted result to the wrong *half* of the result actually produced (see Fig. 2). For approximate *separation*, the “wrong fraction” one-half is generalized to include even larger fractions $1 - \lambda$. (See Fig. 3.)

Definition 1 For each $\varepsilon > 0$ and $\lambda \leq 1/2$, the criterion for ε -approximate λ -separation of a sequence of n input elements is that, for each $\lambda' \leq \lambda$, at most $\varepsilon\lambda'n$ of the $\lfloor \lambda'n \rfloor$ smallest (respectively, largest) elements do not get placed among the $\lfloor \lambda n \rfloor$ first (respectively, last) positions. For $\lambda = 1/2$, this is called ε -halving.

Although these definitions do not restrict n , they will be needed and used only for *even* n .

The AKS construction will use constant-depth networks that perform *both* ε -halving and ε -approximate λ -separation for some λ *smaller* than $1/2$. Note that neither of these quite implies the other, since the former constrains more prefixes and suffixes (all the way up to half the length), while the latter counts more misplacements (the ones to a longer “misplacement area”) as significant.

Lemma 1 For each $\varepsilon > 0$, ε -halving can be performed by comparator networks (one for each even n) of constant depth (depending on ε , but independent of n).

Proof From the study of “expander graphs” [14, e.g.], using the fact that $\frac{1-\varepsilon}{\varepsilon}\varepsilon < 1$, start with a constant d , determined by ε , and a d -regular $n/2$ -by- $n/2$ bipartite graph with the following expansion property: Each subset S of either part, with $|S| \leq \varepsilon n/2$, has more neighbors than $\frac{1-\varepsilon}{\varepsilon}|S|$.

The ε -halver uses a first-half-to-second-half comparator corresponding to each edge in the d -regular bipartite expander graph. This requires depth only d , by repeated application of a minimum-cut argument, to extract d matchings [8, Exercise 26.3-5, for example].

To see that the result is indeed an ε -halver, consider, in any application of the network, the final positions of the strays from the $m \leq n/2$ smallest elements. Those positions *and all their neighbors* must finally contain elements among the m smallest. If the fraction of strays were more than ε , then this would add up to more than $\varepsilon m + \frac{1-\varepsilon}{\varepsilon}\varepsilon m = m$, a contradiction. \square

Using approximate halvers in approximate separation and exact sorting resembles such mundane tasks as sweeping dirt with an imperfect broom or clearing snow with an imperfect shovel. There has to be an efficient, converging strategy for cleaning up the relatively few imperfections. Subsequent strokes should be aimed at where the concentrations of imperfections are currently known to lie, rather than again at the full job – something like skimming spilled oil off the *surface* of the Gulf of Mexico.

The use of approximate halvers in approximate separation involves relatively natural shrinkage of “stroke size” to sweep more extreme keys closer to the ends.

Lemma 2 For each $\varepsilon > 0$ and $\lambda \leq 1/2$, ε -approximate λ -separation and simultaneous ε -halving can be performed by comparator networks (one for each even n) of constant depth (depending on ε and λ , but independent of n).

Proof For ε_0 small in terms of ε and λ , make use of the ε_0 -halvers already provided by Lemma 1 (the result for $\lambda = 1/2$):

First apply the ε_0 -halver for length n to the whole sequence, and then work separately on each resulting half, so that the final result will remain an ε_0 -halver.

The halves are handled symmetrically, in parallel; so focus on the first half. In terms of $m = \lfloor \lambda n \rfloor$ (assuming $n \geq 1/\lambda$), apply ε_0 -halvers to the $\lceil \log_2(n/m) \rceil - 1$ prefixes of lengths $2m, 4m, 8m, \dots, 2^{\lceil \log_2(n/m) - 1 \rceil} m$, in reverse order, where the last one listed (first one performed) is simplified as if the inputs beyond the first $n/2$ were all some very large element. Then the total number of elements from the smallest $\lfloor \lambda' n \rfloor$ (for any $\lambda' \leq \lambda$) that do not end up among the first $\lfloor \lambda n \rfloor = m$ positions is at most $\varepsilon_0 \lambda' n$ in each of the $\lceil \log_2(n/m) \rceil$ intervals $(m, 2m], (2m, 4m], (4m, 8m], \dots, (2^{\lceil \log_2(n/m) - 2 \rceil} m, n/2],$ and $(n/2, n]$, for a total of at most $\lceil \log_2(n/m) \rceil \varepsilon_0 \lambda' n$.

For any chosen $c < 1$ (close to 1, making $\log_2(1/c)$ close to 0), the following holds: Unless n is small ($n < (1/\lambda)(1/(1-c))$),

$$\begin{aligned} 2^{1+\log_2(n/m)} &= 2(n/m) = 2n/\lfloor \lambda n \rfloor < 2n/(\lambda n - 1) \\ &\leq 2n/(c\lambda n) = (2/c)(1/\lambda), \end{aligned}$$

so that $\log_2(n/m)$ is less than $\log_2(1/c) + \log_2(1/\lambda)$, and the number of misplaced elements above is at most $\lceil \log_2(1/c) + \log_2(1/\lambda) \rceil \varepsilon_0 \lambda' n$. This is bounded by $\varepsilon \lambda' n$ as required, provided $\varepsilon_0 \leq \varepsilon / [\log_2(1/c) + \log_2(1/\lambda)]$. \square

Sorting with Approximate Separators

Without loss of generality, assume the n items to be sorted are distinct, and that n is a power of 2 larger than 1. To keep track of the progress of the recursive classification, and the (now not necessarily contiguous) sets of registers to which approximate halvers and separators should be applied, consider each of the registers always to occupy one of $n-1 = 1+2+4+\dots+n/2$ bags – one bag corresponding to each nonsingleton *binary subinterval* of the length- n index sequence: one whole, its first and second halves (two halves), their first and second halves (four quarters), ..., their first and second halves ($n/2$ contiguous pairs). These binary subintervals, and hence the associated bags, correspond nicely to the nodes of a complete binary tree, with each nontrivial binary subinterval serving as *parent* for its first and second halves, which serve respectively as its *left child* and *right child*. (The sets, and even the numbers, of registers corresponding to these bags will change with time, according to time-determined baggings and rebaggings by the algorithm.)

Based on its actual *rank* (position in the actual sorted order), each register's current item can be considered *native* to one bag at each level of the tree. For example, if it lies in the second quartile, it is native to the second of the four bags that are grandchildren of the root. Sometimes an item will occupy a bag to which it is *not native*, where it is a *stranger*; more specifically, it will be *j-strange* if its bag is *at least j* steps off its native path down from the root. (So the 1-strangers are *all* the strangers; and the *additional* 0-strangers are actually native, and not strangers at all.)

Initially, consider all n registers to be in the root bag (the one corresponding to the whole sequence of indices), to which all contents are native. The strategy is to design and follow a schedule, oblivious to the particular data, of applications of certain comparator networks and conceptual rebaggings of the results, that is guaranteed to leave all items in the bags of constant-height subtrees to which they are native. Then to finish, it will suffice to apply a separate sorting network to the $\mathcal{O}(1)$ registers in the bags of each of these constant-size subtrees.

The Structure of the Bagging Schedule

Each stage of the network acts separately on the contents of each nonempty bag, which is an inductively predictable subsequence of the n registers. In terms of the bag's current "capacity" b (an upper bound on its number of registers), a certain fixed "skimming" fraction λ , and other parameters to be chosen later (in retrospect), it applies an approximate separator from Lemma 2 to that sequence of registers, and it evacuates the results to the parent and children bags as follows: If there is a parent, then "kick back" to it $\lfloor \lambda b \rfloor$ items (or as many as are available, if there are not that many) from each end of the results, where too-small and too-large items will tend to accumulate. If the excess (between these ends) is odd (which will be inductively impossible at the root), then kick back any one additional register (the middle one, say) to the parent. Send the first and second halves of any remaining excess down to the respective children. Note that this plan can fail to be feasible in only one case: the number of registers to be evacuated exceeds $2\lfloor \lambda b \rfloor + 1$, but the bag has no children (i.e., it is a leaf).

The parameter b is an imposed *capacity* that will increase (exponentially) with the depth of the bag in the tree but decrease (also exponentially) with time, thus

“squeezing” all the items toward the leaves, as desired. Aim to choose the parameters so that the squeezing is slow enough that the separators from Lemma 2 have time to successfully skim and reroute all strangers back to their native paths.

To complete a (not yet proved) *description* of a network of depth $\mathcal{O}(\log n)$ to sort n items, here is a preview of one set of adequate parameters:

$$\lambda = \varepsilon = 1/99 \text{ and } b = n \cdot 10^d (.65)^t,$$

where $d \geq 0$ is the depth and $t \geq 0$ is the number of previous stages. It turns out that adopting these parameters enables iteration until $b_{\text{leaf}} < 99$, and that at that time every item will be in a height-1 subtree to which it is native, so that the job can be finished with disjoint 4-sorters.

A Suitable Invariant

In this and the next section, the parameters are carefully reintroduced, and constraints on them are accumulated, sufficient for the analysis to work out. For A comfortably larger than 1 (10 in the preview above) and v less than but close to 1 (.65 in the preview above), define the *capacity* (b above) of a depth- d bag after t stages to be $nv^t A^d$. (Again note the dynamic reduction with time, so that this capacity eventually becomes small even compared to the number of items native to the bag.) Let $\lambda < 1$ be chosen as indicated later, and let $\varepsilon > 0$ be small.

Subject to the constraints accumulated in section “Argument that the Invariant is Maintained”, it will be shown there that each successful iteration of the separation–rebagging procedure described in section “The Structure of the Bagging Schedule” (i.e., each stage of the network) preserves the following four-clause invariant.

1. Alternating levels of the tree are entirely empty.
2. On each level, the number of registers currently in each bag (or in the entire subtree below) is the same (and hence at most the size of the corresponding binary interval).
3. The number of registers currently in each bag is bounded by the current capacity of the bag.
4. For each $j \geq 1$, the number of j -strangers currently in the registers of each bag is bounded by $\lambda \varepsilon^{j-1}$ times the bag’s current capacity.

How much successful iteration is enough? Until the leaf capacities b dip below the constant $1/\lambda$. At that point, the subtrees of smallest height k such that $(1/\lambda)(1/A)^{k+1} < 1$ contain all the registers, because higher-level capacities are at most $b/A^{k+1} < (1/\lambda)(1/A)^{k+1} < 1$. And the contents are correctly classified, because the number of $(k-i+1)$ -strangers in each bag at each height $i \leq k$ is bounded by $\lambda \varepsilon^{k-i} b/A^i \leq \lambda b < 1$. So the job can be finished with independent 2^{k+1} -sorters, of depth at most $(k+1)(k+2)/2$ [5]. In fact, for the right choice of parameters (holding $1/\lambda$ to less than A^2), k as small as 1 can suffice, so that the job can be finished with mere 4-sorters, of depth just 3. Therefore, the number t of successful iterations need only exceed $((1+\log_2 A)/\log_2(1/v)) \log_2 n - \log_2(A/\lambda)/\log_2(1/v) = \mathcal{O}(\log n)$, since that is (exactly) enough to get the leaf capacity $nv^t A^{(\log_2 n)-1}$ down to $1/\lambda$.

As noted in the previous section, only one thing can prevent successful iteration of the proposed procedure for each stage: $2\lfloor \lambda b \rfloor + 1$ being less than the number of items to be evacuated from a bag with current capacity b but with no children. Since such a bag is a leaf, it follows from Clause 2 of the invariant that the number of items is at most 2. Thus the condition is $2\lfloor \lambda b \rfloor + 1 < 2$, implying the goal $b < 1/\lambda$ has already been reached.

Therefore, it remains only to choose parameters such that each successful iteration does preserve the entire invariant.

Argument that the Invariant Is Maintained

Only Clauses 3 and 4 are not immediately clear. First consider the former – that capacity will continue after the next iteration to bound the number of registers in each bag. This is nontrivial only for a bag that is currently empty. If the current capacity of such a bag is $b \geq A$, then the next capacity can safely be as low as

$$\begin{aligned} & (\text{number of registers from below}) \\ & + (\text{number of registers from above}) \\ & \leq (4\lambda b A + 2) + b/(2A) \\ & = b(4\lambda A + 1/(2A)) + 2 \\ & \leq b(4\lambda A + 1/(2A) + 2/A), \text{ since } b \geq A. \end{aligned}$$

In the remaining Clause 3 case, the current capacity of the empty bag is $b < A$. Therefore, all higher bags’ capacities are bounded by $b/A < 1$, so that the n registers are equally distributed among the subtrees rooted

on the level below, an even number of registers per subtree. Since each root on that level has passed down an equal net number of registers to each of its children, it currently holds an even number of registers and will not kick back an odd register to the bag of interest. In this case, therefore, the next capacity can safely be as low as just $4\lambda bA$.

In either Clause 3 case, therefore, any $v \geq 4\lambda A + 5/(2A)$ will work.

Finally, turn to restoration of Clause 4. First, consider the relatively easy case of $j > 1$. Again, this is nontrivial only for a bag that is currently empty. Suppose the current capacity of such a bag is b . What is a bound on the number of j -strangers after the next step? It is at most

$$\begin{aligned} & (\text{all } (j+1)\text{-strangers currently in children}) \\ & + ((j-1)\text{-strangers currently in parent, and not} \\ & \quad \text{filtered out by the separation}) \\ & < 2bA\lambda\varepsilon^j + \varepsilon((b/A)\lambda\varepsilon^{j-2}) \\ & \leq \lambda\varepsilon^{j-1}vb, \text{ provided } [2A\varepsilon + 1/A \leq v]. \end{aligned}$$

Note that the bound $\varepsilon((b/A)\lambda\varepsilon^{j-2})$ exploits the “filtering” performance of an approximate separator: At most fraction ε of the “few” smallest (or largest) are permuted “far” out of place.

All that remains is the more involved Clause 4 case of $j = 1$. Consider any currently empty bag B , of current capacity b . At the root there are always *no* strangers; so assume B has a parent, D , and a sibling, C . Let d be the number of registers in D .

There are three sources of 1-strangers in B after the next iteration, two previously strange, essentially as above, and one newly strange:

1. Current 2-strangers at children (at most $2\lambda\varepsilon bA$).
2. Unfiltered current 1-strangers in D (at most $\varepsilon(\lambda(b/A))$, as above).
3. Items in D that are native to C but that now get sent to B instead.

For one of these last items to get sent down to B , it must get permuted by the approximate separator of Lemma 2 into “ B ’s half” of the registers. The number that do is at most the number of C -native items in excess of $d/2$, plus the number of “halving errors” by the approximate separator. By the approximate halving behavior, the latter is at most $\varepsilon b/(2A)$, leaving only the former to estimate.

For this remaining estimate, compare the current “actual” distribution with a more symmetric “benchmark” distribution that has an unchanged number of registers in each bag, but that, for each bag C' on the same level as B , has only C' -native items below C' and has $d/2$ C' -native items in the parent D' of C' . (If d is odd, then the numbers of items in D' native to its two children will be $[d/2]$ and $\lceil d/2 \rceil$, in either order.) That there *is* such a redistribution follows from Clause 2 of the invariant: Start by partitioning the completely sorted list among the bags on B ’s level, and then move items down and up in appropriate numbers to fill the budgeted vacancies.

In the benchmark distribution, the number of C -native items in excess of $d/2$ is 0. If the actual distribution is to have an excess, where can the excess C -native items come from, in terms of the benchmark distribution? They can come only from C -native items on levels above D ’s and from a net reduction in the number of C -native items in C ’s subtree. The latter can only be via the introduction into C ’s subtree of items *not* native to C . By Clause 4 of the invariant, the number of these can be at most

$$\begin{aligned} & 2\lambda\varepsilon bA + 8\lambda\varepsilon^3 bA^3 + 32\lambda\varepsilon^5 bA^5 + 128\lambda\varepsilon^7 bA^7 + \dots \\ & = 2\lambda\varepsilon bA(1 + (2\varepsilon A)^2 + ((2\varepsilon A)^2)^2 \\ & \quad + ((2\varepsilon A)^2)^3 + \dots) < 2\lambda\varepsilon bA/(1 - (2\varepsilon A)^2). \end{aligned}$$

If 2^i is the number of bags C' on the level of C , then the total number of items on levels above D ’s is at most

$$2^{i-3}b/A^3 + 2^{i-5}b/A^5 + 2^{i-7}b/A^7 + \dots$$

Since the number native to each such C' is the same, the number native to C is at most $1/2^i$ times as much:

$$\begin{aligned} & b/(2A)^3 + b/(2A)^5 + b/(2A)^7 + \dots \\ & < b/\left((2A)^3(1 - 1/(2A)^2)\right) \\ & = b/(8A^3 - 2A). \end{aligned}$$

So the total number of 1-strangers from all sources is at most

$$\begin{aligned} & 2\lambda\varepsilon bA + \varepsilon(\lambda(b/A)) + \varepsilon b/(2A) + 2\lambda\varepsilon bA/(1 - (2\varepsilon A)^2) \\ & + b/(8A^3 - 2A). \end{aligned}$$

This total is indeed bounded, as needed, by λvb (λ times the new capacity), provided

$$2\lambda\varepsilon A + \varepsilon\lambda/A + \varepsilon/(2A) + 2\lambda\varepsilon A/(1 - (2\varepsilon A)^2) \\ + 1/(8A^3 - 2A) \leq \lambda v.$$

This completes the argument, subject to the following accumulated constraints:

$$\begin{aligned} A &> 1, \\ v &< 1, \\ \varepsilon &> 0, \\ \lambda &< 1, \\ v &\geq 4\lambda A + 5/(2A), \\ v &\geq 2A\varepsilon + 1/A, \\ \lambda v &\geq 2\lambda\varepsilon A + \varepsilon\lambda/A + \varepsilon/(2A) + 2\lambda\varepsilon A/(1 - (2\varepsilon A)^2) \\ &\quad + 1/(8A^3 - 2A). \end{aligned}$$

Here is one choice order and strategy that works out neatly:

1. Choose A big
2. Choose λ between $1/(8A^3 - 2A)$ and $1/(8A)$
3. Choose ε small
4. Choose v within the resulting allowed range

For example, $A = 10$, $\lambda = 1/100$, $\varepsilon = 1/100$, and $v = 13/20$. In fact, perturbing λ and ε to $1/99$ makes it possible to hold the parameter “ k ” of section “A Suitable Invariant” to 1 and get by with 4-sorters at the very end, as previewed in section “The Structure of the Bagging Schedule”.

Concluding Remarks

Thus the AKS networks are remarkable sorting algorithms, one for each n , that sort their n inputs in just $\mathcal{O}(\log n)$ oblivious compare-exchange steps that involve no concurrent reading or writing.

The amazing kernel of this celebrated result is that the number of such steps for the problem of *approximate halving* (in provable contrast to the problem of *perfect halving*) does not have to grow with n at all. This is yet another (of many) reasons to marvel at the existence of constant-degree expander graphs. And it is another reason to consider approximate solutions to fundamental algorithms. (Could some sort of approximate *merging* algorithm similarly lead to a fast sorting algorithm based on *merge sort*?)

Algorithmically most interesting is the measured use of approximate halvers to clean up their own errors

in a design originally conceived for perfect halvers. The “one-step-backward, two-steps-forward” approach is a valuable one that should and indeed does show up in many other settings. This benefit holds regardless of whether the constant hidden in the big- \mathcal{O} for this particular result can ever be reduced to something practical.

Related Entries

- Bitonic Sort
- Sorting

Bibliographic Notes and Further Reading

Ajtai, Komlós, and Szemerédi submitted the first version of their design and argument directly to a journal [1]. Inspired by Joel Spencer’s ideas for a more accessible exposition, they soon published a quite different conference version [2]. The most influential version, on which most other expositions are based, was the one eventually published by Paterson [12]. The version presented here is based on a much more recent simplification [13], with fewer distinct parameters, but thus with less potential for fine-tuning more quantitative results.

The embedded expander graphs, and thus the constant hidden in the $\mathcal{O}(\log n)$ depth estimate, make these networks impractical to build or use. The competition for the constant is Batcher’s extra $\log_2 n$ factor [5], which is relatively quite small for any remotely practical value of n .

The shallowest approximate halvers can be produced by more direct arguments rather than via expander graphs. By a direct and careful counting argument, Paterson [12] (the theorem in his appendix, specialized to the case $\alpha = 1$) proves that the depth of an ε -halver need be no more than $\lceil(2\log \varepsilon)/\log(1 - \varepsilon) + 2/\varepsilon - 1\rceil$. Others [3, 11] have less precisely sketched or promised asymptotically better results.

Even better, one can focus on the construction of approximate separators. Paterson [12], for example, notes the usefulness in that construction (as presented above, for Lemma 2) of somewhat weaker (and thus shallower) halvers, which perform well only on extreme sets of sizes bounded in terms of an additional parameter $\alpha \leq 1$. Tailoring the parameters to the different levels

of approximate halvers in the construction, he manages to reduce the depth of separators enough to reduce the depth of the resulting sorting network to less than $6100 \log_2 n$.

Ajtai, Komlós, and Szemerédi [3] announce that design in terms of generalized, *multi-way* comparators (i.e., M -sorter units) can lead to drastically shallower approximate halvers and “near-sorters” (their original version [2] of separators). Chvátal [7] pursues this idea carefully and arrives at an improved final bound less than $1830 \log_2 n$, although still somewhat short of the $100 \log_2 n$ privately claimed by Komlós. To date, the tightest analyses have appeared only as sketches in preliminary conference presentations or mere promises of future presentation [3], or as unpublished technical reports [7].

Leighton [10] shows, as a corollary of the AKS result (regardless of the networks), that there is an n -node degree-3 network that can sort n items in $\mathcal{O}(\log n)$ steps.

Considering the VLSI model of parallel computation, Bilardi and Preparata [6] show how to lay out the AKS sorting networks to sort n $\mathcal{O}(\log n)$ -bit numbers in optimal area $\mathcal{O}(n^2)$ and time $\mathcal{O}(\log n)$.

Bibliography

1. Ajtai M, Komlós J, and Szemerédi E (1983) Sorting in $c \log n$ parallel steps. Combinatorica 3(1):1–19
2. Ajtai M, Komlós J, Szemerédi E (1983) An $O(n \log n)$ sorting network, Proceedings of the fifteenth annual ACM symposium on theory of computing, Association for computing machinery, Boston, pp 1–9
3. Ajtai M, Komlós J, Szemerédi E (1992) Halvers and expanders, Proceedings, 33rd annual symposium on foundations of computer science, IEEE computer society press, Los Alamitos, pp 686–692
4. Alekseev VE (1969) Sorting algorithms with minimum memory. Kibernetika 5(5):99–103
5. Batcher KE (1968) Sorting networks and their applications, AFIPS conference proceedings, Spring joint computer conference, vol 32, Thompson, pp 307–314
6. Bilardi G, Preparata FP (1985) The VLSI optimality of the AKS sorting network. Inform Process Lett 20(2):55–59
7. Chvátal V (1992) Lecture notes on the new AKS sorting network, DCS-TR-294, Computer Science Department, Rutgers University
8. Cormen TH, Leiserson CE, Rivest RL, Stein C (2001) Introduction to algorithms, 2nd edn. The MIT Press, Cambridge
9. Knuth DE (1998) The art of computer programming, vol 3, Sorting and searching, 2nd edn. Addison-Wesley, Reading
10. Leighton T (1985) Tight bounds on the complexity of parallel sorting. IEEE Trans Comput C-34(4):344–354

11. Manos H (1999) Construction of halvers. Inform Process Lett 69(6):303–307
12. Paterson MS (1990) Improved sorting networks with $O(\log N)$ depth. Algorithmica 5(1–4):75–92
13. Seiferas J (2009) Sorting networks of logarithmic depth, further simplified. Algorithmica 53(3):374–384
14. Seiferas J (2010) On the counting argument for the existence of expander graphs, manuscript

AKS Sorting Network

►AKS Network

Algebraic Multigrid

MARIAN BREZINA¹, JONATHAN HU², RAY TUMINARO²

¹University of Colorado at Boulder, Boulder, CO, USA

²Sandia National Laboratories, Livermore, CA, USA

Synonyms

AMC

Definition

Multigrid refers to a family of iterative algorithms for solving large sparse linear systems associated with a broad class of integral and partial differential equations [13, 22, 36]. The key to its success lies in the use of efficient coarse scale approximations to dramatically accelerate the convergence so that an accurate approximation is obtained in only a few iterations at a cost that is linearly proportional to the problem size. *Algebraic multigrid* methods construct these coarse scale approximations utilizing only information from the finest resolution matrix. Use of algebraic multigrid solvers has become quite common due to their optimal execution time and relative ease-of-use.

Discussion

Introduction

Multigrid algorithms are used to solve large sparse linear systems

$$Ax = b \quad (1)$$

where A is an $n \times n$ matrix, b is a vector of length n , and one seeks a vector x also of length n . When these

matrices arise from elliptic partial differential equations (PDEs), multigrid algorithms are often provably optimal in that they obtain a solution with $O(n/p + \log p)$ floating point operations where p is the number of processes employed in the calculation. Their rapid convergence rate is a key feature. Unlike most simpler iterative methods, the number of iterations required to reach a given tolerance does not degrade as the system becomes larger (e.g., when A corresponds to a PDE and the mesh used in discretization is refined).

Multigrid development started in the 1970s with the pioneering works of Brandt [6] and Hackbusch [21, 23], though the basic ideas first appeared in the 1960s [4, 18, 19]. The first methods would now be classified as geometric multigrid (GMG). Specifically, applications supply a mesh hierarchy, discrete operators corresponding to the PDE discretization on all meshes, interpolation operators to transfer solutions from a coarse resolution mesh to the next finer one, and restriction operators to transfer solutions from a fine resolution mesh to the next coarsest level. In geometric multigrid, the inter-grid transfers are typically based on a geometrical relationship of a coarse mesh and its refinement, such as using linear interpolation to take a solution on a coarse mesh and define one on the next finer mesh. While early work developed, proved, and demonstrated optimally efficient algorithms, it was the development of algebraic multigrid (AMG) that paved the way for non-multigrid experts to realize this optimal behavior across a broad range of applications including combustion calculations, computational fluid dynamics, electromagnetics, radiation transfer, semiconductor device modeling, structural analysis, and thermal calculations. While better understood for symmetric positive definite (SPD) systems, AMG methods have achieved notable success on other types of linear systems such as those involving strongly convective flows [11, 34] or drift diffusion equations.

The first algebraic multigrid scheme appeared in the mid-1980s and is often referred to as classical algebraic multigrid [7, 33]. This original technique is still one of the most popular, and many improvements have helped extend its applicability on both serial and parallel computers. Although classical algebraic multigrid is frequently referred to as just *algebraic multigrid*, the notation C-AMG is used here to avoid confusion with other algebraic multigrid methods. The list of alternative

AMG methods is quite extensive, but this entry only addresses one alternative referred to as smoothed aggregation [38, 39], denoted here as SA. Both C-AMG and SA account for the primary AMG use by most scientists. Some publicly available AMG codes include [12, 20, 25, 31, 32]. A number of commercial software products also contain algebraic multigrid solvers. Today, research continues on algebraic multigrid to further expand applicability and robustness.

Geometric Multigrid

Although multigrid has been successfully applied to a wide range of problems, its basic principles are easily understood by first studying the two-dimensional Poisson problem

$$-u_{xx} - u_{yy} = f \quad (2)$$

defined over the unit square with homogeneous Dirichlet conditions imposed on the boundary. Discretization leads to an SPD matrix problem

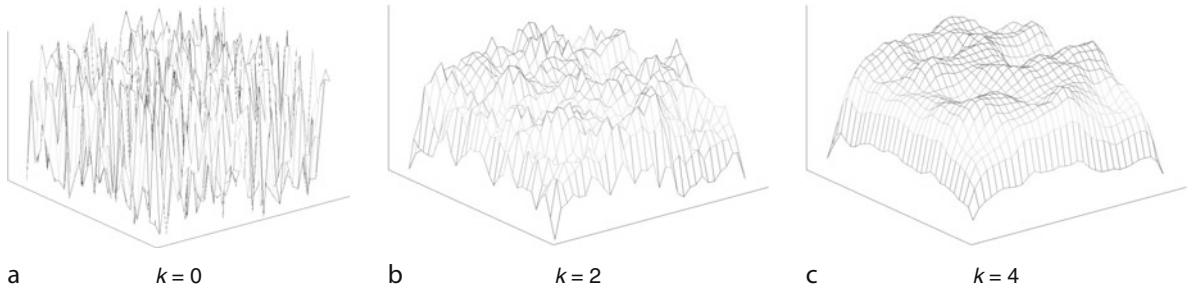
$$Ax = b. \quad (3)$$

The Gauss–Seidel relaxation method is one of the oldest iterative solution techniques for solving (3). It can be written as

$$\mathbf{x}^{(k+1)} \leftarrow (D + L)^{-1}(\mathbf{b} - U\mathbf{x}^{(k)}) \quad (4)$$

where $\mathbf{x}^{(k)}$ denotes the approximate solution at the k^{th} iteration, D is the diagonal of A , L is the strictly lower triangular portion of A , and U is the strictly upper triangular portion of A . While the iteration is easy to implement and economical to apply, a prohibitively large number of iterations (which increases as the underlying mesh is refined) are often required to achieve an accurate solution [40].

Figure 1 depicts errors ($\mathbf{e}^{(k)} = \mathbf{x}^{(*)} - \mathbf{x}^{(k)}$, where $\mathbf{x}^{(*)}$ is the exact solution) over the problem domain as a function of iteration number k for a typical situation on a 32×32 mesh. Notice that while the error remains quite large after four iterations, it has become locally much smoother. Denoting by $\mathbf{r}^{(k)} = \mathbf{b} - Ax^{(k)}$ the residual, the error is found by solving $A\mathbf{e}^{(k)} = \mathbf{r}^{(k)}$. Of course, solving this residual equation appears to be no easier than solving the original system. However, the basic multigrid idea is to recognize that smooth error is well represented

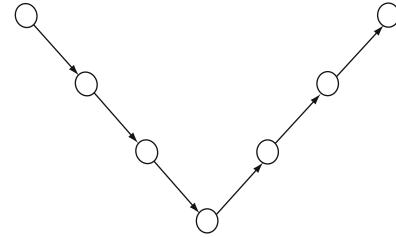


Algebraic Multigrid. Fig. 1 Errors as a function of k , the Gauss–Seidel iteration number

```

MGV( $A_\ell, \mathbf{x}_\ell, \mathbf{b}_\ell, \ell$ ):
  if  $\ell \neq \ell_{max}$ 
     $\mathbf{x}_\ell \leftarrow S_\ell^{pre}(A_\ell, \mathbf{x}_\ell, \mathbf{b}_\ell)$ 
     $\mathbf{r}_\ell \leftarrow \mathbf{b}_\ell - A_\ell \mathbf{x}_\ell$ 
     $\mathbf{x}_{\ell+1} \leftarrow 0$ 
     $\mathbf{x}_{\ell+1} \leftarrow MGV(A_{\ell+1}, \mathbf{x}_{\ell+1}, I_\ell^{\ell+1} \mathbf{r}_\ell, \ell+1)$ 
     $\mathbf{x}_\ell \leftarrow \mathbf{x}_\ell + I_{\ell+1}^\ell \mathbf{x}_{\ell+1}$ 
     $\mathbf{x}_\ell \leftarrow S_\ell^{post}(A_\ell, \mathbf{x}_\ell, \mathbf{b}_\ell)$ 
  else  $\mathbf{x}_\ell \leftarrow A_\ell^{-1} \mathbf{b}_\ell$ 

```



Algebraic Multigrid. Fig. 2 Multigrid V-cycle

on a mesh with coarser resolution. Thus, a coarse version can be formed and used to obtain an approximation with less expense by inverting a smaller problem. This approximation is then used to perform the update $\mathbf{x}^{(*)} \approx \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{e}_c^{(k)}$ where $\mathbf{e}_c^{(k)}$ is defined by interpolating to the fine mesh the solution of the coarse resolution version of $A\mathbf{e}^{(k)} = \mathbf{r}^{(k)}$. This is referred to as *coarse-grid correction*. Since the coarse-level correction may introduce oscillatory components back into the error, it is optionally followed by application of a small number of simple relaxation steps (the post-relaxation). If the size of the coarse discretization matrix is small enough, Gaussian elimination can be applied to directly obtain a solution of the coarse level equations. If the size of the coarse system matrix remains large, the multigrid idea can be applied recursively.

Figure 2 illustrates what is referred to as multigrid V-cycle for solving the linear system $A_\ell \mathbf{x}_\ell = \mathbf{b}_\ell$. Subscripts are introduced to distinguish different resolution approximations. $A_1 = A$ is the operator on the finest level, where one seeks a solution while $A_{\ell_{max}}$ denotes the coarsest level system where Gaussian elimination

can be applied without incurring prohibitive cost. $I_\ell^{\ell+1}$ restricts residuals from level ℓ to level $\ell + 1$, and $I_{\ell+1}^\ell$ prolongates from level $\ell+1$ to level ℓ . $S_\ell^{pre}()$ and $S_\ell^{post}()$ denote a basic iterative scheme (e.g., Gauss–Seidel) that is applied to smooth the error. It will be referred to as *relaxation* in the remainder of the entry, although it is also commonly called *smoothing*. The right side of Fig. 2 depicts the algorithm flow for a four-level method. The lowest circle represents the direct solver. The circles to the left of this represent pre-relaxation while those to the right indicate post-relaxation. Finally, the downward arrows indicate restriction while the upward arrows correspond to interpolation or prolongation, followed by correction of the solution. Each relaxation in the hierarchy effectively damps errors that are oscillatory with respect to the mesh at that level. The net effect is that for a well-designed multigrid, errors at all frequencies are substantially reduced by one sweep of a V-cycle iteration.

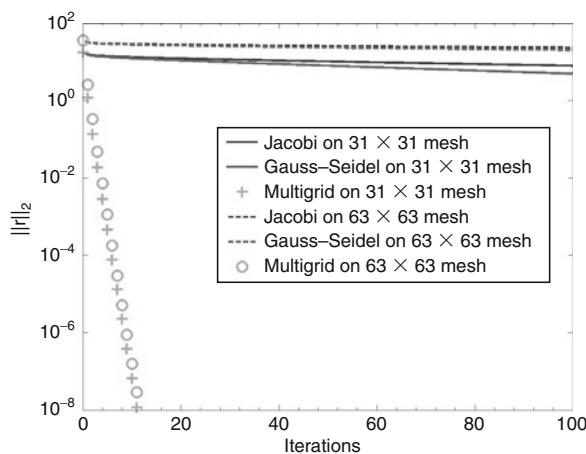
Different multigrid variations visit coarse meshes more frequently. The V-cycle (depicted here) and a W-cycle are the most common. The W-cycle is obtained

by adding a second $\mathbf{x}_{\ell+1} \leftarrow \text{MGV}()$ invocation immediately after the current one in Fig. 2. In terms of cost, grid transfers are relatively inexpensive compared to relaxation. The relaxation cost is typically proportional to the number of unknowns or ck^3 on a $k \times k \times k$ mesh where c is a constant. Ignoring the computational expense of grid transfers and assuming that coarse meshes are defined by halving the resolution in each coordinate direction, the V-cycle cost is

$$\begin{aligned} \text{V-cycle cost} &\approx c \left(k^3 + \frac{k^3}{8} + \frac{k^3}{64} + \frac{k^3}{512} + \dots \right) \\ &\approx \frac{8}{7} \text{ fine-level relaxation cost.} \end{aligned}$$

That is, the extra work associated with the coarse level computations is almost negligible.

In addition to being inexpensive, coarse-level computations dramatically accelerate convergence so that a solution is obtained in a few iterations. Figure 3 illustrates convergence histories for a Jacobi iteration, a Gauss-Seidel iteration, and a multigrid V-cycle iteration where $S_\ell^{pre}()$ and $S_\ell^{post}()$ correspond to one Gauss-Seidel sweep. These experiments use a standard finite difference discretization of (2) on four different meshes with a zero initial guess and a random right hand. Figure 3 clearly depicts the multigrid advantage. *The key to this rapid convergence lies in the complementary nature of relaxation and the coarse level corrections.* Relaxation eliminates high frequency errors while the coarse level correction eliminates low frequency errors



Algebraic Multigrid. Fig. 3 Residuals as a function of k , the Gauss-Seidel iteration number

on each level. Furthermore, errors that are relatively low frequency on a given level appear oscillatory on some coarser level, and will be efficiently eliminated by relaxation on that level.

For complex PDE operators, it can sometimes be challenging to define multigrid components in a way that preserves this complementary balance. Jacobi and Gauss-Seidel relaxations, for example, do not effectively smooth errors in directions of weak coupling when applied to highly anisotropic diffusion PDEs. This is easily seen by examining a damped Jacobi iteration

$$\mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} + \omega D^{-1}(\mathbf{b} - A\mathbf{x}^{(k)}) \quad (5)$$

where D is the diagonal matrix associated with A and ω is a scalar damping parameter typically between zero and one. The error propagation is governed by

$$e^{(k+1)} = (I - \omega D^{-1}A)e^{(k)} = (I - \omega D^{-1}A)^{k+1}e^{(0)} \quad (6)$$

where $e^{(k)} = \mathbf{x}^{(*)} - \mathbf{x}^k$. Clearly, error components in eigenvector directions associated with small eigenvalues of $D^{-1}A$ are almost unaffected by the Jacobi iteration. Gauss-Seidel exhibits similar behavior, though the analysis is more complex. When A corresponds to a standard Poisson operator, these eigenvectors are all low frequency modes, and thus effectively attenuated through the coarse-grid correction process. However, when applied to $u_{xx} + \epsilon u_{yy}$ (with $\epsilon \ll 1$), eigenvectors associated with low eigenvalues are smooth functions in the x direction, but may be oscillatory in the y direction. Within geometric multigrid schemes, where coarsening is dictated by geometrical considerations, these oscillatory error modes are not reduced, and multigrid convergence suffers when based on a simple point-wise relaxation, such as Jacobi. It may thus be necessary to employ more powerful relaxation (e.g., line relaxation) to make up for deficiencies in the choice of coarse grid representations.

Algebraic Multigrid

Algebraic multigrid differs from geometric multigrid in that $I_\ell^{\ell+1}$ and $I_{\ell+1}^\ell$ are not defined from geometric information. They are constructed using only A_ℓ and, optionally, a small amount of additional information (to be discussed). Once grid transfers on a given level are defined, a Galerkin projection is employed to obtain a coarse level discretization:

$$A_{\ell+1} \leftarrow I_\ell^{\ell+1} A_\ell I_{\ell+1}^\ell.$$

Further, $I_\ell^{\ell+1}$ is generally taken as the transpose of the prolongation operator for symmetric problems. In this case, an algebraic V-cycle iteration can be completely specified by a procedure for generating prolongation matrices along with a choice of appropriate relaxation method. In contrast to geometric multigrid, the relaxation process is typically selected upfront, but the prolongation is automatically tailored to the problem at hand. This is referred to as *operator-dependent prolongation*, and can overcome problems such as anisotropies of which geometric multigrid coarsening may not be aware.

Recalling the Jacobi error propagation formula (6), components associated with large eigenvalues of A are easily damped by relaxation while those associated with small eigenvalues remain. (Without loss of generality, one can assume that A has been scaled so that its diagonal is identically one. Thus, A as opposed to $D^{-1}A$ is used to simplify the discussion.) In geometric multigrid, the focus is on finding a relaxation scheme such that all large eigenvalues of its error propagation operator correspond to low frequency eigenvectors. In algebraic multigrid, one assumes that a simple (e.g., Gauss-Seidel) relaxation is employed where the error propagation satisfies

$$\begin{aligned}\|S_\ell e\|_{A_\ell}^2 &\leq \|e\|_{A_\ell}^2 - \alpha \|e\|_{A_\ell}^2 \\ &= \|e\|_{A_\ell}^2 - \alpha \|r\|^2\end{aligned}\quad (7)$$

where α is a positive constant independent of the mesh size. Equation (7) indicates that errors associated with a large A_ℓ -norm are reduced significantly while those associated with small residuals are relatively unchanged by relaxation. The focus is now on constructing $I_\ell^{\ell+1}$ and $I_{\ell+1}^\ell$ such that eigenvectors associated with small eigenvalues of A_ℓ are transferred between grid levels accurately. That is, error modes that are not damped by relaxation on a given level must be transferred to another level so that they can be reduced there. Note that these components need not be smooth in the geometric sense, and are therefore termed *algebraically smooth*.

More formally, the general principle for grid transfers applied to symmetric problems is that they satisfy an approximation property, for example,

$$\|e - I_{\ell+1}^\ell \hat{e}\|^2 \leq \frac{\beta}{\|A_\ell\|} \|e\|_{A_\ell}^2 \quad (8)$$

where \hat{e} is a coarse vector minimizing the left side and β is a constant independent of mesh size and e . Basically, (8) requires that vectors with smaller A_ℓ -norm be more accurately captured on coarse meshes. That is, high and low energy, measured by the A_ℓ -norm, replaces the geometric notions of high and low frequency.

It is possible to show that a two-level method that employs a coarse grid correction satisfying (8) followed by a relaxation method S_ℓ satisfying (7) is itself convergent, *independent of problem size*. The error propagation operator associated with the coarse-grid correction is given by

$$T_\ell = I - I_{\ell+1}^\ell (I_\ell^{\ell+1} A_\ell I_{\ell+1}^\ell)^{-1} I_\ell^{\ell+1} A_\ell. \quad (9)$$

Note that (9) moves fine error to the coarse level, performs an exact solve, interpolates the result back to the fine level, and differences it with the original fine error. Assuming that the grid transfer $I_{\ell+1}^\ell$ is full-rank, the following estimate for a two-level multigrid method can be proved using (8) and (7):

$$\|S_\ell T_\ell\|_{A_\ell} \leq \sqrt{1 - \frac{\alpha}{\beta}}, \quad (10)$$

where post-relaxation only is assumed to simplify the discussion. That is, a coarse-grid correction followed by relaxation converges at a rate independent of the mesh size. Given the general nature of problems treated with algebraic multigrid, however, sharp multilevel convergence results (as opposed to two-level results) are difficult to obtain. The best known multilevel AMG convergence bounds are of the form, $1 - \frac{1}{C(L)}$, where the constant $C(L)$ depends polynomially on the number of multigrid levels [38].

In addition to satisfying some approximation property, grid transfers must be practical. The sparsity patterns of $I_{\ell+1}^\ell$ and $I_\ell^{\ell+1}$ effectively determine the number of nonzeros in $A_{\ell+1}$. If prohibitively large, the method is impractically expensive. Most AMG methods construct grid transfers in several stages. The first stage defines a graph associated with A_ℓ . The second stage coarsens this graph. Coarsening fixes the dimensions of $I_{\ell+1}^\ell$ and $I_\ell^{\ell+1}$. Coarsening may also effectively determine the sparsity pattern of the grid transfer matrices or, within some AMG methods, the sparsity pattern may be constructed separately. Finally, the actual grid transfer

coefficients are determined so that the approximation property is satisfied and that relaxation on the coarse grid is effective.

One measure of expense is the *operator complexity*, $\sum_i(nnz(A_i)/nnz(A_1)$, which compares number of nonzeros on all levels to the number of nonzeros in the finest grid matrix. The operator complexity gives an indication both of the amount of memory required to store the AMG preconditioner and the cost to apply it. Another measure of complexity is the matrix stencil size, which is the average number of coefficients in a row of A_ℓ . Increases in matrix stencil size can lead to increases in communication and matrix evaluation costs. In general, AMG methods must carefully balance accuracy with both types of complexity. Increasing the complexity of an AMG method often leads to better convergence properties, at the cost of each iteration being more expensive. Conversely, decreasing the complexity of a method tends to lead to a method that converges more slowly, but is cheaper to apply.

Within C-AMG, a matrix graph is defined with the help of a strength-of-connection measure. In particular, each vector unknown corresponds to a vertex. A graph edge between vertex i and j is only added if i is strongly influenced by j or if $-a_{ij} \geq \epsilon \max_{i \neq k} (-a_{ik})$, where $\epsilon > 0$ is independent of i and j . The basic idea is to ignore weak connections when defining the matrix graph in order to coarsen only in directions of strong connections (e.g., to avoid difficulties with anisotropic applications). The structure of $I_{\ell+1}^\ell$ is determined by selecting a subset of vertices, called *C-points*, from the fine graph. The *C-points* constitute the coarse graph vertices; the remaining unknowns are called *F-points* and will interpolate from the values at *C-points*. If *C-points* are too close to each other, the resulting complexities are high. If they are too far apart, convergence rates tend to suffer. To avoid these two situations, the selection process attempts to satisfy two conditions. The first is that if point j strongly influences an *F-point* i , then j should either be a *C-point* or j and i should be strongly influenced by a common *C-point*. The second is that the set of *C-points* should form a maximal independent set such that no two *C-points* are strongly influenced by each other. For isotropic problems, the aforementioned C-AMG algorithm tends to select every other unknown as a *C-point*. This yields a *coarsening rate* of roughly 2^d , where d is the spatial problem dimension. With the

C-points identified, the interpolation coefficients are calculated. *C-points* themselves are simply interpolated via injection. The strong influence of an *F-point* j on a different *F-point* i is given by the weighted interpolation from the *C-points* common to i and j . The latter is done in such a manner so as to preserve interpolation of constants. Additionally, it is based on the assumption that residuals are small after relaxation. That is, $A_\ell \mathbf{e} \approx 0$, or, equivalently, modes associated with small eigenvalues should be accurately interpolated. Further details can be found in [36].

SA also coarsens based on the graph associated with matrix A_ℓ . In contrast to C-AMG, however, the coarse grid unknowns are formed from disjoint groups, or *aggregates*, of fine grid unknowns. Aggregates are formed around initially-selected unknowns called root points. An unknown i is included in an aggregate if it is strongly coupled to the root point j . More precisely, unknowns i and j are said to be strongly coupled if $|a_{ij}| > \theta \sqrt{|a_{ii}||a_{jj}|}$, where $\theta \geq 0$ is a tuning parameter independent of i and j . Because the aggregates produced by SA tend to have a graph diameter 3, SA generally coarsens at a rate of 3^d , where d is the geometric problem dimension. After aggregate identification, a tentative prolongator is constructed so that it exactly interpolates a set of user-defined vectors. For scalar PDEs this is often just a single vector corresponding to the constant function. For more complex operators, however, it may be necessary to provide several user-defined vectors. In three-dimensional elasticity, it is customary to provide six vectors that represent the six rigid body modes (three translations and three rotations). In general, these vectors should be near-kernel vectors that are generally problematic for the relaxation. The tentative prolongator is defined by restricting the user-defined vectors to each aggregate. That is, each column of the tentative prolongator is nonzero only for degrees of freedom within a single aggregate, and the values of these nonzeros correspond to one of the user-defined vectors. The columns are locally orthogonalized within an aggregate by a QR algorithm to improve linear independence. The final result is that the user-defined modes are within the range space of the tentative prolongator, the prolongator columns are orthonormal, and the tentative prolongator is quite sparse because each column's nonzeros are associated with a single aggregate. Unfortunately, individual tentative prolongator columns have a high energy

(or large A -norm). This essentially implies that some algebraically smooth modes are not accurately represented throughout the mesh hierarchy even though the user-defined modes are accurately represented. To rectify this, one step of damped Jacobi is applied to all columns of the tentative prolongator:

$$I_{\ell+1}^\ell = (I - \omega D_\ell^{-1} A_\ell) I_\ell^\ell$$

where ω is the Jacobi damping parameter and D_ℓ is the diagonal of A_ℓ . This reduces the energy in each of the columns while preserving the interpolation of user-defined low energy modes. Further details can be found in [38, 39].

Parallel Algebraic Multigrid

Distributed memory parallelization of most simulations based on PDEs is accomplished by dividing the computational domain into subdomains, where one subdomain is assigned to each processor. A processor is then responsible for updating unknowns associated within its subdomain only. To do this, processors occasionally need information from other processors; that information is obtained via communication. Partitioning into boxes or cubes is straightforward for logically rectangular meshes. There are several tools that automate the subdivision of domains for unstructured meshes [5, 26, 27]. A general goal during subdivision is to assign an equal amount of work to each processor and to reduce the amount of communication between processors by minimizing the surface area of the subdomains.

Multigrid parallelization follows in a similar fashion. For example, V-cycle computations within a mesh are performed in parallel, but each mesh in the hierarchy is addressed one at a time as in standard multigrid. A multigrid cycle (see Fig. 2) consists almost entirely of relaxation and sparse matrix–vector products. This means that the parallel performance depends entirely on these two basic kernels. Parallel matrix–vector products are quite straightforward, as are parallel Jacobi and parallel Chebyshev relaxation (which are often preferred in this setting [3]). Chebyshev relaxation requires an estimate of the largest eigenvalue of A_ℓ (which is often available or easily estimated) but not that of the smallest eigenvalue, as relaxation need only damp the high end of the spectrum. Unfortunately, construction of efficient parallel Gauss–Seidel algorithms

is challenging and relies on sophisticated multicoloring schemes on unstructured meshes [2]. As an alternative, most large parallel AMG packages support an option to employ Processor Block (or local) Gauss–Seidel. Here, each processor performs Gauss–Seidel as a subdomain solver for a block Jacobi method. While Processor Block Gauss–Seidel is easy to parallelize, the overall multigrid convergence rate usually suffers and can even lead to divergence if not suitably damped [41].

Given a multigrid hierarchy, the main unique aspect associated with parallelization boils down to partitioning all the operators in the hierarchy. As these operators are associated with matrix graphs, it is actually the graphs that must be partitioned. The partitioning of the finest level graph is typically provided by an outside application and so it ignores the coarse graphs that are created during multigrid construction. While coarse graph partitioning can also be done in this fashion, it is desirable that the coarse and fine graph partitions “match” in some way so that inter-processor communication during grid transfers is minimized. This is usually done by deriving coarse graph partitions from the fine graph partition. For example, when the coarse graph vertices are a subset of fine vertices, it is natural to simply use the fine graph partitioning on the coarse graph. If the coarse graph is derived by agglomerating elements and the fine graph is partitioned by elements, the same idea holds. In cases without a simple correspondence between coarse and fine graphs, it is often natural to enforce a similar condition that coarse vertices reside on the same processors that contain most of the fine vertices that they interpolate [30]. Imbalance can result if the original matrix A_1 itself is poorly load-balanced, or if the coarsening rate differs significantly among processes, leading to imbalance on coarse levels. Synchronization occurs within each level of the multigrid cycle, so the time to process a level is determined by the slowest process. Even when work is well balanced, processors may have only a few points as the total number of graph vertices can diminish rapidly from one level to a coarser level. In fact, it is quite possible that the total number of graph vertices on a given level is actually less than the number of cores/processors on a massively parallel machine. In this case some processors are forced to remain idle while more generally if processes have only a few points, computation time can be dominated by communication. A partial solution

is to redistribute and load-balance points on a subset of processes. While this certainly leaves processes idle during coarse level relaxation, this can speed up run times because communication occurs among fewer processes and expensive global all-to-all communication patterns are avoided. A number of alternative multigrid algorithms that attempt to process coarse level corrections in parallel (as opposed to the standard sequential approach) have been considered [15]. Most of these, however, suffer drawbacks associated with convergence rates or even more complex load balancing issues.

The parallel cost of a simple V-cycle can be estimated to help understand the general behavior. In particular, assume that run time on a single level is modeled by

$$T_{sl} = c_0 \left(\frac{k}{q} \right)^3 + c_1 \left(\alpha + \beta \left(\frac{k}{q} \right)^2 \right)$$

where k^3 is the number of degrees of freedom on the level, q^3 is the number of processors, and $\alpha + \beta w$ measures the cost of sending a message of length w from one processor to another on a distributed memory machine. The constants c_0 and c_1 reflect the ratio of computation to communication inherent in the smoothing and matrix-vector products. Then, if the coarsening rate per dimension in 3D is γ , it is easy to show that the run time of a single AMG V-cycle with several levels, ℓ , is approximately

$$T_{amg} = c_0 \left(\frac{k}{q} \right)^3 \left(\frac{\gamma^3}{\gamma^3 - 1} \right) + c_1 \beta \left(\frac{k}{q} \right)^2 \left(\frac{\gamma^2}{\gamma^2 - 1} \right) + c_1 \alpha \ell$$

where now k^3 is the number of degrees of freedom on the finest mesh. For $\gamma = 2$, a standard multigrid coarsening rate, this becomes

$$T_{amg} = \frac{8}{7} c_0 \left(\frac{k}{q} \right)^3 + \frac{4}{3} c_1 \beta \left(\frac{k}{q} \right)^2 + c_1 \alpha \ell.$$

Comparing this with the single level execution time, we see that the first term is pure computation and it increases by 1.14 to account for the hierarchy of levels. The second term reflects communication bandwidth and it increases by 1.33. The third term is communication latency and it increases by ℓ . Thus, it is to be expected that parallel multigrid spends a higher percentage of time communicating than a single level method. However, if $(k/q)^3$ (the number of degrees of freedom per processor) is relatively large, then the first term dominates and so inefficiencies associated

with the remaining terms are not significant. However, when this first term is not so dominant, then inefficiencies associated with coarse level computations are a concern. Despite a possible loss of efficiency, the convergence benefits of multigrid far outweigh these concerns and so, generally, multilevel methods remain far superior to single level methods, even on massively parallel systems.

The other significant issue associated with parallel algebraic multigrid is parallelizing the construction phase of the algorithm. In principle this construction phase is highly parallel due to the locality associated with most algorithms for generating grid transfers and setting up smoothers. The primary challenge typically centers on the coarsening aspect of algebraic multigrid. For example, the smoothed aggregation algorithm requires construction of aggregates, then a tentative prolongator, followed by a prolongator smoothing step. Prolongator smoothing as well as Galerkin projection (to build coarse level discretizations) require only an efficient parallel matrix–matrix multiplication algorithm. Further, construction of the tentative prolongator only requires that the near-null space be injected in a way consistent with the aggregates followed by a series of independent small QR computations. It is, however, the aggregation phase that can be difficult. While highly parallel in principle, it is difficult to get the same quality of aggregates without having some inefficiencies. Aggregation is somewhat akin to placing floor tiles in a room. If several workers start at different locations and lay tile simultaneously, the end result will likely lead to the trimming of many interior tiles so that things fit. Unfortunately, a large degree of irregularity in aggregates can either degrade the convergence properties of the overall method or significantly increase the cost per iteration. Further, once each process (or worker in our analogy) has only a few rows of the relatively coarse operator A_ℓ , the rate of coarsening slows, thus leading to more multigrid levels. Instead of naively allowing each process to coarsen its portion of the graph independently, several different aggregation-based strategies have been proposed in the context of SA, based on parallel maximal independent sets [1, 37] or using graph-partitioning algorithms [20]. The latter can increase the coarsening rate and decrease the operator complexity. In both cases, an aggregate can span several processors. Parallel variants of C-AMG coarsening

have also been developed to minimize these effects. The ideas are quite similar to the smoothed aggregation context, but the details tend to be distinct due to differences in coarsening rates and the fact that one is based on aggregation while the other centers on identifying vertex subsets. Subdomain blocking [28] coarsens from the process boundary inward, and CLJP coarsening [16, 25] is based on selecting parallel maximal independent sets. A more recently developed aggressive coarsening variant, PMIS [35], addresses the high complexities sometimes seen with CLJP.

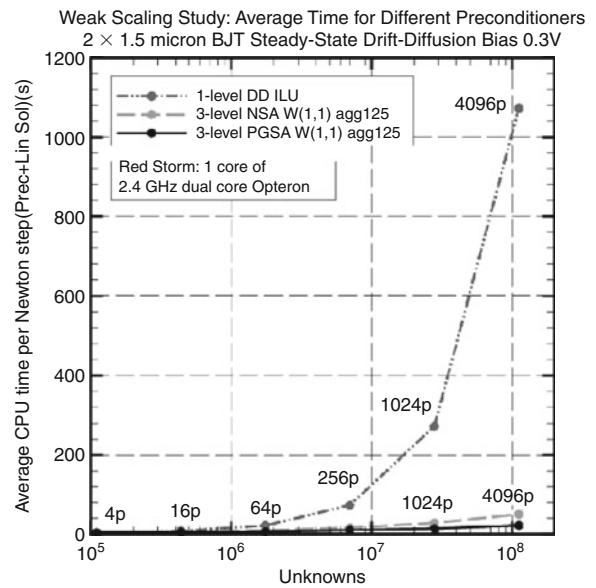
We conclude this section with some performance figures associated with the ML multigrid package that is part of the Trilinos framework. ML implements parallel smoothed aggregation along the lines just discussed. [Figure 4](#) illustrates weak parallel scaling on a semiconductor simulation. Each processor has approximately 27,000 degrees of freedom so that the problem size increases as more processors are used. In an ideal situation, one would like that the total simulation time remains constant as the problem size and processor count increase. Examination of [Fig. 4](#) shows that execution times rise only slightly due to relatively constant run time per iteration.

The emergence of very large scale multi-core architectures presents increased stresses on the underlying multigrid parallelization algorithms and so research must continue to properly take advantage of these new architectures.

Related Multigrid Approaches

When solving difficult problems, it is often advantageous to “wrap” the multigrid solver with a Krylov method. This amounts to using a small number of AMG iterations as a preconditioner in a Krylov method. The additional cost per iteration amounts to an additional residual computation and a small number of inner products, depending on Krylov method used. Multigrid methods exhibiting a low-dimensional deficiency can have good convergence rates restored this way.

Standard implementations of C-AMG and SA base their coarsening on assumptions of smoothness. For C-AMG, this is that an algebraically smooth error behaves locally as a constant. When this is not the case, convergence will suffer. One advantage of SA is that it is possible to construct methods capable of approximating *any* prescribed user-defined error component accurately on coarse meshes. Thus, in principle, if one knows



Algebraic Multigrid. Fig. 4 Weak scaling timings for a semiconductor parallel simulation. The *left image* is the steady-state electric potential; *red* represents high potential, *blue* indicates low potential. The scaling results compare GMRES preconditioned by domain-decomposition, a nonideal AMG method, and an AMG method intended for nonsymmetric problems, respectively. Results courtesy of [29]

difficult components for relaxation, suitable grid transfers can be defined that accurately transfer these modes to coarse levels. However, there are applications where such knowledge is lacking. For instance, QCD problems may have multiple difficult components that are not only oscillatory, but are not *a priori* known. *Adaptive multigrid* methods have been designed to address such problems [9, 10]. The key feature is that they determine the critical components and modify coarsening to ensure their attenuation. These methods can be intricate, but are based on the simple observation that an iteration process applied to the homogeneous problem, $A_\ell \mathbf{x}_\ell = \mathbf{0}$, will either converge with satisfactory rate, or reveal, as its solution, the error that the current method does not attenuate. As the method improves, any further components are revealed more rapidly. The adaptive methods are currently more extensively developed within the SA framework, but progress has also been made within the context of C-AMG, [10].

Another class of methods that attempt to take advantage of additional available information are known as AMGe [8, 14]. These utilize local finite element information, such as local stiffness matrices, to construct inter-grid transfers. Although this departs from the AMG framework, variants related to the methodology have been designed that do not explicitly require local element matrices [17, 24].

AMG methods have been successfully applied directly to high-order finite element discretizations. However, given the much denser nature of a matrix, A^H , obtained from high-order discretization, it is usually advantageous to precondition the high-order system by an inverse of the low-order discretization, A^L , corresponding to the problem over the same nodes that are used to define the high-order Lagrange interpolation. One iteration of AMG can then be used to efficiently approximate the action of $(A^L)^{-1}$, and the use of A^H is limited to computing a fine-level residual, which may be computed in a matrix-free fashion.

Related Entries

- Metrics
- Preconditioners for Sparse Iterative Methods
- Rapid Elliptic Solvers

Acknowledgment

Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin company, for the US Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Bibliography

1. Adams MF (1998) A parallel maximal independent set algorithm. In: Proceedings 5th Copper Mountain conference on iterative methods, Copper Mountain
2. Adams MF (2001) A distributed memory unstructured Gauss-Seidel algorithm for multigrid smoothers. In: ACM/IEEE Proceedings of SC01: High Performance Networking and Computing, Denver
3. Adams M, Brezina M, Hu J, Tuminaro R (July 2003) Parallel multigrid smoothing: polynomial versus Gauss-Seidel. *J Comp Phys* 188(2):593–610
4. Bakhalov NS (1966) On the convergence of a relaxation method under natural constraints on an elliptic operator. *Z Vycisl Mat Mat Fiz* 6:861–883
5. Boman E, Devine K, Heaphy R, Hendrickson B, Leung V, Riesen LA, Vaughan C, Catalyurek U, Bozdag D, Mitchell W, Teresco J, Zoltan (2007) 3.0: Parallel partitioning, load balancing, and data-management services; user's guide. Sandia National Laboratories, Albuquerque, NM, 2007. Tech. Report SAND2007-4748W. <http://www.cs.sandia.gov/Zoltan/ughtml/ug.html>
6. Brandt A (1977) Multi-level adaptive solutions to boundary value problems. *Math Comp* 31:333–390
7. Brandt A, McCormick S, Ruge J (1984) Algebraic multigrid (AMG) for sparse matrix equations. In: Evans DJ (ed) Sparsity and its applications. Cambridge University Press, Cambridge
8. Brezina M, Cleary AJ, Falgout RD, Henson VE, Jones JE, Manteuffel TA, McCormick SF, Ruge JW (2000) Algebraic multigrid based on element interpolation (AMGe). *SIAM J Sci Comp* 22:1570–1592
9. Brezina M, Falgout R, MacLachlan S, Manteuffel T, McCormick S, Ruge J (2005) Adaptive smoothed aggregation (α SA) multigrid. *SIAM Rev* 47(2):317–346
10. Brezina M, Falgout R, MacLachlan S, Manteuffel T, McCormick S, Ruge J (2006) Adaptive algebraic multigrid. *SIAM J Sci Comp* 27:1261–1286
11. Brezina M, Manteuffel T, McCormick S, Ruge J, Sanders G (2010) Towards adaptive smoothed aggregation (α SA) for nonsymmetric problems. *SIAM J Sci Comput* 32:14
12. Brezina M (2002) SAMISdat (AMG) version 0.98 - user's guide
13. Briggs WL, Henson VE, McCormick S (2000) A multigrid tutorial, 2nd ed. SIAM, Philadelphia
14. Chartier T, Falgout RD, Henson VE, Jones J, Manteuffel T, McCormick S, Ruge J, Vassilevski PS (2003) Spectral AMGe (ρ AMGe). *SIAM J Sci Comp* 25:1–26
15. Chow E, Falgout R, Hu J, Tuminaro R, Meier-Yang U (2006) A survey of parallelization techniques for multigrid solvers.

- In: Parallel processing for scientific computing, SIAM book series on software, environments, and tools. SIAM, Philadelphia, pp 179–201
16. Cleary AJ, Falgout RD, Henson VE, Jones JE (1998) Coarse-grid selection for parallel algebraic multigrid. In: Proceedings of the fifth international symposium on solving irregularly structured problems in parallel. Lecture Notes in Computer Science, vol 1457. Springer New York, pp 104–115
 17. Dohrmann CR (2007) Interpolation operators for algebraic multigrid by local optimization. *SIAM J Sci Comp* 29:2045–2058
 18. Fedorenko RP (1961/1962) A relaxation method for solving elliptic difference equations. *Z Vycisl Mat Mat Fiz* 1:922–927 (1961). Also in U.S.S.R. Comput Math Math Phys 1:1092–1096 (1962)
 19. Fedorenko RP (1964) The speed of convergence of one iterative process. *Z. Vycisl. Mat Mat Fiz* 4:559–563. Also in U.S.S.R. Comput Math Math Phys 4:227–235
 20. Gee M, Siefert C, Hu J, Tuminaro R, Sala M (2006) ML 5.0 smoothed aggregation user's guide. Technical Report SAND2006-2649, Sandia National Laboratories, Albuquerque, NM, 87185
 21. Hackbusch W (1977) On the convergence of a multi-grid iteration applied to finite element equations. Technical Report Report 77-8, Institute for Applied Mathematics, University of Cologne, West Germany
 22. Hackbusch W (1985) Multigrid methods and applications, vol 4. Computational mathematics. Springer, Berlin
 23. Hackbusch W (1980) Convergence of multi-grid iterations applied to difference equations. *Math Comput* 34(150):425–440
 24. Henson VE, Vassilevski PS (2001) Element-free AMGe: general algorithms for computing interpolation weights in AMG. *SIAM J Sci Comp* 23:629–650
 25. Henson VE, Yang UM (2002) BoomerAMG: a parallel algebraic multigrid solver and preconditioner. *Appl Numer Math* 41(1): 155–177
 26. Karypis G, Kumar V (1995) Multilevel k-way partitioning scheme for irregular graphs. Technical Report 95-064, Department of Computer Science, University of Minnesota
 27. Karypis G, Kumar V (1995) ParMETIS: Parallel graph partitioning and sparse matrix ordering library. Technical Report 97-060, Department of Computer Science, University of Minnesota
 28. Krechel A, Stüben K (2001) Parallel algebraic multigrid based on subdomain blocking. *Parallel Comput* 27(8):1009–1031
 29. Lin PT, Shadid JN, Tuminaro RS, Sala M (2010) Performance of a Petrov-Galerkin algebraic multilevel preconditioner for finite element modeling of the semiconductor device drift-diffusion equations. *Int J Num Meth Eng.* Early online publication, doi:10.1002/nme.2902
 30. Mavriplis DJ (2002) Parallel performance investigations of an unstructured mesh Navier-Stokes solver. *Intl J High Perf Comput Appl* 16:395–407
 31. Prometheus. http://www.columbia.edu/_m2325/promintro.html.
 32. Stefan Reitzinger. <http://www.numa.unilinz.ac.at/research/projects/pebbles.html>
 33. Ruge J, Stüben K (1987) Algebraic multigrid (AMG). In: McCormick SF (ed) Multigrid methods, vol 3 of Frontiers in applied mathematics. SIAM, Philadelphia, pp 73–130
 34. Sala M, Tuminaro R (2008) A new Petrov-Galerkin smoothed aggregation preconditioner for nonsymmetric linear systems. *SIAM J Sci Comput* 31(1):143–166
 35. De Sterck H, Yang UM, Heys JJ (2006) Reducing complexity in parallel algebraic multigrid preconditioners. *SIAM J Matrix Anal Appl* 27(4):1019–1039
 36. Trottenberg U, Oosterlee C, Schüller A (2001) Multigrid. Academic, London
 37. Tuminaro R, Tong C (2000) Parallel smoothed aggregation multigrid: aggregation strategies on massively parallel machines. In: Donnelley J (ed) Supercomputing 2000 proceedings, 2000
 38. Vaněk P, Brezina M, Mandel J (2001) Convergence of algebraic multigrid based on smoothed aggregation. *Numerische Mathematik* 88:559–579
 39. Vaněk P, Mandel J, Brezina M (1996) Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. *Computing* 56:179–196
 40. Varga RS (1962) Matrix iterative analysis. Prentice-Hall, Englewood Cliffs
 41. Yang UM (2004) On the use of relaxation parameters in hybrid smoothers. *Numer Linear Algebra Appl* 11:155–172

Algorithm Engineering

PETER SANDERS

Universitaet Karlsruhe, Karlsruhe, Germany

Synonyms

[Experimental parallel algorithmics](#)

Definition

Algorithmics is the subdiscipline of computer science that studies the systematic development of efficient algorithms. Algorithm Engineering (AE) is a methodology for algorithmic research that views design, analysis, implementation, and experimental evaluation of algorithms as a cycle driving algorithmic research. Further components are realistic models, algorithm libraries, and a multitude of interrelations to applications. Fig. 1 gives an overview. A more detailed definition can be found in [6]. This article is concerned with particular issues that arise in engineering parallel algorithms.

Discussion

Introduction

The development of algorithms is one of the core areas of computer science. After the early days of the 1940s–1960s, in the 1970s and 1980s, algorithmics was largely viewed as a subdiscipline of computer science that is concerned with “paper-and-pencil” theoretical work – design of algorithms with the goal to prove worst-case performance guarantees. However, in the 1990s it became more and more apparent that this purely theoretical approach delays the transfer of algorithmic results into applications. Therefore, in *algorithm engineering* implementation and experimentation are viewed as equally important as design and analysis of algorithms. Together these four components form a feedback cycle: Algorithms are designed, then analyzed and implemented. Together with experiments using *realistic inputs*, this process induces new insights that lead to modified and new algorithms. The methodology of algorithm engineering is augmented by using *realistic models* that form the basis of algorithm descriptions, analysis, and implementation and by *algorithm libraries* that give high quality reusable implementations.

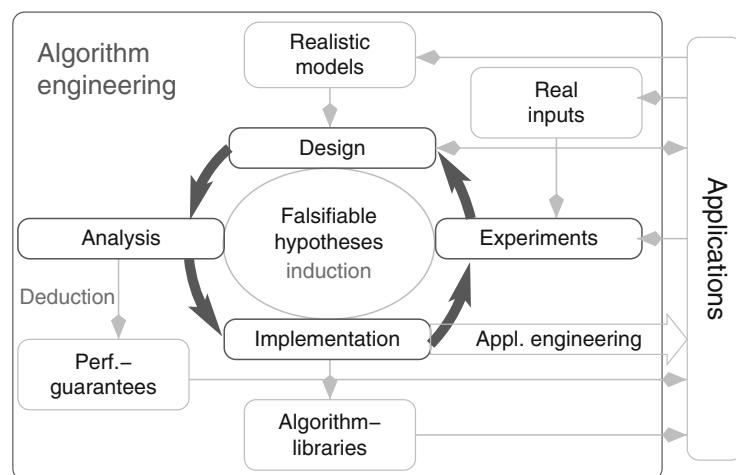
The history of parallel algorithms is exemplary for the above development, where many clever algorithms were developed in the 1980s that were based on the Parallel Random Access Machine (PRAM) model of computation. While this yielded interesting insights into the

basic aspects of parallelizable problems, it has proved quite difficult to implement PRAM algorithms on mainstream parallel computers.

The remainder of this article closely follows Fig. 1, giving one section for each of the main areas’ models, design, analysis, implementation, experiments, instances/benchmarks, and algorithm libraries.

Models

Parallel computers are complex systems containing processors, memory modules, and networks connecting them. It would be very complicated to take all these aspects into account at all times when designing, analyzing, and implementing parallel algorithms. Therefore we need simplified models. Two families of such models have proved very useful: In a *shared memory* machine, all processors access the same global memory. In a *distributed memory* machine, several sequential computers communicate via an interconnection network. While these are useful abstractions, the difficulty is to make these models more concrete by specifying what operations are supported and how long they take. For example, shared memory models have to specify how long it takes to access a memory location. From sequential models we are accustomed to constant access time and this also reflects the best case behavior of many parallel machines. However, in the worst case, most real world machines will exhibit severe contention



Algorithm Engineering. Fig. 1 Algorithm engineering as a cycle of design, analysis, implementation, and experimental evaluation driven by falsifiable hypotheses

when many processors access the same memory module. Hence, despite many useful models (e.g., QRQW PRAM – Queue Read Queue Write Parallel Random Access Machine [3]), there remains a considerable gap to reality when it comes to large-scale shared memory systems.

The situation is better for distributed memory machines, in particular, when the sequential machines are connected by a powerful switch. We can then assume that all processors can communicate with an arbitrary partner with predictable performance. The LogP model [2] and the Bulk Synchronous Parallel model [10] put this into a simple mathematical form. Equally useful are folklore models where one simply defines the time needed for exchanging a message as the sum of a startup overhead and a term proportional to the message length. Then the assumption is usually that every processor can only communicate with a single other processor at a time, or perhaps it can receive from one processor and send to another one. Also note that algorithms *designed* for a distributed memory model often yield efficient *implementations* on shared memory machines.

Design

As in algorithm theory, we are interested in efficient algorithms. However, in algorithm engineering, it is equally important to look for simplicity, implementability, and possibilities for code reuse. In particular, since it can be very difficult to debug parallel code, the algorithms should be designed for simplicity and testability. Furthermore, efficiency means not just asymptotic worst-case efficiency, but we also have to look at the constant factors involved. For example, we have to be aware that operations where several processors interact can be a large constant factor more expensive than local computations. Furthermore, we are not only interested in worst-case performance but also in the performance for real-world inputs. In particular, some theoretically efficient algorithms have similar best-case and worst-case behavior whereas the algorithms used in practice perform much better on all but contrived examples.

Analysis

Even simple and proven practical algorithms are often difficult to analyze and this is one of the main reasons

for gaps between theory and practice. Thus, the analysis of such algorithms is an important aspect of AE.

For example, a central problem in parallel processing is partitioning of large graphs into approximately equal-sized blocks such that few edges are cut. This problem has many applications, for example, in scientific computing. Currently available algorithms with performance guarantees are too slow for practical use. Practical methods iteratively contract the graph while preserving its basic structure until only few nodes are left, compute an initial solution on this coarse representation, and then improve by local search on each level. These algorithms are very fast and yield good solutions in many situations, yet no performance guarantees are known (see [11] for a recent overview).

Implementation

Despite huge efforts in parallel programming languages and in parallelizing compilers, implementing parallel algorithms is still one of the main challenges in the algorithm engineering cycle. There are several reasons for this. First, there are huge semantic gaps between the abstract algorithm description, the programming tools used, and the actual hardware. In particular, really efficient codes often use fairly low-level programming interfaces such as MPI or atomic memory operations in order to keep the overheads for processor interaction manageable. Perhaps more importantly, *debugging* parallel programs is notoriously difficult.

Since performance is the main reason for using parallel computers in the first place, and because of the complexity of parallel hardware, performance tuning is an important part of the implementation phase. Although the line between implementation and experimentation is blurred here, there are differences. In particular, performance tuning is less systematic. For example, there is no need for reproducibility, detailed studies of variances, etc., when one finds out that sequential file I/O is a prohibitive bottleneck or when it turns out that a collective communication routine of a particular MPI implementation has a performance bug.

Experiments

Meaningful experiments are the key to closing the cycle of the AE process. Compared to the natural sciences, AE is in the privileged situation where it can perform many experiments with relatively little effort.

However, the other side of the coin is highly nontrivial planning, evaluation, archiving, postprocessing, and interpretation of results. The starting point should always be falsifiable hypotheses on the behavior of the investigated algorithms, which stem from the design, analysis, implementation, or from previous experiments. The result is a confirmation, falsification, or refinement of the hypothesis. The results complement the analytic performance guarantees, lead to a better understanding of the algorithms, and provide ideas for improved algorithms, more accurate analysis, or more efficient implementation.

Experiments with parallel algorithms are challenging because the number of processors (let alone other architectural parameters) provide another degree of freedom for the measurements, because even parallel programs without randomization may show nondeterministic behavior on real machines, and because large parallel machines are an expensive resource.

Experiments on comparing the quality of the computed results are not so much different from the sequential case. In the following, we therefore concentrate on performance measurements.

Measuring Running Time

The *CPU time* is a good way to characterize the time used by a sequential process (without I/Os), even in the presence of some operating system interferences. In contrast, in parallel programs we have to measure the actual elapsed time (*wall-clock time*) in order to capture all aspects of the parallel program, in particular, communication and synchronization overheads. Of course, the experiments must be performed on an otherwise unloaded machine, by using dedicated job scheduling and by turning off unnecessary components of the operating system on the processing nodes. Usually, further aspects of the program, like startup, initialization, and shutdown, are not interesting for the measurement. Thus timing is usually done as follows: All processors perform a barrier synchronization immediately before the piece of program to be timed; one processor x notes down its local time and all processors execute the program to be measured. After another barrier synchronization, processor x measures the elapsed time. As long as the running time is large compared to the time for a barrier synchronization, this is an easy way to measure wall-clock time. To get reliable results,

averaging over many repeated runs is advisable. Nevertheless, the measurements may remain unreliable since rare delays, for example, due to work done by the operating system, can become quite frequent when they can independently happen on any processor.

Speedup and Efficiency

In parallel computing, running time depends on the number of processors used and it is sometimes difficult to see whether a particular execution time is good or bad considering the amount of resources used. Therefore, derived measures are often used that express this more directly. The *speedup* is the ratio of the running time of the fastest known sequential implementation to that of the parallel running time. The speedup directly expresses the impact of parallelization. The *relative speedup* is easier to measure because it compares with the parallel algorithm running on a single processor. However, note that the relative speedup can significantly overstate the actual usefulness of the parallel algorithm, since the sequential algorithm may be much faster than the parallel algorithm run on a single processor.

For a fixed input and a good parallel algorithm, the speedup will usually start slightly below one for a single processor, and initially goes up linearly with the number of processors. Eventually, the speedup curve gets more and more flat until parallelization overheads become so large that the speedup goes down again. Clearly, it makes no sense to add processors beyond the maximum of the speedup curve. Usually it is better to stop much earlier in order to keep the parallelization cost-effective. *Efficiency*, the ratio of the speedup to the number of processors, more directly expresses this. Efficiency usually starts somewhere below one and then slowly decreases with the number of processors. One can use a threshold for the minimum required efficiency to decide on the maximum number of efficiently usable processors.

Often, parallel algorithms are not really used to decrease execution time but to increase the size of the instances that can be handled in reasonable time. From the point of view of speedup and efficiency, this is good news because for a scalable parallel algorithm, by sufficiently increasing the input size, one can efficiently use any number of processors. One can check this experimentally by scaling the input size together with the number of processors. An interesting property of an

algorithm is *how much* one has to increase the input size with the number of processors. The *isoefficiency function* [4] expresses this relation analytically, giving the input size needed to achieve some given, constant efficiency. As usual in algorithmics, one uses asymptotic notation to get rid of constant factors and lower order terms.

Speedup Anomalies

Occasionally, efficiency exceeding one (also called *superlinear speedup*) causes confusion. By Brent's principle (a single processor can simulate a p -processor algorithm with a uniform slowdown factor of p) this should be impossible. However, genuine superlinear absolute speedup can be observed if the program relies on resources of the parallel machine not available to a simulating sequential machine, for example, main memory or cache.

A second reason is that the computations done by an algorithm can be done in many different ways, some leading to a solution fast, some more slowly. Hence, the parallel program can be “lucky” to find a solution more than p times earlier than the sequential program. Interestingly, such effects do *not* always disappear when averaging over all inputs. For example, Schöning [7] gives a randomized algorithm for finding satisfying assignments to formulas in propositional calculus that are in conjunctive normal form. This algorithm becomes exponentially faster when run in parallel on many (possibly simulated) processors. Moreover, its worst-case performance is better than any sequential algorithm. Brent's principle is not violated since the best sequential algorithm turns out to be the emulation of the parallel algorithm.

Finally, there are many cases where superlinear speedup is not genuine, mostly because the sequential algorithm used for comparison is not really the best one for the inputs considered.

Instances and Benchmarks

Benchmarks have a long tradition in parallel computing. Although their most visible use is for comparing different machines, they are also helpful within the AE cycle. During implementation, benchmarks of basic operations help to select the right approach. For example, SKaMPI [5] measures the performance of most MPI

calls and thus helps to decide which of several possible calls to use, or whether a manual implementation could help.

Benchmark suites of input instances for an important computational problem can be key to consistent progress on this problem. Compared to the alternative that each working group uses its own inputs, this has obvious advantages: there can be wider range of inputs, results are easier to compare, and bias in instance selection is less likely. For example, Chris Walshaw's graph partitioning archive [9] has become an important reference point for graph partitioning.

Synthetic instances, though less realistic than real-world inputs can also be useful since they can be generated in any size and sometimes are good as stress tests for the algorithms (though it is often the other way round – naively constructed random instances are likely to be unrealistically easy to handle). For example, for the graph partitioning problem, one can generate graphs that almost look like random graphs but have a pre-designed structure that can be more or less easy to detect according to tunable parameters.

Algorithm Libraries

Algorithm libraries are made by assembling implementations of a number of algorithms using the methods of software engineering. The result should be efficient, easy to use, well documented, and portable. Algorithm libraries accelerate the transfer of know-how into applications. Within algorithmics, libraries simplify comparisons of algorithms and the construction of software that builds on them. The software engineering involved is particularly challenging, since the applications to be supported are *unknown* at library implementation time and because the separation of interface and (often highly complicated) implementation is very important. Compared to an application-specific reimplementation, using a library should save development time without leading to inferior performance. Compared to simple, easy to implement algorithms, libraries should improve performance. To summarize, the triangle between generality, efficiency, and ease of use leads to challenging trade-offs because often optimizing one of these aspects will deteriorate the others. It is also worth mentioning that *correctness* of algorithm libraries is even more important than for other softwares because it is extremely difficult for a user to debug a library code

that has not been written by his team. All these difficulties imply that implementing algorithms for use in a library is several times more difficult than implementations for experimental evaluation. On the other hand, a good library implementation might be *used* orders of magnitude more frequently. Thus, in AE there is a natural mechanism leading to many exploratory implementations and a few selected library codes that build on previous experimental experience.

In parallel computing, there is a fuzzy boundary between software libraries whose main purpose is to shield the programmer from details of the hardware and genuine algorithm libraries. For example, the basic functionality of MPI (message passing) is of the first kind, whereas its collective communication routines have a distinctively algorithmic flavor. The Intel Thread Building Blocks (<http://www.threadingbuildingblocks.org>) offers several algorithmic tools including a load balancer hidden behind a task concept and distributed data structures such as hash tables. The standard libraries of programming languages can also be parallelized. For example, there is a parallel version of the C++ STL in the GNU distribution [8].

The Computational Geometry Algorithms Library (CGAL, <http://www.cgal.org>) is a very sophisticated example of an algorithms library that is now also getting partially parallelized [1].

Conclusion

This article explains how algorithm engineering (AE) provides a methodology for research in parallel algorithmics that allows to bridge gaps between theory and practice. AE does not abolish theoretical analysis but contains it as an important component that, when applicable, provides particularly strong performance and robustness guarantees. However, adding careful implementation, well-designed experiments, realistic inputs, algorithm libraries, and a process coupling all of this together provides a better way to arrive at algorithms useful for real-world applications.

Bibliography

1. Batista VHF, Millman DL, Pion S, Singler J (2009) Parallel geometric algorithms for multi-core computers. In: 25th ACM symposium on computational geometry, pp 217–226
2. Culler D, Karp R, Patterson D, Sahay A, Schauske KE, Santos E, Subramonian R, Eicken T (1993) LogP: towards a realistic model

of parallel computation. In: 4th ACM SIGPLAN symposium on principles and practice of parallel programming, pp 1–12, San Diego, 19–22 May, 1993. ACM, New York

3. Gibbons PB, Matias Y, Ramachandran V (1998) The queue-read queue-write pram model: accounting for contention in parallel algorithms. SIAM J Comput 28(2):733–769
4. Grama AY, Gupta A, Kumar V (1993) Isoefficiency: measuring the scalability of parallel algorithms and architectures. IEEE Concurr 1(3):12–21
5. Reussner R, Sanders P, Prechelt L, Müller M (1998) SKaMPI: a detailed, accurate MPI benchmark. In: EuroPVM/MPI, number 1497 in LNCS, pp 52–59
6. Sanders P (2009) Algorithm engineering – an attempt at a definition. In: Efficient Algorithms. Lecture Notes in Computer Science, vol 5760. Springer, pp 321–340
7. Schöning U (1999) A probabilistic algorithm for k-sat and constraint satisfaction problems. In: 40th IEEE symposium on foundations of computer science, pp 410–414
8. Singler J, Sanders P, Putze F (2007) MCSTL: the multi-core standard template library. In: 13th international Euro-Par conference. LNCS, vol 4641. Springer, pp 682–694
9. Sopern AJ, Walshaw C, Cross M (2004) A combined evolutionary search and multilevel optimisation approach to graph partitioning. J Global Optim 29(2):225–241
10. Valiant L (1994) A bridging model for parallel computation. Commun ACM 33(8)
11. Walshaw C, Cross M (2007) JOSTLE: parallelmultilevel graph-partitioning software – an overview. In: Magoules F (ed) Mesh partitioning techniques and domain decomposition techniques, pp 27–58. Civil-Comp Ltd. (Invited chapter)

Algorithmic Skeletons

- ▶ Parallel Skeletons

All Prefix Sums

- ▶ Reduce and Scan
- ▶ Scan for Distributed Memory, Message-Passing Systems

Allen and Kennedy Algorithm

- ▶ Parallelism Detection in Nested Loops, Optimal

Allgather

JESPER LARSSON TRÄFF¹, ROBERT A. VAN DE GEIJN²

¹University of Vienna, Vienna, Austria

²The University of Texas at Austin, Austin, TX, USA

Synonyms

All-to-all broadcast; Collect; Concatenation; Gather-to-all; Gossiping; Total exchange

Definition

Among a group of processing elements (nodes) each node has a data item that is to be transferred to *all* other nodes, such that all nodes in the group end up having all of the data items. The allgather operation accomplishes this total data exchange.

Discussion

The reader may consider first visiting the entry on *collective communication*.

It is assumed that the p nodes in the group of nodes participating in the allgather operation are indexed consecutively, each having an index i with $0 \leq i < p$. It is furthermore assumed that the data items are to be collected in some fixed order determined by the node indices; each node may apply a different order. Assume that each node i initially has a vector of data x_i of some number of elements n_i with $n = \sum_{i=0}^{p-1} n_i$ being the total number of subvector elements. Upon completion of the allgather operation each node i will have the full vector x consisting of the subvectors x_i , $i = 0, \dots, p - 1$. This is shown in Fig. 1.

All nodes in the group are assumed to explicitly take part in the allgather communication operation. If all subvectors x_i have the same number of elements $n_i = n/p$ the allgather operation is said to be *regular*, otherwise *irregular*. It is

Before			After		
Node 0	Node 1	Node 2	Node 0	Node 1	Node 2
x_0			x_0	x_0	x_0
	x_1		x_1	x_1	x_1
		x_2	x_2	x_2	x_2

Allgather. Fig. 1 Allgather on three nodes

commonly assumed that all nodes know the size of all subvectors in advance. The Message Passing Interface (MPI), for instance, makes this assumption for both its regular MPI_Allgather and its irregular MPI_Allgatherv collective operations. Other collective interfaces that support the operation make similar assumptions. This can be assumed without loss of generality. If it is not the case, a special, one item per node allgather operation can be used to collect the subvector sizes at all nodes.

The allgather operation is a symmetric variant of the broadcast operation in which all nodes concurrently broadcast their data item, and is therefore often referred to as *all-to-all broadcast*. It is semantically equivalent to a gather operation that collects data items from all nodes at a specific *root* node, followed by a broadcast from that root node, or to p concurrent gather operations with each node serving as root in one such operation. This explains the term *allgather*. The term *concatenation* can be used to emphasize that the data items are gathered in increasing index order at all nodes if such is the case. When concerned with the operation for specific communication networks (graphs) the term *gossiping* has been used. The problem has, like broadcast, been extensively studied in the literature, and other terminology is occasionally found.

Lower Bounds

To obtain lower bounds for the allgather operation a *fully connected, k -ported, homogeneous* communication system with *linear communication cost* is assumed. This means that

- All nodes can communicate directly with all other nodes, at the same communication cost,
- In each communication operation, a node can send up to k distinct messages to k other nodes, and simultaneously receive k messages from k possibly different nodes, and
- The cost of transmitting a message of size n (in some unit) between any two nodes is modeled by a simple, linear function $\alpha + n\beta$. Here α is a start-up latency and β the transfer cost per unit.

With these approximations, lower bounds on the number of communication rounds during which nodes are involved in k -ported communication, and the total

amount of data that have to be transferred along a critical path of any algorithm, can be easily established:

- Since the allgather operation implies a broadcast of a data item from each node, the number of communication rounds is at least $\lceil \log_{k+1} p \rceil$. This follows because the number of nodes to which this particular item has been broadcast can at most increase by a factor of $k + 1$ per round, such that the number of nodes to which the item has been broadcast after d rounds is at most $(k + 1)^d$.
- The total amount of data to be received by node j is $n - n_j$, and since this can be received over k simultaneously active ports, a lower bound is $\max_{0 \leq j < p} \lceil \frac{n-n_j}{k} \rceil$. For the regular case this is $\lceil \frac{n-n/p}{k} \rceil = \lceil \frac{(p-1)n}{pk} \rceil$.

In the linear cost model, a lower bound for the allgather operation is therefore $\lceil \log_{k+1} p \rceil \alpha + \max_{0 \leq j < p} \lceil \frac{(n-n_j)}{k} \rceil \beta$. For regular allgather problems this simplifies to

$$\lceil \log_{k+1} p \rceil \alpha + \lceil \frac{(p-1)n}{pk} \rceil \beta.$$

The model approximations abstract from any specific network topology, and for specific networks, including networks with a hierarchical communication structure, better, more precise lower bounds can sometimes be established. The network diameter will for instance provide another lower bound on the number of communication rounds.

Algorithms

At first it is assumed that $k = 1$ and that the allgather is regular, that is, the size n_i of each subvector x_i is equal to n/p . A brief survey of practically useful, common algorithmic approaches to the problem follows.

Ring, Mesh, and Hypercube

A flexible class of algorithms can be described by first viewing the nodes as connected as a ring where node i sends to node $(i + 1) \bmod p$ and receives from node $(i - 1) \bmod p$. An allgather operation can be accomplished in $p - 1$ communication rounds. In round j , $0 \leq j < p - 1$, node i sends subvector $x_{(i-j) \bmod p}$ and receives subvector $x_{(i-1-j) \bmod p}$.

The cost of this algorithm is

$$(p - 1) \left(\alpha + \frac{n}{p} \beta \right) = (p - 1)\alpha + \frac{(p - 1)n}{p} \beta.$$

This simple algorithm achieves the lower bound in the second term, and is useful when the vector size n is large. The linear, first term is far from the lower bound.

If the p nodes are instead organized in a $r_0 \times r_1$ mesh, with $p = r_0 r_1$, the complete allgather operation can be accomplished by first gathering (simultaneously) along the first dimension, and secondly gathering the larger subvectors along the second dimension. The cost of this algorithm becomes

$$\begin{aligned} & (r_1 - 1) \left(\alpha + \frac{n}{p} \beta \right) + (r_0 - 1) \left(\alpha + r_1 \frac{n}{p} \beta \right) \\ &= (r_1 + r_0 - 2)\alpha + \frac{(r_1 - 1)n + (r_0 - 1)r_1 n}{p} \beta \\ &= (r_1 + r_0 - 2)\alpha + \frac{(p - 1)n}{p} \beta. \end{aligned}$$

Generalizing the approach, if the p nodes are organized in a d -dimension with $p = r_0 \times r_1 \times \dots \times r_{d-1}$, the complete allgather operation can be accomplished by d successive allgather operations along the d dimensions. The cost of this algorithm becomes

$$\begin{aligned} & (r_{d-1} - 1) \left(\alpha + \frac{n}{p} \beta \right) + (r_{d-2} - 1) \left(\alpha + r_{d-1} \frac{n}{p} \beta \right) + \dots \\ &= \left(\sum_{j=0}^{d-1} (r_j - 1)\alpha \right) + \frac{(p - 1)n}{p} \beta. \end{aligned}$$

Notice that as the number of dimensions increases, the α term decreases while the (optimal) β term does not change.

If $p = 2^d$ so that $d = \log_2 p$ this approach yields an algorithm with cost

$$\left(\sum_{j=0}^{\log_2 p - 1} (2 - 1)\alpha \right) + \frac{(p - 1)n}{p} \beta = (\log_2 p)\alpha + \frac{(p - 1)n}{p} \beta.$$

The allgather in each dimension now involves only two nodes, and becomes a bidirectional exchange of data. This algorithm maps easily to hypercubes, and conversely a number of essentially identical algorithms were originally developed for hypercube systems. For fully connected networks it is restricted to the situation where p is a power of two, and does not easily,

```

 $j \leftarrow 1$ 
 $j' \leftarrow 2$ 
while  $j' < p$  do
    /* next round */
    par/* simultaneous send-receive */
        Send subvector  $(x_i, x_{(i+1) \bmod p}, \dots, x_{(i+j-1) \bmod p})$  to node  $(i-j) \bmod p$ 
        Receive subvector  $(x_{(i+j) \bmod p}, x_{(i+j+1) \bmod p}, \dots, x_{(i+j+j'-1) \bmod p})$  from node  $(i+j) \bmod p$ 
    endpar
     $j \leftarrow j'$ 
     $j' \leftarrow 2j'$ 
endwhile
/* last subvector */
 $j' \leftarrow p-j$ 
par/* simultaneous send-receive */
    Send subvector  $(x_i, x_{(i+1) \bmod p}, \dots, x_{(i+j'-1) \bmod p})$  to node  $(i-j) \bmod p$ 
    Receive subvector  $(x_{(i+j+1) \bmod p}, x_{(i+j+1) \bmod p}, \dots, x_{(i+j+j'-1) \bmod p})$  from node  $(i+j) \bmod p$ 
endpar

```

Allgather. Fig. 2 The dissemination allgather algorithm for node i , $0 \leq i < p$, and $1 < p$

without loss of theoretical performance, generalize to arbitrary values of p . It is sometimes called the *bidirectional exchange* algorithm, and relies only on restricted, *telephone-like* bidirectional communication. For this algorithm both the α and the β terms achieve their respective lower bounds. In contrast to, for instance, the broadcast operation, optimality can be achieved without the use of pipelining techniques.

Dissemination Allgather

On networks supporting bidirectional, fully connected, single-ported communication the allgather problem can be solved in the optimal $\lceil \log_2 p \rceil$ number of communication rounds for any p as shown in Fig. 2. In round k , node i communicates with nodes $(i + 2^k) \bmod p$ and $(i - 2^k) \bmod p$, and the size of the subvectors sent and received doubles in each round, with the possible exception of the last. Since each node sends and receives $p-1$ subvectors each of n/p elements the total cost of the algorithm is

$$\lceil \log_2 p \rceil \alpha + \frac{(p-1)n}{p} \beta,$$

for any number of nodes p , and meets the lower bound for single-ported communication systems. It can be generalized optimally (for some k) to k -ported communication systems.

This scheme is useful in many settings for implementation of other collective operations. The

$\lceil \log_2 p \rceil$ -regular communication pattern is an instance of a so-called *circulant* graph.

Composite Algorithms

Different, sometimes practically efficient algorithms for the allgather operation can be derived by combinations of algorithms for *broadcast* and *gather*. The full vector can be gathered at a chosen root node and broadcast from this node to all other nodes. This approach is inherently a factor of two off from the optimal number of communication rounds, since both gather and broadcast requires at least $\lceil \log p \rceil$ communication rounds even for fully connected networks. Other algorithms can be derived from broadcast or gather, by doing these operations simultaneously with each of the p nodes acting as either broadcast or gather root node.

Related Entries

- [Broadcast](#)
- [Collective Communication](#)
- [Message Passing Interface \(MPI\)](#)

Bibliographic Notes and Further Reading

The allgather operation is a symmetric variant of the broadcast operation, and together with this, one

of the most studied collective communication operations. Early theoretical surveys under different communication assumptions can be found in [5, 7, 8, 11]. For (near-)optimal algorithms for hypercubes, meshes, and tori, see [10, 18]. Practical implementations, for instance, for MPI for a variety of parallel systems have frequently been described, see, for instance, [1, 2, 12, 16]. Algorithms and implementations that exploit multi-port communication capabilities can be found in [2, 4, 14]. Algorithms based on ring shifts are sometimes called cyclic or bucket algorithms; in [3, 13] it is discussed how to create hybrid algorithms for meshes of lower dimension and fully connected architectures where the number of nodes is not a power of two. The ideas behind these algorithms date back to the early days of distributed memory architectures [6, 15]. The dissemination allgather algorithm is from the fundamental work of Bruck et al. [2], although the term is not used in that paper [1, 9]. Attention has almost exclusively been given to the regular variant of the problem. A pipelined algorithm for very large, *irregular* allgather problems was given in [17]. More general bibliographic notes can be found in the entry on collective communication.

Bibliography

1. Benson GD, Chu C-W, Huang Q, Caglar SG (2003) A comparison of MPICH allgather algorithms on switched networks. In: Recent advances in parallel virtual machine and message passing interface, 10th European PVM/MPI users' group meeting. Lecture notes in computer science, vol 2840. Springer, Berlin, pp 335–343
2. Bruck J, Ho C-T, Kipnis S, Upfal E, Weathersby D (1997) Efficient algorithms for all-to-all communications in multipoint message-passing systems. IEEE Trans Parallel Distrib Syst 8(11):1143–1156
3. Chan E, Heimlich M, Purkayastha A, van de Geijn RA (2007) Collective communication: theory, practice, and experience. Concurrency Comput: Pract Exp 19(13):1749–1783
4. Chan E, van de Geijn RA, Gropp W, Thakur R (2006) Collective communication on architectures that support simultaneous communication over multiple links. In: ACM SIGPLAN symposium on principles and practice of parallel programming (PPoPP), ACM, New York, pp 2–11
5. Chen M-S, Chen J-C, Yu PS (1996) On general results for all-to-all broadcast. IEEE Trans Parallel Distrib Syst 7(4):363–370
6. Fox G, Johnson M, Lyzenga G, Otto S, Salmon J, Walker D (1988) Solving problems on concurrent processors, vol I. Prentice-Hall, Englewood Cliffs
7. Fraigniaud P, Lazard E (1994) Methods and problems of communication in usual networks. Discret Appl Math 53(1–3):79–133
8. Hedetniemi SM, Hedetniemi T, Liestman AL (1988) A survey of gossiping and broadcasting in communication networks. Networks 18:319–349
9. Hensgen D, Finkel R, Manber U (1988) Two algorithms for barrier synchronization. Int J Parallel Program 17(1):1–17
10. Ho C-T, Kao M-Y (1995) Optimal broadcast in all-port wormhole-routed hypercubes. IEEE Trans Parallel Distrib Syst 6(2):200–204
11. Krumme DW, Cybenko G, Venkataraman KN (1992) Gossiping in minimal time. SIAM J Comput 21(1):111–139
12. Mamidala AR, Vishnu A, Panda DK (2006) Efficient shared memory and RDMA based design for MPI Allgather over InfiniBand. In: Recent advances in parallel virtual machine and message passing interface, 13th European PVM/MPI users' group meeting. Lecture notes in computer science, vol 4192. Springer, Berlin, pp 66–75
13. Mitra P, Payne DG, Schuler L, van de Geijn R (1995) Fast collective communication libraries, please. In: Intel Supercomputer Users' Group Meeting, University of Texas, 22 June 1995
14. Qian Y, Afshari A (2007) RDMA-based and SMP-aware multi-port all-gather on multi-rail QsNetII SMP clusters. In: International conference on parallel processing (ICPP 2007) Xi'an, China, p. 48
15. Saad Y, Schultz MH (1989) Data communication in parallel architectures. Parallel Comput 11(2):131–150
16. Träff JL (2006) Efficient allgather for regular SMP-clusters. In: Recent advances in parallel virtual machine and message passing interface, 13th European PVM/MPI users' group meeting, Lecture notes in computer science, vol 4192. Springer, Berlin, pp 58–65
17. Träff JL, Ripke A, Siebert C, Balaji P, Thakur R, Gropp W (2010) A pipelined algorithm for large, irregular all-gather problems. Int J High Perform Comput Appl 24(1):58–68
18. Yang Y, Wang J (2002) Near-optimal all-to-all broadcast in multi-dimensional all-port meshes and tori. IEEE Trans Parallel Distrib Syst 13(2):128–141

All-to-All

JESPER LARSSON TRÄFF¹, ROBERT A. VAN DE GEIJN²

¹University of Vienna, Vienna, Austria

²The University of Texas at Austin, Austin, TX, USA

Synonyms

Complete exchange; Index; Personalized all-to-all exchange; Transpose

Definition

Among a set of processing elements (nodes) each node has distinct (personalized) data items destined for each of the other nodes. The all-to-all operation accomplishes this total data exchange among the set of

nodes, such that each node ends up having an individual data item from each of the other nodes.

Discussion

The reader may consider first visiting the entry on *collective communication*.

Let the p nodes be indexed consecutively, $0, 1, \dots, p - 1$. Initially each node i has a (column)vector of data $x^{(i)}$ that is further subdivided into subvectors $x_j^{(i)}$ for $j = 0, \dots, p - 1$. The subvector $x_j^{(i)}$ is to be sent to node j from node i . Upon completion of the all-to-all exchange operation node i will have the vector consisting of the subvectors $x_i^{(j)}$ for $j = 0, \dots, p - 1$. In effect, the matrix consisting of the i columns $x^{(i)}$ is *transposed* with the i th row consisting of subvectors $x_i^{(j)}$ originally distributed over the p nodes $j = 0, \dots, p - 1$ now transferred to node i . This *transpose* all-to-all operation is illustrated in Fig. 1. The subvectors $x_i^{(i)}$ do not have to be communicated, but for symmetry reasons it is convenient to think of the operation as if each node also contributes a subvector to itself.

The all-to-all exchange can be interpreted as an actual matrix transpose operation in the case where all subvectors $x_i^{(j)}$ have the same number of elements n . In that case the all-to-all problem is called *regular* and the operation is also sometimes termed *index*. The all-to-all operation is, however, also well defined in cases where subvectors have different number of elements. Subvectors could for instance differ for each row index j , e.g., $n_j = |x_j^{(i)}|$ for all nodes i , or each subvector could have a possibly different number of elements $n_{ij} = |x_j^{(i)}|$ without any specified relation between the subvector sizes. In all such cases, the all-to-all problem is called *irregular*. Likewise, the subvectors $x_j^{(i)}$ can be structured objects, for instance matrix blocks. It is common to assume that for both regular and irregular all-to-all problems each

Before			After		
Node 0	Node 1	Node 2	Node 0	Node 1	Node 2
$x_0^{(0)}$	$x_0^{(1)}$	$x_0^{(2)}$	$x_0^{(0)}$	$x_1^{(0)}$	$x_2^{(0)}$
$x_1^{(0)}$	$x_1^{(1)}$	$x_1^{(2)}$	$x_0^{(1)}$	$x_1^{(1)}$	$x_2^{(1)}$
$x_2^{(0)}$	$x_2^{(1)}$	$x_2^{(2)}$	$x_0^{(2)}$	$x_1^{(2)}$	$x_2^{(2)}$

All-to-All. Fig. 1 All-to-all communication for three nodes

node knows not only the number of elements of all subvectors it has to send to other nodes, but also the number of elements in all subvectors it has to receive from other nodes. This can be assumed without loss of generality, since a regular all-to-all operation with subvectors of size one can be used to exchange and collect the required information on number of elements to be sent and received.

All-to-all communication is the most general and most communication intensive collective data-exchange communication pattern. Since it is allowed that some $|x_j^{(i)}| = 0$ and that some subvectors $x_j^{(i)}$ are identical for some i and j , the irregular all-to-all operation subsumes other common *collective communication* patterns like broadcast, gather, scatter, and allgather. Algorithms for specific patterns are typically considerably more efficient than general, all-to-all algorithms, which motivates the inclusion of a spectrum of collective communication operations in collective communication interfaces and libraries. The *Message-Passing Interface* (MPI), for instance, has both regular MPI_Alltoall and irregular MPI_Alltoallv and MPI_Alltoallw operations in its repertoire, as well as operations for the specialized operations broadcast, gather, scatter, and allgather, and structured data can be flexibly described by so-called user-defined datatypes.

All-to-all communication is required in FFT computations, matrix transposition, generalized permutations, etc., and is thus a fundamental operation in a large number of scientific computing applications.

Lower Bounds

The (regular) all-to-all communication operation requires that each node exchanges data with each other node. Therefore, a lower bound on the all-to-all communication time will be determined by a minimal *bisection* and the *bisection bandwidth* of the communication system or network. The number of subvectors that have to cross any bisection of the nodes (i.e., partition into two roughly equal-sized subsets) is $p^2/2$ (for even p), namely $p/2 \times p/2 = p^2/4$ subvectors from each subset of the partition. The number of communication links that can be simultaneously active in a minimal partition determines the number of communication rounds that are at the least needed to transfer all subvectors across the bisection assuming that subvectors are

not combined. A d -dimensional, symmetric torus with unidirectional communication has a bisection of $2k^{d-1}$ with $k = \sqrt[d]{p}$, and the number of required communication rounds is therefore $p\sqrt[d]{p}/4$. A hypercube (which can also be construed as a $\log_2 p$ dimensional mesh) has bisection $p/2$. The low bisection of the torus limits the bandwidth that can be achieved for all-to-all communication. If the bisection bandwidth is B vector elements per time unit, any all-to-all algorithms requires at least $np^2/2B$ units of time.

Assume now a fully connected, homogeneous, k -ported, bidirectional send-receive communication network. Each node can communicate directly with any other node at the same cost, and can at the same time receive data from at most k nodes and send distinct data to at most k , possibly different nodes. In this model, the following general lower bounds on the number of communication rounds and the amount of data transferred in sequence per node can be proved. Here n is the amount of data per subvector for each node.

- A complete all-to-all exchange requires at least $\lceil \log_{k+1} p \rceil$ communication rounds and transfers at least $\lceil \frac{n(p-1)}{k} \rceil$ units of data per node.
- Any all-to-all algorithm that uses $\lceil \log_{k+1} p \rceil$ communication rounds must transfer at least $\Omega(\frac{np}{k+1} \log_{k+1} p)$ units of data per node.
- Any all-to-all algorithm that transfers exactly $\frac{n(p-1)}{k}$ units of data per node requires at least $\frac{p-1}{k}$ communication rounds.

These lower bounds bound the minimum required time for the all-to-all operation. In contrast to, for instance, allgather communication, there is a trade-off between the number of communication rounds and the amount of data transferred. In particular, it is not possible to achieve a logarithmic number of communication rounds *and* a linear number of subvectors per node. Instead, fewer rounds can only be achieved at the expense of combining subvectors and sending the same subvector several times. As a first approximation, communication cost is modeled by a linear cost function such that the time to transfer n units of data is $\alpha + n\beta$ for a start-latency α and cost per unit β . All-to-all communication time is then at least α times the number of communication rounds plus β times the units of data transferred in sequence by a node.

Algorithms

In the following, the regular all-to-all operation is considered. Each node i has a subvector $x_j^{(i)}$ to transfer to each other node j , and all subvectors have the same number of elements $n = |x_i^{(j)}|$.

Fully Connected Systems

In a *direct algorithm* each node i sends each subvector directly to its destination node j . It is assumed that communication takes place in rounds, in which all or some nodes send and/or receive data. The difficulty is to organize the communication in such a way that no node stays idle for too many rounds, and that in the same round no node is required to send or receive more than the k subvectors permitted by the k -ported assumption.

For one-ported, bidirectional systems the simplest algorithm takes $p-1$ communication rounds. In round r , $1 \leq r < p$, node i sends subvector $x_{(i+r) \bmod p}^{(i)}$ to node $(i+r) \bmod p$, and receives subvector $x_i^{((i-r) \bmod p)}$ from node $(i-r) \bmod p$. If required, subvector $x_i^{(i)}$ is copied in a final noncommunication round. This algorithm achieves the lower bound on data transfer, at the expense of $p-1$ communication rounds, and trivially generalizes to k -ported systems. In this case, the last round may not be able to utilize all k communication ports.

For single-ported systems ($k = 1$) with weaker bidirectional communication capabilities allowing only that each node i sends and receives from the same node j (often referred to as the *telephone model*) in a communication round, or even unidirectional communication capabilities allowing each node to only either send or receive in one communication round, a different algorithm solves the problem. Such an algorithm is described in the following.

A fully connected communication network can be viewed as a complete, undirected graph with processing nodes modeled as graph nodes. For even p the complete graph can be partitioned into $p-1$ disjoint 1-factors (perfect matchings), each of which associates each node i with a different node j . For odd p this is not possible, but the formula $j = (r - i) \bmod p$ for $r = 0, \dots, p-1$ associates a different node j with i for each round r such that if i is associated with j in round r , then j is associated with i in the same round. In each round there is exactly one node that becomes associated with itself.

This can be used to achieve the claimed 1-factorization for even p . Let the last node $i = p - 1$ be *special*. Perform the $p - 1$ rounds on the $p - 1$ non-special nodes. In the round where a non-special node becomes associated with itself, it instead performs an exchange with the special node. It can be shown that p and $p - 1$ communication rounds for odd and even p , respectively, is *optimal*. The algorithm is depicted in Fig. 2. It takes $p - 1$ communication rounds for even p , and p rounds for odd p . If a self-copy is required it can for odd p be done in the round where a node is associated with itself and for even p either before or after the exchange. Note that in the case where p is even the formula $j = (r - i) \bmod p$

```

if odd( $p$ ) then
    for  $r \leftarrow 0, 1, \dots, p - 1$  do
         $j \leftarrow (r - i) \bmod p$ 
        if  $i \neq j$  then
            par/* simultaneous send-receive */
            Send subvector  $x_j^{(i)}$  to node  $j$ 
            Receive subvector  $x_i^{(j)}$  from node  $j$ 
        end par
    end if
    end for
else if  $i < p - 1$  then /*  $p$  even */
    /* non-special nodes */
    for  $r \leftarrow 0, 1, \dots, p - 2$  do
         $j \leftarrow (r - i - 1) \bmod (p - 1)$ 
        if  $i = j$  then  $j \leftarrow p - 1$ 
        par/* simultaneous send-receive */
        Send subvector  $x_j^{(i)}$  to node  $j$ 
        Receive subvector  $x_i^{(j)}$  from node  $j$ 
    end par
    end for
else /* special node */
    for  $r \leftarrow 0, 1, \dots, p - 2$  do
        if even ( $r$ ) then  $j \leftarrow r/2$  else  $j \leftarrow (p-1+r)/2$ 
        par/* simultaneous send-receive */
        Send subvector  $x_j^{(i)}$  to node  $j$ 
        Receive subvector  $x_i^{(j)}$  from node  $j$ 
    end par
    end for
end if

```

All-to-All. Fig. 2 Direct, telephone-like all-to-all communication based on 1-factorization of the complete p node graph. In each communication round r each node i becomes associated with a unique node j with which it performs an exchange. The algorithm falls into three special cases for odd and even p , respectively, and for the latter for nodes $i < p - 1$ and the special node $i = p - 1$. For odd p the number of communication rounds is p , and $p - 1$ for even p . In both cases, this is optimal in the number of communication rounds

also pairs node i with itself in some round, and the self-exchange could be done in this round. This would lead to an algorithm with p rounds, in some of which some nodes perform the self-exchange. If bidirectional communication is not supported, each exchange between nodes i and j can be accomplished by the smaller numbered node sending and then receiving from the larger numbered node, and conversely the larger numbered node receiving and then sending to the smaller numbered node.

If p is a power of two the pairing $j = i \oplus r$, where \oplus denotes the bitwise exclusive-or operation, likewise produces a 1-factorization and this has often been used.

Hypercube

By combining subvectors the number of communication rounds and thereby the number of message start-ups can be significantly reduced. The price is an increased communication volume because subvectors will have to be sent multiple times via intermediate nodes. Such *indirect* algorithms were pioneered for hypercubes and later extended to other networks and communication models. In particular for the fully connected, k -ported model algorithms exist that give the optimal trade-off (for many values of k and p) between the number of communication rounds and the amount of data transferred per node.

A simple, indirect algorithm for the d -dimensional hypercube that achieves the lower bound on the number of communication rounds at the expense of a logarithmic factor more data is as follows. Each hypercube node i communicates with each of its d neighbors, and the nodes pair up with their neighbors in the same fashion. In round r , node i pairs up and performs an exchange with neighbor $j = i \oplus 2^{d-r}$ for $r = 1, \dots, d$. In the first round, node i sends in one message the 2^{d-1} subvectors destined to the nodes of the $d - 1$ -dimensional subcube to which node j belongs. It receives from node j the 2^{d-1} subvectors for the $d - 1$ -dimensional subcube to which it itself belongs. In the second round, node i pairs up with a neighbor of a $d - 2$ -dimensional subcube, and exchanges the 2^{d-2} own subvectors and the additional 2^{d-2} subvectors for this subcube received in the first round. In general, in each round r node i receives and sends $2^r 2^{d-1-r} = 2^{d-1} = p/2$ subvectors.

Total cost of this algorithm in the linear cost model is $\log_2 p(\alpha + \beta \frac{p}{2} n)$.

Irregular All-to-All Communication

The irregular all-to-all communication problem is considerably more difficult both in theory and in practice. In the general case with no restrictions on the sizes of the subvectors to be sent and received, finding communication schedules that minimize the number of communication rounds and/or the amount of data transferred is an NP-complete optimization problem. For problem instances that are not too irregular a decomposition into a sequence of more regular problems and other collective operations sometimes work, and such approaches have been used in practice. Heuristics and approximation algorithms for many variations of the problem have been considered in the literature. For concrete communication libraries, a further practical difficulty is that full information about the problem to be solved, that is the sizes of *all* subvectors for all the nodes is typically not available to any single node. Instead each node knows only the sizes of the subvectors it has to send (and receive). To solve the problem this information has to be centrally gathered entailing a sequential bottleneck, or distributed algorithms or heuristics for computing a communication schedule must be employed.

Related Entries

- [Allgather](#)
- [Collective Communication](#)
- [FFT \(Fast Fourier Transform\)](#)
- [Hypercubes and Meshes](#)
- [MPI \(Message Passing Interface\)](#)

Bibliographic Notes and Further Reading

All-to-all communication has been studied since the early days of parallel computing, and many of the results presented here can be found in early, seminal work [6, 15]. Classical references on indirect all-to-all communication for hypercubes are [7, 16], see also later [2] for combinations of different approaches leading to *hybrid* algorithms. The generalization to arbitrary node counts for the k -ported, bidirectional communication model was developed in [3], which also proves

the lower bounds on number of rounds and the trade-off between amount of transferred data and required number of communication rounds. Variants and combinations of these algorithms have been implemented in various MPI like libraries for collective communication [1, 14, 19].

A proof that a 1-factorization of the p node complete graph exists when p is even can be found in [6] and elsewhere.

All-to-all algorithms not covered here for meshes and tori can be found in [17, 18, 20, 21], and algorithms for multistage networks in [11, 22]. Lower bounds for tori and meshes based on counting link load can be found in [9]. Implementation considerations for some of these algorithms for MPI for the Blue Gene systems can be found in [8].

Irregular all-to-all exchange algorithms and implementations have been considered in [12, 13] (decomposition into a series of easier problems) and later in [5, 10], the former summarizing many complexity results.

Bibliography

1. Bala V, Bruck J, Cypher R, Elustondo P, Ho A, Ho CT, Kipnis S, Snir M (1995) CCL: a portable and tunable collective communications library for scalable parallel computers. *IEEE T Parall Distr* 6(2):154–164
2. Bokhari SH (1996) Multiphase complete exchange: a theoretical analysis. *IEEE T Comput* 45(2):220–229
3. Bruck J, Ho CT, Kipnis S, Upfal E, Weathersby D (1997) Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE T Parall Distr* 8(11):1143–1156
4. Fox G, Johnson M, Lyzenga G, Otto S, Salmon J, Walker D (1988) Solving problems on concurrent processors, vol I. Prentice-Hall, Englewood Cliffs
5. Goldman A, Peters JG, Trystram D (2006) Exchanging messages of different sizes. *J Parallel Distr Com* 66(1):1–18
6. Harary F (1969) Graph theory. Addison-Wesley, Reading, Mass
7. Johnsson SL, Ho CT (1989) Optimum broadcasting and personalized communication in hypercubes. *IEEE T Comput* 38(9):1249–1268
8. Kumar S, Sabharwal Y, Garg R, Heidelberger P (2008) Optimization of all-to-all communication on the blue gene/l supercomputer. In: International conference on parallel processing (ICPP), Portland, pp 320–329
9. Lam CC, Huang CH, Sadayappan P (1997) Optimal algorithms for all-to-all personalized communication on rings and two dimensional tori. *J Parallel Distr Com* 43(1):3–13
10. Liu W, Wang CL, Prasanna VK (2002) Portable and scalable algorithm for irregular all-to-all communication. *J Parallel Distr Com* 62:1493–1526

11. Massini A (2003) All-to-all personalized communication on multistage interconnection networks. *Discrete Appl Math* 128(2–3):435–446
12. Ranka S, Wang JC, Fox G (1994) Static and run-time algorithms for all-to-many personalized communication on permutation networks. *IEEE T Parall Distr* 5(12):1266–1274
13. Ranka S, Wang JC, Kumar M (1995) Irregular personalized communication on distributed memory machines. *J Parallel Distr Com* 25(1):58–71
14. Ritzdorf H, Träff JL (2006) Collective operations in NEC's high-performance MPI libraries. In: International parallel and distributed processing symposium (IPDPS 2006), p 100
15. Saad Y, Schultz MH (1989) Data communication in parallel architectures. *Parallel Comput* 11(2):131–150
16. Scott DS (1991) Efficient all-to-all communication patterns in hypercube and mesh topologies. In: Proceedings 6th conference on distributed memory concurrent Computers, pp 398–403
17. Suh YJ, Shin KG (2001) All-to-all personalized communication in multidimensional torus and mesh networks. *IEEE T Parall Distr* 12(1):38–59
18. Suh YJ, Yalamanchili S (1998) All-to-all communication with minimum start-up costs in 2D/3D tori and meshes. *IEEE T Parall Distr* 9(5):442–458
19. Thakur R, Gropp WD, Rabenseifner R (2004) Improving the performance of collective operations in MPICH. *Int J High Perform C* 19:49–66
20. Tseng YC, Gupta SKS (1996) All-to-all personalized communication in a wormhole-routed torus. *IEEE T Parall Distr* 7(5): 498–505
21. Tseng YC, Lin TH, Gupta SKS, Panda DK (1997) Bandwidth-optimal complete exchange on wormhole-routed 2D/3D torus networks: A diagonal-propagation approach. *IEEE T Parall Distr* 8(4):380–396
22. Yang Y, Wang J (2000) Optimal all-to-all personalized exchange in self-routable multistage networks. *IEEE T Parall Distr* 11(3): 261–274
23. Yang Y, Wang J (2002) Near-optimal all-to-all broadcast in multidimensional all-port meshes and tori. *IEEE T Parall Distr* 13(2):128–141

AMD Opteron Processor Barcelona

BENJAMIN SANDER

Advanced Micro Device Inc., Austin, TX, USA

Synonyms

[Microprocessors](#)

Definition

The AMD Opteron™ processor codenamed “Barcelona” was introduced in September 2007 and was notable as the industry’s first “native” quad-core x86 processor, containing four cores on single piece of silicon. The die also featured an integrated L3 cache which was accessible by all four cores, Rapid Virtualization Indexing to improve virtualization performance, new power management features including split power planes for the integrated memory controller and the cores, and a higher-IPC (Instructions per Clock) core including significantly higher floating-point performance. Finally, the product was a plug-in replacement for the previous-generation AMD Opteron processor – “Barcelona” used the same Direct Connect system architecture, the same physical socket including commodity DDR2 memory, and fit in approximately the same thermal envelope.

Discussion

Previous Opteron Generations

The first-generation AMD Opteron processor was introduced in April of 2003. AMD Opteron introduced the AMD64 instruction set architecture, which was an evolutionary extension to the x86 instruction set that provided additional registers and 64-bit addressing [1]. An important differentiating feature of AMD64 was that it retained compatibility with the large library of 32-bit x86 applications which had been written over the 30-year history of that architecture – other 64-bit architectures at that time either did not support the x86 ISA or supported it only through a slow emulation mode. The first-generation AMD Opteron also introduced Direct Connect architecture which featured a memory controller integrated on the same die as the

All-to-All Broadcast

► [Allgather](#)

Altivec

► [IBM Power Architecture](#)

► [Vector Extensions, Instruction-Set Architecture \(ISA\)](#)

processor, and HyperTransport™ Technology bus links to directly connect the processors and enable glueless two-socket and four-socket multiprocessor systems.

The second-generation AMD Opteron processor was introduced two years later in April of 2005 and enhanced the original offering by integrating two cores (Dual-Core) with the memory controller. The dual-core AMD Opteron was later improved with DDR2 memory for higher bandwidth, and introduced AMD-V™ hardware virtualization support.

“Barcelona” introduced the third generation of the AMD Opteron architecture, in September of 2007.

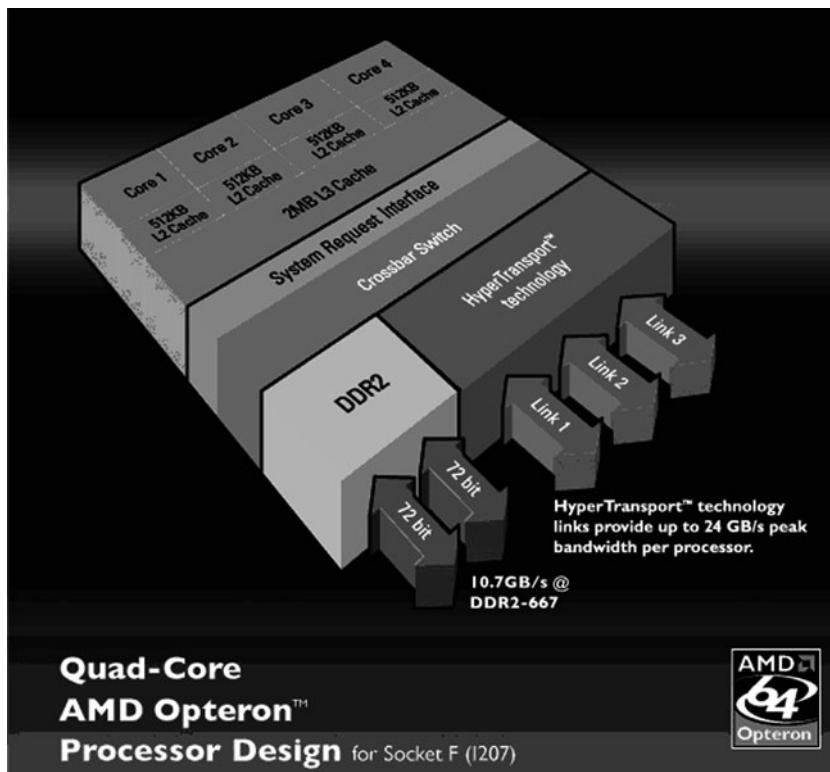
System-on-a-Chip

“Barcelona” was the industry’s first native quad-core processor – as shown in Fig. 1, the processor contained four processing cores (each with a private 512 k L2 cache), a shared 2 MB L3 cache, an integrated crossbar which enabled efficient multiprocessor communication, and DDR2 memory controller and three HyperTransport links. The tight integration on a single die enabled efficient communication between all four cores, notably

through an improved power management flexibility and the shared L3 cache.

The L3 cache was a non-inclusive architecture – the contents of the L2 caches were independent and typically not replicated in the L3, enabling more efficient use of the cache hierarchy. The L2 and L3 caches were designed as a victim-cache architecture. A newly retrieved cache line would initially be written to the L1 cache. Eventually, the processor would evict the line from the L1 cache and would then write it into the L2 cache. Finally, the line would be evicted from the L2 cache and the processor would write it to the L3 cache.

The shared cache used a “sharing-aware” fill and replacement policy. Consider the case where a line-fill request hits in the L3 cache: The processor has the option of leaving a copy of the line in the L3 (predicting that other cores are likely to want that line in the future) or moving the line to the requesting core, invalidating it from the L3 to leave a hole which could be used by another fill. Lines which were likely to be shared (for example, instruction code or lines which had been shared in the past) would leave a copy of the line in



AMD Opteron Processor Barcelona. Fig. 1 Quad-Core AMD Opteron™ processor design

the L3 cache for other cores to access. Lines which were likely to be private (for example, requested with write permission or which had never been previously shared in the past) would move the line from the L3 to the L1. The L3 cache maintained a history of which lines had been shared to guide the fill policy. Additionally, the sharing history information influenced the L3 replacement policy – the processor would preferentially keep lines which had been shared in the past by granting them an additional trip through the replacement policy.

In Client CPUs, AMD later introduced a “Triple-Core” product based on the same die used by “Barcelona” – this product served an important product category (for users who valued an affordable product with more than dual-core processing power). The Triple-Core product was targeted at the consumer market and was based on the same die used by “Barcelona” with three functional cores rather than four.

Socket Compatibility

“Barcelona” was socket-compatible with the previous-generation Dual-Core AMD OpteronTM product: “Barcelona” had the same pinout, used the same two channels of DDR2 memory, and also fit in a similar thermal envelope enabling the use of the same cooling and thermal solution. Even though “Barcelona” used four cores (rather than two) and each core possessed twice the peak floating-point capability of the previous generation, “Barcelona” was able to fit in the same thermal envelope through the combination of a smaller process (“Barcelona” was the first AMD server product based on 65 nm SOI technology) and a reduction in peak operating frequency (initially “Barcelona” ran at 2.3 GHz). The doubling of core density in the same platform was appealing and substantially increased performance on many important server workloads compared to the previous-generation AMD Opteron product. Customers could upgrade their AMD Opteron products to take advantage of quad-core “Barcelona,” and OEMs could leverage much of their platform designs from the previous generation.

Delivering More Memory Bandwidth

As mentioned above, the “Barcelona” processor’s socket-compatibility eased its introduction to the marketplace. However, the four cores on a die placed additional load on the memory controller – the platform had the

same peak bandwidth as the previous generation: Each socket still had two channels of DDR2 memory, running at up to 667 MHz and delivering a peak bandwidth of 10.7 GB/s (per socket). To feed the additional cores, the memory controller in “Barcelona” included several enhancements which improved the *delivered* bandwidth.

One notable feature was the introduction of “independent” memory channels. In the second-generation AMD Opteron product, each memory channel serviced half of a 64-byte memory request – i.e., in parallel the memory controller would read 32 bytes from each channel. This was referred to as a “ganged” controller organization and has the benefit of perfectly balancing the memory load between the two memory channels. However, this organization also effectively reduces the number of available dram banks by a factor of two – when a dram page is opened on the first channel, the controller opens the same page on the other channel at the same time. Effectively, the ganged organization creates pages which are twice as big, but provides half as many dram pages as a result.

With four cores on a die all running four different threads, the memory accesses tend to be unrelated and more random. DRAMs support only a small number of open banks, and requests which map to the same bank but at different addresses create a situation called a “page conflict.” The page conflict can dramatically reduce the efficiency and delivered bandwidth of the DRAM, because the DRAM has to repeatedly switch between the multiple pages which are competing for the same open bank resources. Additionally, write traffic coming from the larger multi-level “Barcelona” cache hierarchy created another level of mostly random memory traffic which had to be efficiently serviced. All of these factors led to a new design in which the two memory channels were controlled independently – i.e., each channel could independently service an entire 64-byte cache line rather than using both channels for the same cache line. This change enabled the two channels to independently determine which pages to open, effectively doubling the number of banks and enabling the design to better cope with the more complex memory stream coming from the quad-core processor.

A low-order address bit influenced the channel selection, which served to spread the memory traffic between the two controllers and avoid overloading

one of the channels. The design hashed the low-order address bit with other higher-order address bit to spread the traffic more evenly, and in practice, applications showed an near-equal allocation between the two channels, enabling the benefits of the extra banks provided by the independent channels along with equal load balancing. The independent channels also resulted in a longer burst length (twice as long as the ganged organization), which reduced pressure on the command bus; essentially each Read or Write command performed twice as much work with the independent organization. This was especially important for some DIMMs which required commands to be sent for two consecutive cycles (“2T” mode).

“Barcelona” continued to support a dynamic open-page policy, in which the memory controller leaves dram pages in the “open” state if the pages are likely to be accessed in the future. As compared to a closed-page design (which always closes the dram pages), the dynamic open-page design can improve latency and bandwidth in cases where the access stream has locality to the same page (by leaving the page open), and also deliver best-possible bandwidth when the access stream contains conflicts (by recognizing the page-conflict stream and closing the page). “Barcelona” introduced a new predictor which examined the history of accesses to each bank to determine whether to leave the page open or to close it. The predictor was effective at increasing the number of page hits (delivering lower latency) and reducing the number of page conflicts (improving bandwidth).

“Barcelona” also introduced a new DRAM prefetcher which monitored read traffic and prefetched the next cache line when it detected a pattern. The prefetcher had sophisticated pattern detection logic which could detect both forward and backward patterns, unit and non-unit strides, as well as some more complicated patterns. The prefetcher targeted a dedicated buffer to store the prefetched data (rather than a cache) and thus the algorithm could aggressively exploit unused dram bandwidth without concern for generating cache pollution. The prefetcher also had a mechanism to throttle the prefetcher if the prefetches were inaccurate or if the dram bandwidth was consumed by non-prefetch requests. Later generations of the memory controller would improve on both the prefetch and the throttling mechanisms.

The DRAM controller was also internally re-plumbed with wider busses (the main busses were increased from 64-bits to 128-bits) and with additional buffering. Many of these changes served to prepare the controller to support future higher-speeds memory technologies. One notable change was the addition of write-bursting logic, which buffered memory writes until a watermark level was achieved, at which time the controller would burst all the writes to the controller. As compared to trickling the writes to the DRAM as they arrived at the memory controller, the write-bursting mode was another bandwidth optimization. Typically, the read and write requests address different banks, so switching between a read mode and a write mode both minimizes the number of read/write bus turnarounds and minimizes the associated page open/close traffic.

“Barcelona” Core Architecture

The “Barcelona” core was based on the previous “K8” core design but included comprehensive IPC (Instruction-Per-Clock) improvements throughout the entire pipeline. One notable feature was the “Wide Floating-Point Accelerator,” which doubled the raw computational floating-point data paths and floating-point execution units from 64-bits to 128-bits. A 2.5 GHz “Barcelona” core possessed a peak rate of 20 GFlops of single-precision computation; the quad-core “Barcelona” could then deliver 80 GFlops at peak (twice as many cores, each with twice as much performance). This doubling of each core’s raw computation bandwidth was accompanied by a doubling of the instruction fetch bandwidth (from 16-bytes/cycle to 32-bytes/cycle) and a doubling of the data cache bandwidth (two 128-bit loads could be serviced each clock cycle), enabling the rest of the pipeline to feed the new higher-bandwidth floating-point units. Notably, SSE instructions include one or more prefix bytes, frequently use the REX prefix to encode additional registers, and can be quite large. The increased instruction fetch bandwidth was therefore important to keep the pipeline balanced and able to feed the high-throughput wide floating-point units.

“Barcelona” introduced a new “unaligned SSE mode,” which allowed a single SSE instruction to both load and execute an SSE operation, without concern for alignment. Previously, users had to use two instructions

(an unaligned load followed by an execute operation). This new mode further reduced pressure on the instruction decoders and also reduced register pressure. The mode relaxed a misguided attempt to simplify the architecture by penalizing unaligned operations and was later adopted as the x86 standard. Many important algorithms such as video decompression can benefit from vector instructions but cannot guarantee alignment in the source data (for example, if the input data is compressed).

The “Barcelona” pipeline included an improved branch predictor, using more bits for the global history to improve the accuracy and also doubling the size of the return stack. “Barcelona” added a dedicated predictor for indirect branches to improve performance when executing virtual functions commonly used in modern programming styles. The wider 32-byte instruction fetch improved both SSE instruction throughput as well as higher throughput in codes with large integer instructions, particularly when using some of the more complicated addressing modes.

“Barcelona” added a Sideband Stack Optimizer feature which executed common stack operations (i.e., the PUSH and POP instructions) with dedicated stack adjustment logic. The logic broke the serial dependence chains seen in consecutive strings of PUSH and POP instructions (common at function entry and exit), and also freed the regular functional units to execute other operations.

“Barcelona” also improved the execution core with a data-dependent divide, which provided an early out for the common case where the dividend was small. “Barcelona” introduced the SSE4a instruction set, which added a handful of instructions including leading-zero count and population count, bit INSERT and EXTRACT, and streaming single-precision store operations.

“Barcelona” core added an out-of-order load feature, which enabled loads to bypass other loads in the pipeline. Other memory optimizations included a wider L2 bus, larger data and instruction TLBs, 1GB page size, and 48-bit physical address to support large server database footprints. The L1 data TLB was expanded to 48 entries, and each entry could hold any of the three page sizes in the architecture (4K, 2M, or 1GB); this provided flexibility for the architecture to efficiently run applications with both legacy and newer page sizes.

Overall the “Barcelona” core was a comprehensive but evolutionary improvement over the previous design. The evolutionary improvement provided a consistent optimization strategy for compilers and software developers: optimizations for previous-generation AMD Opteron™ processors were largely effective on the “Barcelona” core as well.

Virtualization and Rapid Virtualization Indexing

In 2007, virtualization was an emerging application class driven by customer desire to more efficiently utilize multi-core server systems and thus was an important target for the quad-core “Barcelona.” One performance bottleneck in virtualized applications was the address translation performed by the hypervisor – the hypervisor virtualizes the physical memory in the system and thus has to perform an extra level of address translation between the guest physical and the actual host physical address. Effectively, the hypervisor creates another level of page tables for this final level of translation. Previous-generation processors performed this translation with a software-only technique called “shadow paging.” Shadow paging required a large number of hypervisor intercepts (to maintain the page tables) which slowed performance and also suffered from an increase in the memory footprint. “Barcelona” introduced Rapid Virtualization Indexing (also known as “nested paging”) which provided hardware support for performing the final address translation; effectively, the hardware was aware of both the host and guest page tables and could walk both as needed [2]. “Barcelona” also provided translation caching structures to accelerate the nested table walk.

“Barcelona” continued to support AMD-V™ hardware virtualization support, tagged TLBs to reduce TLB flushing when switching between guests, and the Device Exclusion Vector for security. Additionally, well-optimized virtualization applications typically demonstrate a high degree of local memory accesses, i.e., accesses to the integrated memory controller rather than to another socket in the multi-socket system. AMD’s Direct Connect architecture, which provided lower latency and higher bandwidth for local memory accesses, was thus particularly well suited for running virtualization applications.

Power Reduction Features

The “Barcelona” design included dedicated power supplies for the CPU cores and the memory controller, allowing the voltage for each to be separately controlled. This allowed the cores to operate at reduced power consumption levels while the memory controller continued to run at full speed and service memory requests from other cores in the system. In addition, the core design included the use of fine-gaters to reduce the power to logic on the chip which was not currently in use. One example was the floating-point unit – for integer code, when the floating-point unit was not in use, “Barcelona” would gate the floating-point unit and significantly reduce the consumed power. The fine-gaters could be re-enabled in a single cycle and did not cause any visible increased latency.

The highly integrated “Barcelona” design also reduced the overall system chip count and thus reduced system power. Notably the AMD Opteron™ system architecture integrated the northbridge on the same die as the processor (reducing system chip count by one device present in some legacy system architectures), and also used commodity DDR2 memory (which consumed less power than the competing FB-DIMM standard).

Future Directions

The AMD Opteron™ processor codenamed “Barcelona” was the third generation in the AMD Opteron processor line. “Barcelona” was followed by the AMD Opteron processor codenamed “Shanghai,” which was built in 45 nm SOI process technology, included a larger 6 M shared L3 cache, further core and northbridge performance improvements, faster operating frequencies, and faster DDR2 and HT interconnect frequencies. “Shanghai” was introduced in November of 2008.

“Shanghai” was followed by the AMD Opteron processor codenamed “Istanbul,” which integrated six cores onto a single die, and again plugged into the same socket as “Barcelona” and “Shanghai.” “Istanbul” also included an “HT Assist” feature which substantially reduced probe traffic in the system. HT Assist adds cache directory to each memory controller; the directory tracks lines in the memory range serviced by the memory controller which are cached somewhere in the system. Frequently, a memory access misses the directory (indicating the line is not cached anywhere in the

system) and thus the system can immediately return the requested data without having to probe the system. HT Assist enables the AMD Opteron platform to efficiently scale bandwidth to 4-socket and 8-socket server systems.

The next AMD Opteron™ product is on the near horizon as well. Codenamed “Magny-Cours,” this processor is planned for introduction in the first quarter of 2010, includes up to 12-cores in each socket, and introduces a new G34 platform. Each G34 socket contains 4 DDR3 channels and 4 HyperTransport links. “Magny-Cours” continues to use evolved versions of the core and memory controller that were initially introduced in “Barcelona.”

“Barcelona” broke new ground as the industry’s first native quad-core device, including a shared L3 cache architecture and leadership memory bandwidth. The “Barcelona” core was designed with an evolutionary approach, and delivered higher core performance (especially on floating-point codes), and introduced new virtualization technologies to improve memory translation performance and ease Hypervisor implementation. Finally, “Barcelona” was plug-compatible with the previous-generation AMD Opteron processors, leveraging the stable AMD Direct Connect architecture and cost-effective commodity DDR2 memory technology.

Bibliography

1. Advanced Micro Devices, Inc. x86-64™ Technology White Paper. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/x86-64_wp.pdf
2. Advanced Micro Devices, Inc. (2008) AMD-V™ Nested Paging. <http://developer.amd.com/assets/NPT-WP-1%201-final-TM.pdf>
3. Advanced Micro Devices, Inc. (2009) AMD64 architecture tech docs. http://www.amd.com/us-en/Processors/DevelopWithAMD/0_30_2252_739_7044_00.html
4. Sander B (2006) Core optimizations for system-level performance. <http://www.instat.com/fallmpf/06/conf1.htm> <http://www.instat.com/Fallmpf/06/>

Amdahl's Argument

- [Amdahl's Law](#)

Amdahl's Law

JOHN L. GUSTAFSON
Intel Labs, Santa Clara, CA, USA

Synonyms

Amdahl's argument; Fixed-size speedup; Law of diminishing returns; Strong scaling

Definition

Amdahl's Law says that if you apply P processors to a task that has serial fraction f , the predicted net speedup is

$$\text{Speedup} = \frac{1}{f + \frac{1-f}{P}}.$$

More generally, it shows the speedup that results from applying any performance enhancement by a factor of P to only one part of a given workload.

A corollary of Amdahl's Law, often confused with the law itself, is that even when one applies a very large number of processors P (or other performance enhancement) to a problem, the net improvement in speed cannot exceed $1/f$.

Discussion

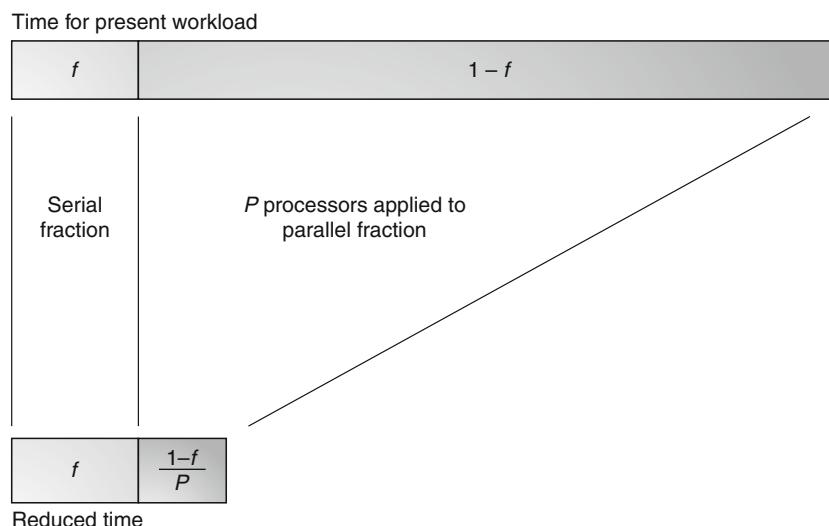
Graphical Explanation

The diagram in Fig. 1 graphically explains the formula in the definition.

The model sets the time required to solve the present workload (top bar) to unity. The part of the workload that is serial, f , is unaffected by parallelization. (See discussion below for the effect of including the time for interprocessor communication.) The model assumes that the remainder of the time, $1 - f$, parallelizes perfectly so that it takes only $1/P$ as much time as on the serial processor. The ratio of the top bar to the bottom bar is thus $1/(f + (1-f)/P)$.

History

In the late 1960s, research interest increased in the idea of achieving higher computing performance by using many computers working in parallel. At the Spring 1967 meeting of the American Federation of Information Processing Societies (AFIPS), organizers set up a session entitled "The best approach to large computing capability – A debate." Daniel Slotnick presented "Unconventional Systems," a description of a 256-processor ensemble controlled by a single instruction stream, later known as the ILLIAC IV [12]. IBM's chief architect, Gene Amdahl, presented a counterargument entitled "Validity of the single processor approach to achieving large scale computing capabilities" [1]. It



Amdahl's Law. Fig. 1 Graphical explanation of Amdahl's Law

was in this presentation that Amdahl made a specific argument about the merits of serial mainframes over parallel (and pipelined) computers.

The formula known as Amdahl's Law does *not* appear anywhere in that paper. Instead, the paper shows a hand-drawn graph that includes the performance speedup of a 32-processor system over a single processor, as the fraction of parallel work increases from 0% to 100%. There are no numbers or labels on the axes in the original, but Fig. 2 reproduces his graph more precisely.

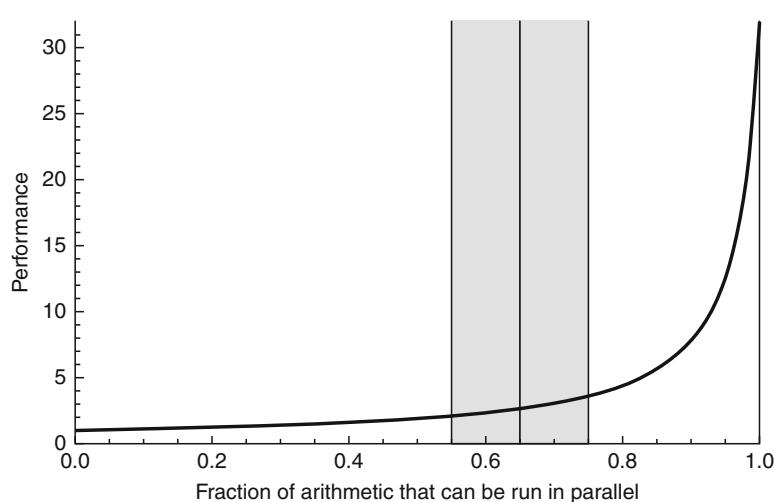
Amdahl estimated that about 10% of an algorithm was inherently serial, and data management imposed another 25% serial overhead, which he showed as the gray area in the figure centered about 65% parallel content. He asserted this was the most probable region of operation. From this, he concluded that a parallel system like the one Slotnick described would only yield from about 2X to 4X speedup. At the debate, he presented the formula used to produce the graph, but did not include it in the text of the paper itself.

This debate was so influential that in less than a year, the computing community was referring to the argument against parallel computing as "Amdahl's Law." The person who first coined the phrase may have been Willis H. Ware, who in 1972 first put into print the phrase "Amdahl's Law" and the usual form of the formula, in a RAND report titled "The Ultimate Computer" [14].

The argument rapidly became part of the commonly accepted guidelines for computer design, just as the Law

of Diminishing Returns is a classic guideline in economics and business. In the early 1980s, computer users attributed the success of Cray Research vector computers over rivals such as those made by CDC to Cray's better attention to Amdahl's Law. The Cray designs did not take vector pipelining to such extreme levels relative to the rest of their system and often got a higher fraction of peak performance as a result. The widely used textbook on computer architecture by Hennessy and Patterson [7] harkens back to the traditional view of Amdahl's Law as guidance for computer designers, particularly in its earlier editions.

The formula Amdahl used was simply the use of elementary algebra to combine two different speeds, here defined as *work* per unit time, not *distance* per unit time: it simply compares two cases of the net speed as the total work divided by the total time. This common result is certainly not due to Amdahl, and he was chagrined at receiving credit for such an obvious bit of mathematics. "Amdahl's Law" really refers to the argument that the formula (along with its implicit assumptions about typical serial fractions and the way computer costs and workloads scale) predicts harsh limits on what parallel computing can achieve. For those who wished to avoid the change to their software that parallelism would require, either for economic or emotional reasons, Amdahl's Law served as a technical defense for their preference.



Amdahl's Law. Fig. 2 Amdahl's original 32-processor speedup graph (reconstructed)

Estimates of the “Serial Fraction” Prove Pessimistic

The algebra of Amdahl's Law is unassailable since it describes the fundamental way speeds add algebraically, for a fixed amount of work. However, the estimate of the serial fraction f (originally 35%, give or take 10%) was only an estimate by Amdahl and was not based on mathematics.

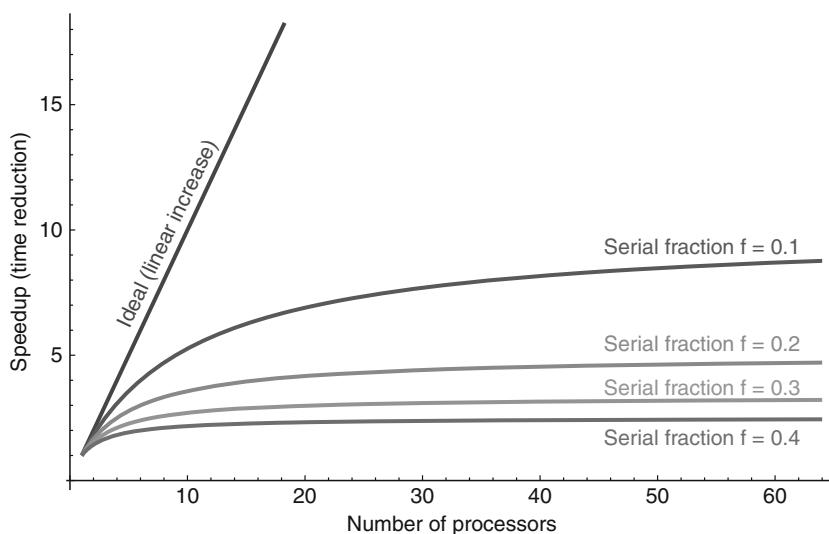
There exist algorithms that are inherently almost 100% serial, such as a time-stepping method for a physics simulation involving very few spatial variables. There are also algorithms that are almost 100% parallel, such as ray tracing methods for computer graphics, or computing the Mandelbrot set. It therefore seems *reasonable* that there might be a rather even distribution of serial fraction from 0 to 1 over the entire space of computer applications. The following figure shows another common way to visualize the effects of Amdahl's Law, with speedup as a function of the number of processors. Figure 3 shows performance curves for serial fractions 0.1, 0.2, 0.3, and 0.4 for a 64-processor computer system.

The limitations of Amdahl's Law for performance prediction were highlighted in 1985, when IBM scientist Alan Karp publicized a skeptical challenge (and a token award of \$100) to anyone who could demonstrate a speedup of over 200 times on three real computer applications [9]. He had just returned from a conference at which startup companies nCUBE and Thinking

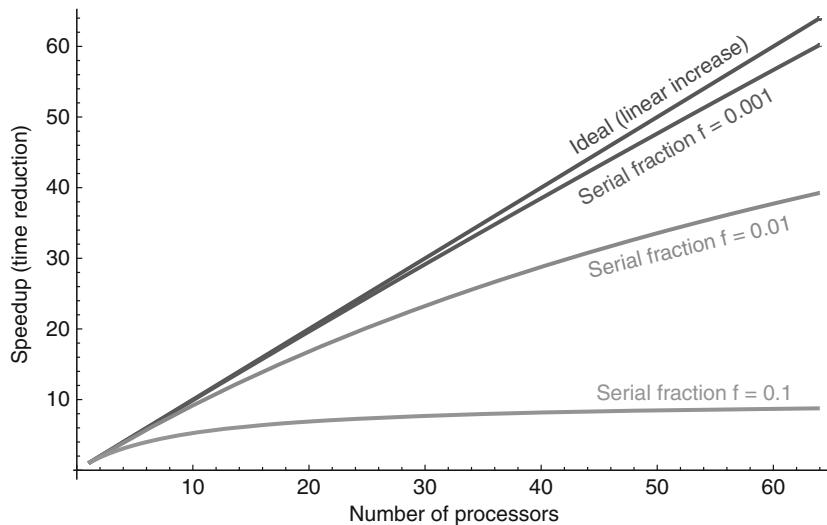
Machines had announced systems with over 1,000 processors, and Karp gave the community 10 years to solve the problem putting a deadline at the end of 1995 to achieve the goal. Karp suggested fluid dynamics, structural analysis, and econometric modeling as the three application areas to draw from, to avoid the use of contrived and unrealistic applications. The published speedups of the 1985 era tended to be less than tenfold and used applications of little economic value (like how to place N queens on a chessboard so that no two can attack each other).

The Karp Challenge was widely distributed by e-mail but received no responses for years, suggesting that if 200-fold speedups were possible, they required more than a token amount of effort. C. Gordon Bell, also interested in promoting the advancement computing, proposed a similar challenge but with two alterations: He raised the award to \$1,000, and said that it would be given annually to the greatest parallel speedup achieved on three real applications, but only awarded if the speedup was at least twice that of the previous award. This definition was the original Gordon Bell Prize [2], and Bell envisioned that the first award might be for something close to tenfold speedup, with increasingly difficult advances after that.

By late 1987, Sandia scientists John Gustafson, Gary Montry, and Robert Benner undertook to demonstrate high parallel speedup on applications from fluid



Amdahl's Law. Fig. 3 Speedup curves for large serial fractions



Amdahl's Law. Fig. 4 Speedup curves for smaller serial fractions

dynamics, structural mechanics, and acoustic wave propagation. They recognized that an Amdahl-type speedup (now called “strong scaling”) was more challenging to achieve than some of the speedups claimed for distributed memory systems like Caltech’s Cosmic Cube [5] that altered the problem according to the number of processors in use. However, using a 1,024-processor nCUBE 10, the three Sandia researchers were able to achieve performance on 1,024 processors ranging from 502X to 637X that of a single processor running the same size problem, implying the Amdahl serial fraction was only 0.0006–0.001 for those applications. This showed that the historical estimates of values for the serial fraction in Amdahl’s formula might be far too high, at least for some applications. While the mathematics of Amdahl’s Law is unassailable, it was a poorly substantiated opinion that the actual values of the serial fraction for computing workloads would always be too high to permit effective use of parallel computing.

Figure 4 shows Amdahl’s Law, again for a 64-processor system, but with serial fractions of 0.1, 0.01, and 0.001.

The TOP500 list ranks computers by their ability to solve a dense system of linear equations. In November 2009, the top-ranked system (Jaguar, Oak Ridge National Laboratories) achieved over 75% parallel efficiency using 224,256 computing cores. For Amdahl’s Law to hold, the serial fraction must be about *one part per million* for this system.

Observable Fraction and Superlinear Speedup

For many scientific codes, it is simple to instrument and measure the amount of time f spent in serial execution. One can place timers in the program around serial regions and obtain an estimate of f that might or might not strongly depend on the input data. One can then apply this fraction for Amdahl’s Law estimates of time reduction, or Gustafson’s Law estimates of scaled speedup. Neither law takes into account communication costs or intermediate degrees of parallelism.

A more common practice is to measure the parallel speedup as the number of processors is varied, and fit the resulting curve to derive f . This approach may yield some guidance for programmers and hardware developers, but it confuses serial fraction with communication overhead, load imbalance, changes in the relative use of the memory hierarchy, and so on. The term “strong scaling” refers to the requirement to keep the problem size the same for any number of processors. A common phenomenon that results from “strong scaling” is that when spreading a problem across more and more processors, the *memory per processor* goes down to the point where the data fits entirely in cache, resulting in *superlinear speedup* [4]. Sometimes, the superlinear speedup effects and the communication overheads partially cancel out, so what appears to be a low value of f is actually the result of the combination of the two effects. In modern parallel systems, performance

analysis with Amdahl's original law alone will usually be inaccurate, since so many other parallel processing phenomena have large effects on the speedup.

Impact on Parallel Computing

Even at the time of the 1967 AFIPS conference, there was already enough investment in serial computing software that the prospect of rewriting all of it to use parallelism was quite daunting. Amdahl's Law served as a strong defense against having to rewrite the code to exploit multiple processors. Efforts to create experimental parallel computer systems proceeded in the decades that followed, especially in academic or research laboratory settings, but the major high-performance computer companies like IBM, Digital, Cray, and HP did not create products with a high degree of parallelism, and cited Amdahl's Law as the reason. It was not until the 1988 solution to the Karp Challenge, which made clear that Amdahl's Law need not limit the utility of highly parallel computing, that vendors began developing commercial parallel products in earnest.

Implicit Assumptions, and Extensions to the Law

Fixed Problem Size

The assumption that the computing community overlooked for 20 years was that the problem size is fixed. If one applies many processors (or other performance enhancement) to a workload, it is not necessarily true that users will keep the workload fixed and accept shorter times for the execution of the task. It is common to increase the workload on the faster machine to the point where it takes the same amount of time as before.

In comparing two things, the scientific approach is to control all but one variable. The natural choice when comparing the performance of two computations is to run the same problem in both situations and look for a change in the execution time. If speed is w/t where w is work and t is time, then speedup for two situations is the ratio of the speeds: $(w_1/t_1)/(w_2/t_2)$. By keeping the work the same, that is, $w_1 = w_2$, the speedup simplifies to t_2/t_1 , and this avoids the difficult problem of defining "work" for a computing task. Amdahl's Law uses this "fixed-size speedup" assumption. While the assumption is reasonable for small values of speedup, it

is less reasonable when the speeds differ by many orders of magnitude. Since the execution time of an application tends to match human patience (which differs according to application), people might scale the problem such that the *time* is constant and thus is the controlled variable. That is, $t_1 = t_2$, and the speedup simplifies to w_1/w_2 . See *Gustafson's Law*.

System Cost: Linear with the Number of Processors?

Another implicit assumption is that system cost is linear in the number of processors, so anything less than perfect speedup implies that cost-effectiveness goes down every time an approach uses more processors. At the time Amdahl made his argument in 1967, this was a reasonable assumption: a system with two IBM processors would probably have cost almost exactly twice that of an individual IBM processor. Amdahl's paper even states that "... by putting two processors side by side with shared memory, one would find approximately 2.2 times as much hardware," where the additional 0.2 hardware is for sharing the memory with a crossbar switch. He further estimated that memory conflicts would add so much time that net price performance of a dual-processor system would be 0.8 that of a single processor.

His cost assumptions are not valid for present-era system designs. As Moore's Law has decreased the cost of transistors to the point where a single silicon chip holds many processors in the same package that formerly held a single processor, it is apparent that system costs are far below linear in the number of processors. Processors share software and other facilities that can cost much more than individual processor cores. Thus, while Amdahl's algebraic formula is true, the implications it provided in 1967 for optimal system design have changed. For example, it might be that increasing the number of processors by a factor of 4 only provides a net speedup of 1.3X for the workload, but if the quadrupling of processors only increases system cost by 1.2X, the cost-effectiveness of the system increases with parallelism. Put another way, the point of diminishing returns for adding processors in 1967 might have been a single processor. With current economics, it might be a very large number of processors, depending on the application workload.

All-or-None Parallelism

In the Amdahl model, there are only two levels of concurrency for the use of N processors: N -fold parallel or serial. A more realistic and detailed model recognizes that the amount of exploitable parallelism might vary from one to N processors through the execution of a program [13]. The speedup is then

$$\text{Speedup} = 1 / (f_1 + f_2/2 + f_3/3 + \dots + f_N/N),$$

where f_j is the fraction of the program that can be run on j processors in parallel, and $f_1 + f_2 + \dots + f_N = 1$.

Sharing of Resources

Because the parallelism model was that of multiple processors controlled by a single instruction stream, Amdahl formulated his argument for the parallelism of a single job, not the parallelism of multiple users running multiple jobs. For parallel computers with multiple instruction streams, if the duration of a serial section is longer than the time it takes to swap in another job in the queue, there is no reason that $N - 1$ of the N processors need to go idle as long as there are users waiting for the system. As the previous section mentions, the degree of parallelism can vary throughout a program. A sophisticated queuing system can allocate processing resources to other users accordingly, much as systems partition memory dynamically for different jobs.

Communication Cost

Because Amdahl formulated his argument in 1967, he treated the cost of *communication* of data between processors as negligible. At that time, computer arithmetic took so much longer than data motion that the data motion was overlapped or insignificant. Arithmetic speed has improved much more than the speed of interprocessor communication, so many have improved Amdahl's Law as a performance model by incorporating communication terms in the formula.

Some have suggested that communication costs are part of the serial fraction of Amdahl's Law, but this is a misconception. Interprocessor communication can be serial or parallel just as the computation can. For example, a communication algorithm may ask one processor to send data to all others in sequence (completely serial) or it may ask each processor j to send data to

processor $j - 1$, except that processor 1 sends to processor N , forming a communication ring (completely parallel).

Analogies

Without mentioning Amdahl's Law by name, others have referred, often humorously, to the limitations of parallel processing. Fred Brooks, in *The Mythical Man-Month* (1975), pointed out the futility of trying to complete software projects in less time by adding more people to the project [3]. "Brooks' Law" is his observation that adding engineers to a project can actually make the project take longer. Brooks quoted the well-known quip, "Nine women can't have a baby in one month," and may have been the first to apply that quip to computer technology.

From 1898 to 1906, Ambrose Bierce wrote a collection of cynical definitions called *The Devil's Dictionary*. It includes the following definition:

- ▶ Logic, *n.* The art of thinking and reasoning in strict accordance with the limitations and incapacities of human misunderstanding. The basis of logic is the syllogism, consisting of a major and a minor premise and a conclusion – thus:

Major Premise: Sixty men can do a piece of work 60 times as quickly as one man.

Minor Premise: One man can dig a post-hole in 60s; therefore –

Conclusion: Sixty men can dig a post-hole in 1s.

This may be called the syllogism arithmetical, in which, by combining logic and mathematics, we obtain a double certainty, and are twice blessed.

In showing the absurdity of using 60 processors (men) for an inherently serial task, he predated Amdahl by almost 70 years.

Transportation provides accessible analogies for Amdahl's Law. For example, if one takes a trip at 30 miles per hour and immediately turns around, how fast does one have to go to average 60 miles per hour? This is a trick question that many people incorrectly answer, "90 miles per hour." To average 60 miles per hour, one would have to travel back at infinite speed and *instantly*. For a fixed travel distance, just as for a fixed workload, speeds do not combine as a simple

arithmetic average. This is contrary to our intuition, which may be the reason some consider Amdahl's Law such a profound observation.

Perspective

Amdahl's 1967 argument became a justification for the avoidance of parallel computing for over 20 years. It was appropriate for many of the early parallel computer designs that shared an instruction stream or the memory fabric or other resources. By the 1980s, hardware approaches emerged that looked more like collections of autonomous computers that did not share anything yet were capable of cooperating on a single task. It was not until Gustafson published his alternative formulation for parallel speedup in 1988, along with several examples of actual 1,000-fold speedups from a 1024-processor system, that the validity of the parallel-computing approach became widely accepted outside the academic community. Amdahl's Law still is the best rule-of-thumb when the goal of the performance improvement is to reduce execution time for a fixed task, whereas Gustafson's Law is the best rule-of-thumb when the goal is to increase the problem size for a fixed amount of time. Amdahl's and Gustafson's Laws do not contradict one another, nor is either a corollary or equivalent of the other. They are for different assumptions and different situations.

Gene Amdahl, in a 2008 personal interview, stated that he never intended his argument to be applied to the case where each processor had its own operating system and data management, and would have been far more open to the idea of parallel computing as a viable approach had it been posed that way. He is now a strong advocate of parallel architectures and sits on the technical advisory board of Massively Parallel Technologies, Inc.

With the commercial introduction of single-image systems with over 100,000 processors such as Blue Gene, and clusters with similar numbers of server processor cores, it becomes increasingly unrealistic to use a fixed-size problem to compare the performance of a single processor with that of the entire system. Thus, scaled speedup (Gustafson's Law) applies to measure performance of the largest systems, with Amdahl's Law applied mainly where the number of processors changes over a narrow range.

Related Entries

- Brent's Theorem
- Gustafson's Law
- Metrics
- Pipelining

A

Bibliographic Notes and Further Reading

Amdahl's original 1967 paper is short and readily available online, but as stated in the Discussion section, it has neither the formula nor any direct analysis. An objective analysis of the Law and its implications can be found in [8] or [10]. The series of editions of the textbook on computer architecture by Hennessey and Patterson [7] began in 1990 with a strong alignment to Amdahl's 1967 debate position against parallel computing, and has evolved a less strident stance in more recent editions.

For a rigorous mathematical treatment of Amdahl's Law that covers many of the extensions and refinements mentioned in the Discussion section, see [13]. One of the first papers to show how fixed-sized speedup measurement is prone to superlinear speedup effects is [4].

A classic 1989 work on speedup and efficiency is "Speedup versus efficiency in parallel systems" by D. L. Eager, J. Zahorjan, and E. D. Lazowska, in *IEEE Transactions*, March 1989, 408–423. DOI=10.1109/12.21127.

Bibliography

1. Amdahl GM (1967) Validity of the single-processor approach to achieve large scale computing capabilities. AFIPS Joint Spring Conference Proceedings 30 (Atlantic City, NJ, Apr. 18–20), AFIPS Press, Reston VA, pp 483–485, At <http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf>
2. Bell G (interviewed) (July 1987) An interview with Gordon Bell. *IEEE Software*, 4(4):102–104
3. Brooks FP (1975) The mythical man-month: Essays on software engineering. Addison-Wesley, Reading. ISBN 0-201-00650-2
4. Gustafson JL (April 1990) Fixed time, tiered memory, and superlinear speedup. Proceedings of the 5th distributed memory conference, vol 2, pp 1255–1260. ISBN: 0-8186-2113-3
5. Gustafson JL, Montry GR, Benner RE (July 1988) Development of parallel methods for a 1024-processor hypercube. *SIAM J Sci Statist Comput*, 9(4):609–638
6. Gustafson (May 1988) Reevaluating Amdahl's law. *Commun ACM*, 31(5):532–533. DOI=10.1145/42411.42415

7. Hennessy JL, Patterson DA (1990, 1996, 2003, 2007) Computer architecture: A quantitative approach. Elsevier Inc.
8. Hwang K, Briggs F (1990) Computer architecture and parallel processing, McGraw-Hill, New York. ISBN: 0070315566
9. Karp A (1985) <http://www.netlib.org/benchmark/karp-challenge>
10. Lewis TG, El-Rewini H (1992) Introduction to parallel computing, Prentice Hall. ISBN: 0-13-498924-4, 32-33
11. Seitz CL (1986) Experiments with VLSI ensemble machines. Journal of VLSI and computer systems, vol 1. No. 3, pp 311–334
12. Slotnick D (1967) Unconventional systems. AFIPS joint spring conference proceedings 30 (Atlantic City, NJ, Apr. 18–20). AFIPS Press, Reston VA, pp 477–481
13. Sun X-H, Ni L (1993) Scalable problems and memory-bounded speedup. Journal of parallel and distributed computing, vol 19. No 1, pp 22–37
14. Ware WH (1972) The Ultimate Computer. IEEE spectrum, vol 9. No. 3, pp 84–91

AMG

- Algebraic Multigrid

Analytics, Massive-Scale

- Massive-Scale Analytics

Anomaly Detection

- Race Detection Techniques
- Intel Parallel Inspector

Anton, A Special-Purpose Molecular Simulation Machine

RON O. DROR¹, CLIFF YOUNG¹, DAVID E. SHAW^{1,2}

¹D. E. Shaw Research, New York, NY, USA

²Columbia University, New York, NY, USA

Definition

Anton is a special-purpose supercomputer architecture designed by D. E. Shaw Research to dramatically

accelerate molecular dynamics (MD) simulations of biomolecular systems. Anton performs massively parallel computation on a set of identical MD-specific ASICs that interact in a tightly coupled manner using a specialized high-speed communication network. Anton enabled, for the first time, the simulation of proteins at an atomic level of detail for periods on the order of a millisecond – about two orders of magnitude beyond the previous state of the art – allowing the observation of important biochemical phenomena that were previously inaccessible to both computational and experimental study.

Discussion

Introduction

Classical molecular dynamics (MD) simulations give scientists the ability to trace the motions of biological molecules at an atomic level of detail. Although MD simulations have helped yield deep insights into the molecular mechanisms of biological processes in a way that could not have been achieved using only laboratory experiments [17, 18], such simulations have historically been limited by the speed at which they can be performed on conventional computer hardware.

A particular challenge has been the simulation of functionally important biological events that often occur on timescales ranging from tens of microseconds to a millisecond, including the “folding” of proteins into their native three-dimensional shapes, the structural changes that underlie protein function, and the interactions between two proteins or between a protein and a candidate drug molecule. Such long-timescale simulations pose a much greater challenge than simulations of larger chemical systems at more moderate timescales: the number of processors that can be used effectively in parallel scales with system size but not with simulation length, because of the sequential dependencies within a simulation.

Anton, a specialized, massively parallel supercomputer developed by D. E. Shaw Research, accelerated such calculations by several orders of magnitude compared with the previous state of the art, enabling the simulation of biological processes on timescales that might otherwise not have been accessible for many years. The first 512-node Anton machine (Fig. 1), which became operational in late 2008, completed an all-atom



A

Anton, A Special-Purpose Molecular Simulation Machine. Fig. 1 A 512-node Anton machine

Anton, A Special-Purpose Molecular Simulation Machine. Table 1 The longest (to our knowledge) all-atom MD simulations of proteins in explicitly represented water published through the end of 2009

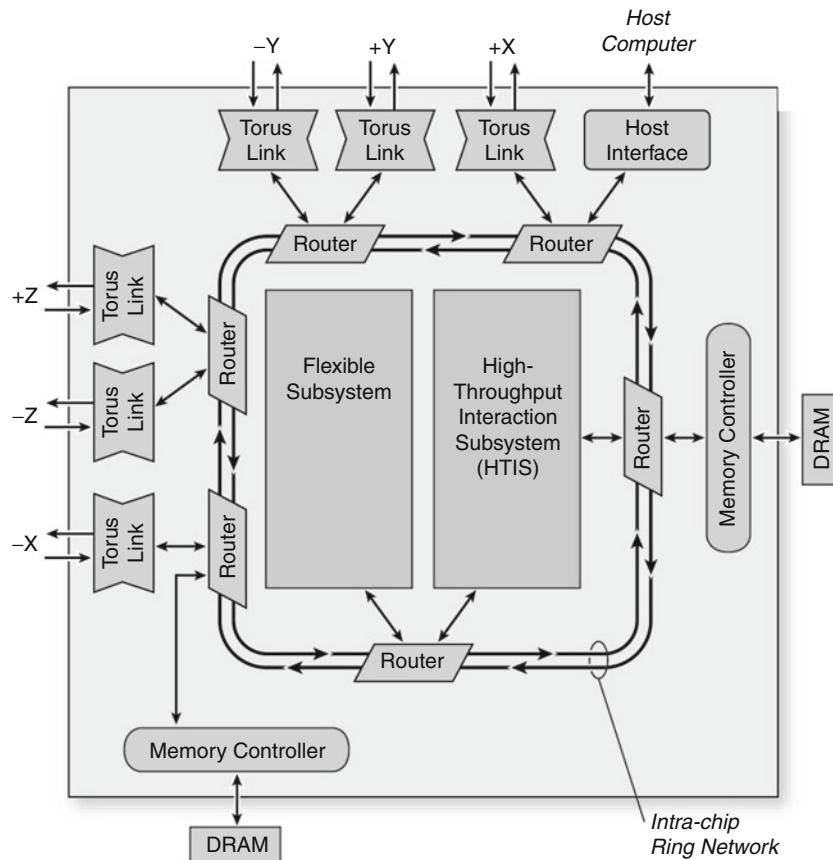
Length (μ s)	Protein	Hardware	Software	Citation
1,031	BPTI	Anton	[native]	[30]
236	gpW	Anton	[native]	[30]
10	WW domain	x86 cluster	NAMD	[9, 10]
9	Villin HP-35	x86 cluster	NAMD	[11]
2	Villin HP-35	x86 cluster	GROMACS	[6]
2	Rhodopsin	Blue Gene/L	Blue Matter	[12, 22]
2	β_2 AR	x86 cluster	Desmond	[4]

protein simulation spanning more than a millisecond of biological time in 2009 [30]. By way of comparison, the longest such simulation previously reported in the literature, which was performed on general-purpose computer hardware using the MD code NAMD, was 10 microseconds (μ s) in length [9]; at the time, few other published simulations had reached 2 μ s (Table 1).

An Anton machine comprises a set of identical processing nodes, each containing a specialized MD computation engine implemented as a single ASIC (Fig. 2). These processing nodes are connected through a specialized high-performance network to form a three-dimensional torus. Anton was designed to use both novel parallel algorithms and special-purpose logic to dramatically accelerate those calculations that dominate the time required for a typical MD simulation [29]. The remainder of the simulation algorithm

is executed by a programmable portion of each chip that achieves a substantial degree of parallelism while preserving the flexibility necessary to accommodate anticipated advances in physical models and simulation methods.

Anton was created to attack a somewhat different problem than the ones addressed by several other projects that have deployed significant computational resources for MD simulations. The Folding@Home project [24], for example, uses hundreds of thousands of PCs (made available over the Internet by volunteers) to simulate a very large number of *separate* molecular trajectories, each of which is limited to the timescale accessible on a single PC. While a great deal can be learned from a large number of independent MD trajectories, many other important problems require the examination of a single, very long trajectory – the principal



Anton, A Special-Purpose Molecular Simulation Machine. Fig. 2 Block diagram of a single Anton ASIC, comprising the specialized high-throughput interaction subsystem, the more general-purpose flexible subsystem, six inter-chip torus links, an intra-chip communication ring, and two memory controllers

task for which Anton was designed. Other projects have produced special-purpose hardware (e.g., FAST-RUN [7], MDGRAPE [33], and MD Engine [34]) to accelerate the most computationally expensive elements of an MD simulation. Such hardware reduces the effective *cost* of simulating a given period of biological time, but Amdahl's law and communication bottlenecks prevent the efficient use of enough such chips in parallel to extend individual simulations beyond timescales of a few microseconds.

Anton was named after Anton van Leeuwenhoek, often referred to as the “father of microscopy.” In the seventeenth century, van Leeuwenhoek built high-precision optical instruments that allowed him to visualize bacteria and other microorganisms, as well as blood cells and spermatozoa, revealing for the first time an entirely new biological world. In pursuit of an

analogous goal, Anton (the machine) was designed as a sort of “computational microscope,” providing contemporary biological and biomedical researchers with a tool for understanding organisms and their diseases at previously inaccessible spatial and temporal scales. Anton has enabled substantial advances in the study of the processes by which proteins fold, function, and interact with drugs [26, 31].

Structure of a Molecular Dynamics Computation

An MD simulation computes the motion of a collection of atoms – for example, a protein surrounded by water – over a period of time according to the laws of classical physics. Time is broken into a series of discrete *time steps*, each representing a few femtoseconds of simulated time. For each time step, the simulation

performs a computationally intensive force calculation for each atom, followed by a less expensive integration operation that advances the positions and velocities of the atoms.

Forces are evaluated based on a model known as a *force field*. Anton supports a variety of commonly used biomolecular force fields, which express the total force on an atom as a sum of three types of component forces: (1) bonded forces, which involve interactions between small groups of atoms connected by one or more covalent bonds; (2) van der Waals forces, which include interactions between all pairs of atoms in the system, but which fall off quickly with distance and are typically only evaluated for nearby pairs of atoms; and (3) electrostatic forces, which include interactions between all pairs of charged atoms, and fall off slowly with distance.

Electrostatic forces are typically computed by one of several fast, approximate methods that account for long-range effects without requiring the explicit interaction of all pairs of atoms. Anton, like most MD codes for general-purpose hardware, divides electrostatic interactions into two contributions. The first decays rapidly with distance, and is thus computed directly for all atom pairs separated by less than some cutoff radius. This contribution and the van der Waals interactions together constitute the *range-limited interactions*. The second contribution (*long-range interactions*) decays more slowly, but can be expressed as a convolution and efficiently computed using fast Fourier transforms (FFTs) [27]. This process requires the mapping of charges from atoms to nearby mesh points before the FFT computations (*charge spreading*), and the calculation of forces on atoms based on values associated with nearby mesh points after the FFT computations (*force interpolation*).

The Role of Specialization in Anton

During the five years spent designing and building Anton, the number of transistors on a chip increased by roughly tenfold, as predicted by Moore's law. Anton, on the other hand, enabled simulations approximately 1,000 times faster than was possible at the beginning of that period, providing access to biologically critical millisecond timescales significantly sooner than would have been possible on commodity hardware. Achieving this performance required reengineering

how MD is done, simultaneously considering changes to algorithms, software, and, especially, hardware.

Hardware specialization allows Anton to redeploy resources in ways that benefit MD. Compared to other high-performance computing applications, MD uses much computation and communication but surprisingly little memory. Anton exploits this property by using only SRAMs and small first-level caches on the ASIC, constraining all code and data to fit on-chip in normal operation (for chemical systems that exceed SRAM size, Anton pages state to each node's local DRAM). The area that would have been spent on large caches and aggressive memory hierarchies is instead dedicated to computation and communication. Each Anton ASIC contains dedicated, specialized hardware datapaths to evaluate the range-limited interactions and perform charge spreading and force interpolation, packing much more computational logic on a chip than is typical of general-purpose architectures. Each ASIC also contains programmable processors with specialized instruction set architectures tailored to the remainder of the MD computation. Anton's specialized network fabric not only delivers bandwidth and latency two orders of magnitude better than Gigabit Ethernet, but also sustains a large fraction of peak network bandwidth when delivering small packets and provides hardware support for common MD communication patterns such as multicast [5].

The most computationally intensive parts of an MD simulation – in particular, the electrostatic interactions – are also the most well established and unlikely to change as force field models evolve, making these calculations particularly amenable to hardware acceleration. Dramatically speeding up MD, however, requires that one accelerates more than just an “inner loop.” Calculation of electrostatic and van der Waals forces accounts for roughly 90% of the computational time for a representative MD simulation on a single general-purpose processor. Amdahl's law states that no matter how much one accelerates this calculation, the remaining computations, left unaccelerated, would limit the maximum speedup to a factor of 10. Hence, Anton dedicates a significant fraction of silicon area to accelerating other tasks, such as bonded force computation and integration, incorporating programmability as appropriate to accommodate a variety of force fields and simulation features.

System Architecture

The building block of an Anton system is a node, which includes an ASIC with two major computational subsystems (Fig. 2). The first is the *high-throughput interaction subsystem* (HTIS), designed for computing massive numbers of range-limited pairwise interactions of various forms. The second is the *flexible subsystem*, which is composed of programmable cores used for the remaining, less structured part of the MD calculation. The Anton ASIC also contains a pair of high-bandwidth DRAM controllers (augmented with the ability to accumulate forces and other quantities), six high-speed (50.6 Gbit/s per direction) channels that provide communication to neighboring ASICs, and a host interface that communicates with an external host computer for input, output, and general control of the Anton system. The ASICs are implemented in 90-nm technology and clocked at 485 MHz, with the exception of the arithmetically intensive portion of the HTIS, which is clocked at 970 MHz.

An Anton machine may incorporate between 1 and 32,768 nodes, each of which is responsible for updating the position of particles within a distinct region of space during a simulation. One 2048-node machine, one 1024-node machine, ten 512-node machines, and several smaller machines were operational as of June 2010. For a given machine size, the nodes are connected to form a three-dimensional torus (i.e., a three-dimensional mesh that wraps around in each dimension, which maps naturally to the periodic boundary conditions used during most MD simulations). Four nodes are incorporated in each node board, and 32 node boards fit in a 19-inch rack; larger machines are constructed by linking racks together.

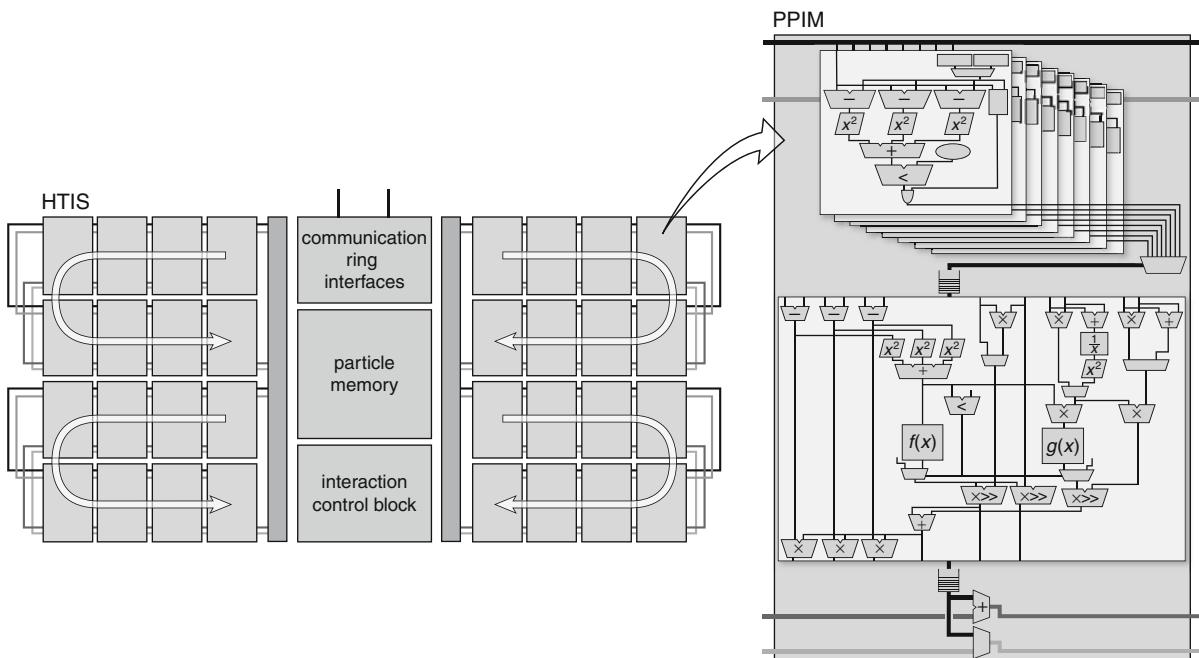
Almost all computation on Anton uses fixed-point arithmetic, which can be thought of as operating on twos-complement numbers in the range $[-1, 1]$. In practice, most of the quantities handled in an MD simulation fall within well-characterized, bounded ranges (e.g., bonds are between 1 and 3 Å in length), so there is no need for software or hardware to dynamically normalize fixed-point values. Use of fixed-point arithmetic reduces die area requirements and facilitates the achievement of certain desirable numerical properties: for example, repeated Anton simulations will produce bitwise identical results even when performed on different numbers of nodes, and molecular trajectories

produced by Anton in certain modes of operation are exactly reversible (a physical property guaranteed by Newton's laws but rarely achieved in numerical simulation) [30].

The High-Throughput Interaction Subsystem (HTIS)

The HTIS is the largest computational accelerator in Anton, handling the range-limited interactions, charge spreading, and force interpolation. These tasks account for a substantial majority of the computation involved in an MD simulation and require several hundred microseconds per time step on general-purpose supercomputers. The HTIS accelerates these computations such that they require just a few microseconds on Anton, using an array of 32 hardwired *pairwise point interaction modules* (PPIMs) (Fig. 3). The heart of each PPIM is a force calculation pipeline that computes the force between a pair of particles; this is a 26-stage pipeline (at 970 MHz) of adders, multipliers, function evaluation units, and other specialized datapath elements. The functional units of this pipeline use customized numerical precisions: bit width varies across the different stages to minimize die area while ensuring an accurate 32-bit result. The HTIS keeps these pipelines operating at high utilization through careful choreography of data flow both between chips and within a chip. A single HTIS can perform 25,600 interactions per microsecond; a modern x86 core, by contrast, can perform about 60 interactions per microsecond [20]. Despite its name, the HTIS also addresses latency: a 512-node Anton performs the entire range-limited interaction computation of a 23,000-atom MD time step in just 3 μs, over two orders of magnitude faster than any contemporaneous general-purpose computer.

The computation is parallelized across chips using a novel technique, the NT method [28], which requires less communication bandwidth than traditional methods for parallelizing range-limited interactions. Figure 4 shows the spatial volume from which particle data must be imported into each node using the NT method compared with the import volume required by the traditional “half-shell” approach. As the level of parallelism increases, the import volume of the NT method becomes progressively smaller in both absolute and asymptotic terms than that of the traditional method.



Anton, A Special-Purpose Molecular Simulation Machine. Fig. 3 High-throughput interaction subsystem (HTIS) and detail of a single pairwise point interaction module (PPIM). In addition to the 32 PPIMs, the HTIS includes two communication ring interfaces, a buffer area for particle data, and an embedded control core called the *interaction control block*. The U-shaped arrows show the flow of data through the particle distribution and force reduction networks. Each PPIM includes eight matchmaking units (shown stacked), a number of queues, and a force calculation pipeline that computes pairwise interactions

The NT method requires that each chip compute interactions between particles in one spatial region (the *tower*) and particles in another region (the *plate*). The HTIS uses a streaming approach to bring together all pairs of particles from the two sets. Figure 3 depicts the internal structure of the HTIS, which is dominated by the two halves of the PPIM array. The HTIS loads tower particles into the PPIM array and streams the plate particles through the array, past the tower particles. Each plate particle accumulates the force from its interactions with tower particles as it streams through the array. While the plate particles are streaming by, each tower particle also accumulates the force from its interactions with plate particles. After the plate particles have been processed, the accumulated tower forces are streamed out.

Not all plate particles need to interact with all tower particles; some pairs, for example, exceed the cutoff distance. To improve the utilization of the force calculation

pipelines, each PPIM includes eight dedicated *match units* that collectively check each arriving plate particle against the tower particles stored in the PPIM to determine which pairs may need to interact, using a low-precision distance test. Each particle pair that passes this test and satisfies certain other criteria proceeds to the PPIM's force calculation pipeline. As long as at least one-eighth of the pairs checked by the match units proceed, the force calculation pipeline approaches full utilization.

In addition to range-limited interactions, the HTIS performs charge spreading and force interpolation. Anton is able to map these tasks to the HTIS by employing a novel method for efficient electrostatics computation, *k*-space Gaussian Split Ewald (*k*-GSE) [27], which employs radially symmetric spreading and interpolation functions. This radial symmetry allows the hardware that computes pairwise nonbonded interactions between pairs of particles to be reused for

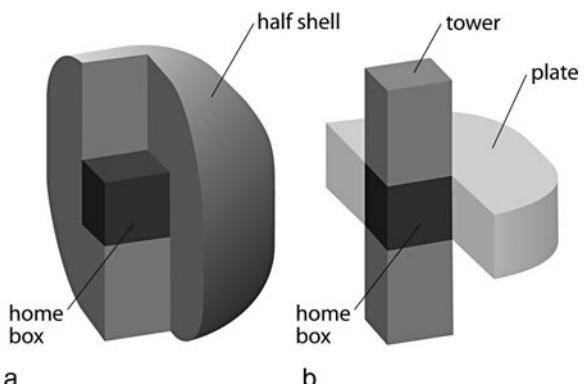
interactions between particles and grid points. Both the *k*-GSE method and the NT method were developed while re-examining fundamental MD algorithms during the design phase of Anton.

The Flexible Subsystem

Although the HTIS handles the most computationally intensive parts of an Anton calculation, the flexible subsystem performs a far wider variety of tasks. It initiates each force computation phase by sending particle positions to multiple ASICs. It handles those parts of force computation not performed in the HTIS, including calculation of bonded force terms and the FFT. It performs all integration tasks, including updating positions and velocities, constraining some particle pairs to be separated by a fixed distance, modulating temperature and pressure, and migrating atoms between nodes as the molecular system evolves. Lastly, it performs all boot, logging, and maintenance activities. The computational details of these tasks vary substantially from one MD simulation to another, making programmability a requirement.

The flexible subsystem contains eight *geometry cores* (GCs) that were designed at D. E. Shaw Research to perform fast numerical computations, four control cores (Tensilica LXs) that coordinate the overall data flow in the Anton system, and four data transfer engines that allow communication to be hidden behind computation. The GCs perform the bulk of the flexible subsystem's computational tasks, and they have been customized in a number of ways to speed up MD. Each GC is a dual-issue, statically scheduled SIMD processor with pipelined multiply accumulate support. The GC's basic data type is a vector of four 32-bit fixed-point values, and two independent SIMD operations on these vectors issue each cycle. The GC's instruction set includes element-wise vector operations (for example, vector addition), more complicated vector operations such as a dot product (which is used extensively in calculating bonded forces and applying distance constraints), and scalar operations that read and write arbitrary scalar components of the vector registers (essentially accessing the SIMD register file as a larger scalar register file).

Each of the four control cores manages a corresponding programmable data transfer engine, used to coordinate communication and synchronization for the



Anton: A Special-Purpose Molecular Simulation Machine.

Fig. 4 Import regions associated with two parallelization methods for range-limited pairwise interactions. (a) In a traditional spatial decomposition method, each node imports particles from the half-shell region so that they can interact with particles in the home box. (b) In the NT method, each node computes interactions between particles in a tower region and particles in a plate region. Both of these regions include the home box, but particles in the remainder of each region must be imported. In both methods, each pair of particles within the cutoff radius of one another will have their interaction computed on some node of the machine

flexible subsystem. In addition to the usual system interface and cache interfaces, each control core also connects to a 32-KB scratchpad memory, which holds MD simulation data for background transfer by the data transfer engine. These engines can be programmed to write data from the scratchpad to network destinations and to monitor incoming writes for synchronization purposes. The background data transfer capability provided by these engines is crucial for performance, as it enables overlapped communication and computation. The control cores also handle maintenance tasks, which tend not to be performance-critical (e.g., checkpointing every million time steps).

Considerable effort went into keeping the flexible subsystem from becoming an Amdahl's law bottleneck. Careful scheduling allows some of the tasks performed by the flexible subsystem to be partially overlapped with or completely hidden behind communication or HTIS computation [19]. Adjusting parameters for the algorithm used to evaluate electrostatic forces (including the

cutoff radius and the FFT grid density) shifts computational load from the flexible subsystem to the HTIS [30]. A number of mechanisms balance load among the cores of the flexible subsystem and across flexible subsystems on different ASICs to minimize Anton's overall execution time. Even with these and other optimizations, the flexible subsystem remains on the critical path for up to one-third of Anton's overall execution time.

Communication Subsystem

The *communication subsystem* provides high-speed, low-latency communication both between ASICs and among the subsystems within an ASIC [5]. Within a chip, two 256-bit, 485-MHz communication rings link all subsystems and the six inter-chip torus ports. Between chips, each torus link provides 50.6-Gbit/s full-duplex communication with a hop latency around 50 ns. The communication subsystem supports efficient multicast, provides flow control, and provides class-based admission control with rate metering.

In addition to achieving high bandwidth and low latency, Anton supports fine-grained inter-node communication, delivering half of peak bandwidth on messages of just 28 bytes. These properties are critical to delivering high performance in the communication-intensive tasks of an MD simulation. A 512-node Anton machine, for example, performs a $32 \times 32 \times 32$, spatially distributed 3D FFT in under 4 μs , an order of magnitude faster than the contemporary implementations in the literature [35].

Software

Although Anton's hardware architecture incorporates substantial flexibility, it was designed to perform variants of a single application: molecular dynamics. Anton's software architecture exploits this fact to maximize application performance by eliminating many of the layers of a traditional software stack. The Anton ASICs, for example, do not run a traditional operating system; instead, the control cores of the flexible subsystem run a loader that installs code and data on the machine, simulates for a time, then unloads the results of the completed simulation segment.

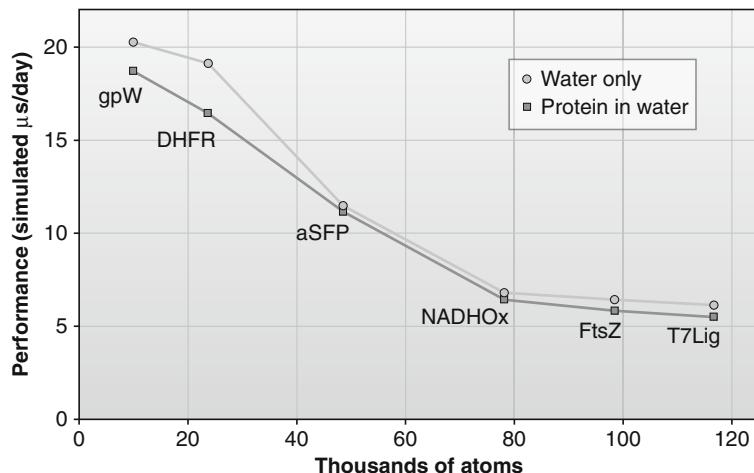
Programming Anton is complicated by the heterogeneous nature of its computational units. The geometry cores of the flexible subsystem are programmed in

assembly language, while the control cores of the flexible subsystem and a control processor in the HTIS are programmed in C augmented with intrinsics. Various fixed-function hardware units, such as the PPIMs, are programmed by configuring state machines and filling tables. Anton's design philosophy emphasized performance over ease of programmability, although increasingly sophisticated compilers and other tools to simplify programming are under development.

Anton Performance

Figure 5 shows the performance of a 512-node Anton machine on several different chemical systems, varying in size and composition. On the widely used Joint AMBER-CHARMM benchmark system, which contains 23,558 atoms and represents the protein dihydrofolate reductase (DHFR) surrounded by water, Anton simulates 16.4 μs per day of wall-clock time [30]. The fastest previously reported simulation of this system was obtained using a software package, called Desmond, which was developed within our group for use on commodity clusters [2]. This Desmond simulation executed at a rate of 471 nanoseconds (ns) per day on a 512-node 2.66-GHz Intel Xeon E5430 cluster connected by a DDR InfiniBand network, using only two of the eight cores on each node in order to maximize network bandwidth per core [3]. (Using more nodes, or more cores per node, leads to a decrease in performance as a result of an increase in communication requirements.) Due to considerations related to the efficient utilization of resources, however, neither Desmond nor other high-performance MD codes for commodity clusters are typically run at such a high level of parallelism, or in a configuration with most cores on each node idle. In practice, the performance realized in such cluster-based simulations is generally limited to speeds on the order of 100 ns/day. The previously published simulations listed in Table 1, for example, ran at 100 ns/day or less – over two orders of magnitude short of the performance we have demonstrated on Anton.

Anton machines with fewer than 512 nodes may prove more cost effective when simulating certain smaller chemical systems. A 512-node Anton machine can be partitioned, for example, into four 128-node machines, each of which achieves 7.5 $\mu\text{s}/\text{day}$ on the DHFR system – well over 25% of the 16.4 $\mu\text{s}/\text{day}$



Anton: A Special-Purpose Molecular Simulation Machine. Fig. 5 Performance of a 512-node Anton machine for chemical systems of different sizes. All simulations used 2.5-femtosecond time steps with long-range interactions evaluated at every other time step; additional simulation parameters can be found in Table 4 of reference [30]

achieved when parallelizing the same simulation across all 512 nodes. Configurations with more than 512 nodes deliver increased performance for larger chemical systems, but do not benefit chemical systems with only a few thousand atoms, for which the increase in communication latency outweighs the increase in parallelism.

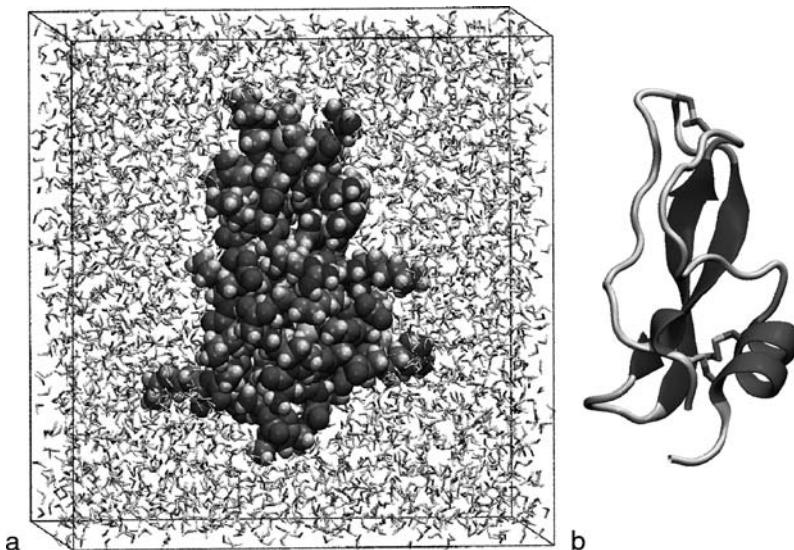
Prior to Anton's completion, few reported all-atom protein simulations had reached 2 μ s, the longest being a 10- μ s simulation that took over 3 months on the NCSA Abe supercomputer [9] (Table 1). On June 1, 2009, Anton completed the first millisecond-long simulation – more than 100 times longer than any reported previously. This 1,031- μ s simulation modeled a protein called bovine pancreatic trypsin inhibitor (BPTI) (Fig. 6), which had been the subject of many previous MD simulations; in fact, the first MD study of a protein, published in 1977 [23], simulated BPTI for 9.2 ps. The Anton simulation, which was over 100 million times longer, revealed unanticipated behavior that was not evident at previously accessible timescales, including transitions among several distinct structural states [30, 31].

Future Directions

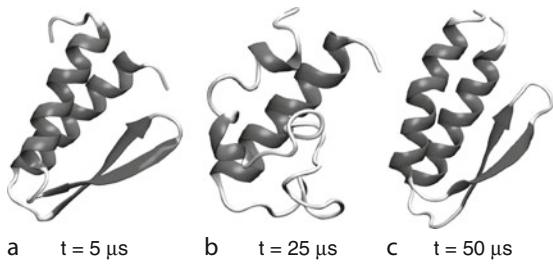
Commodity computing benefits from economies of scale but imposes limitations on the extent to which

an MD simulation, and many other high-performance computing problems, can be accelerated through parallelization. The design of Anton broke with commodity designs, embraced specialized architecture and co-designed algorithms, and achieved a three-order-of-magnitude speedup over a development period of approximately five years. Anton has thus given scientists, for the first time, the ability to perform MD simulations on the order of a millisecond – 100 times longer than any atomically detailed simulation previously reported on either general-purpose or special-purpose hardware. For computer architects, Anton's level of performance raises the questions of which other high-performance computing problems might be similarly accelerated, and whether the economic or scientific benefits of such acceleration would justify building specialized machines for those problems.

In its first two years of operation, Anton has begun to serve as a “computational microscope,” allowing the observation of biomolecular processes that have been inaccessible to laboratory experiments and that were previously well beyond the reach of computer simulation. Anton has revealed, for example, the atomic-level mechanisms by which certain proteins fold (Fig. 7) and the structural dynamics underlying the function of important drug targets [26, 31]. Anton's predictive power has been demonstrated



Anton: A Special-Purpose Molecular Simulation Machine. Fig. 6 Two renderings of a protein (BPTI) taken from a molecular dynamics simulation on Anton. (a) The entire simulated system, with each atom of the protein represented by a sphere and the surrounding water represented by thin lines. For clarity, water molecules in front of the protein are not pictured. (b) A “cartoon” rendering showing important structural elements of the protein (secondary and tertiary structure)



Anton: A Special-Purpose Molecular Simulation Machine. Fig. 7 Unfolding and folding events in a 236- μ s simulation of the protein gpW, at a temperature that equally favors the folded and unfolded states. Panel (a) shows a snapshot of a folded structure early in the simulation, (b) is a snapshot after the protein has partially unfolded, and (c) is a snapshot after it has folded again. Anton has also simulated the folding of several proteins from a completely extended state to the experimentally observed folded state

through comparison with experimental observations [25, 30]. Anton thus provides a powerful complement to laboratory experiments in the investigation of fundamental biological processes, and holds promise as a tool for the design of safe, effective, precisely targeted drugs.

Related Entries

- [Amdahl’s Law](#)
- [Distributed-Memory Multiprocessor](#)
- [GRAPE](#)
- [IBM Blue Gene Supercomputer](#)
- [NAMD \(NAnoscale Molecular Dynamics\)](#)
- [N-Body Computational Methods](#)
- [QCDSP and QCDOC Computers](#)

Bibliographic Notes and Further Reading

The first Anton machine was developed at D. E. Shaw Research between 2003 and 2008. The overall architecture was described in [29], with the HTIS, the flexible subsystem, and the communication subsystem described in more detail in [20], [19], and [5], respectively. Initial performance results were presented in [30], and initial scientific results in [31] and [26]. Other aspects of the Anton architecture, software, and design process are described in several additional papers [13, 14, 16, 35].

A number of previous projects built specialized hardware for MD simulation, including MD-GRAPE [33], MD Engine [34], and FASTRUN [7]. Extensive effort has focused on efficient parallelization of MD on general-purpose architectures, including IBM’s Blue

Gene [8] and commodity clusters [1, 2, 15]. More recently, MD has also been ported to the Cell BE processor [21] and to GPUs [32].

Bibliography

1. Bhatele A, Kumar S, Mei C, Phillips JC, Zheng G, Kalé LV (2008) Overcoming scaling challenges in biomolecular simulations across multiple platforms. In: Proceedings of the IEEE international parallel and distributed processing symposium, Miami
2. Bowers KJ, Chow E, Xu H, Dror RO, Eastwood MP, Gregersen BA, Klepeis JL, Kolossváry I, Moraes MA, Sacerdoti FD, Salmon JK, Shan Y, Shaw DE (2006) Scalable algorithms for molecular dynamics simulations on commodity clusters. In: Proceedings of the ACM/IEEE conference on supercomputing (SC06). IEEE, New York
3. Chow E, Rendleman CA, Bowers KJ, Dror RO, Hughes DH, Gullingsrud J, Sacerdoti FD, Shaw DE (2008) Desmond performance on a cluster of multicore processors. D. E. Shaw Research Technical Report DESRES/TR-2008-01, New York. <http://deshawresearch.com>
4. Dror RO, Arlow DH, Borhani DW, Jensen MØ, Piana S, Shaw DE (2009) Identification of two distinct inactive conformations of the β_2 -adrenergic receptor reconciles structural and biochemical observations. Proc Natl Acad Sci USA 106:4689–4694
5. Dror RO, Grossman JP, Mackenzie KM, Towles B, Chow E, Salmon JK, Young C, Bank JA, Batson B, Deneroff MM, Kuskin JS, Larson RH, Moraes MA, Shaw DE (2010) Exploiting 162-nanosecond end-to-end communication latency on Anton. In: Proceedings of the conference for high performance computing, networking, storage and analysis (SC10). IEEE, New York
6. Ensign DL, Kasson PM, Pande VS (2007) Heterogeneity even at the speed limit of folding: large-scale molecular dynamics study of a fast-folding variant of the villin headpiece. J Mol Biol 374:806–816
7. Fine RD, Dimmiller G, Levinthal C (1991) FASTRUN: a special purpose, hardwired computer for molecular simulation. Proteins 11:242–253
8. Fitch BG, Rayshubskiy A, Eleftheriou M, Ward TJC, Giampapa ME, Pitman MC, Pitera JW, Swope WC, Germain RS (2008) Blue Matter: scaling of N-body simulations to one atom per node. IBM J Res Dev 52:145
9. Freddolino PL, Liu F, Gruebele MH, Schulten K (2008) Ten-microsecond MD simulation of a fast-folding WW domain. Biophys J 94:L75–L77
10. Freddolino PL, Park S, Roux B, Schulten K (2009) Force field bias in protein folding simulations. Biophys J 96:3772–2780
11. Freddolino P, Schulten K (2009) Common structural transitions in explicit-solvent simulations of villin headpiece folding. Biophys J 97:2338–2347
12. Grossfield A, Pitman MC, Feller SE, Soubias O, Gawrisch K (2008) Internal hydration increases during activation of the G-protein-coupled receptor rhodopsin. J Mol Biol 381:478–486
13. Grossman JP, Salmon JK, Ho CR, Ierardi DJ, Towles B, Batson B, Spengler J, Wang SC, Mueller R, Theobald M, Young C, Gagliardo J, Deneroff MM, Dror RO, Shaw DE (2008) Hierarchical simulation-based verification of Anton, a special-purpose parallel machine. In: Proceedings of the 26th IEEE international conference on computer design (ICCD '08), Lake Tahoe
14. Grossman JP, Young C, Bank JA, Mackenzie K, Ierardi DJ, Salmon JK, Dror RO, Shaw DE (2008) Simulation and embedded software development for Anton, a parallel machine with heterogeneous multicore ASICs. In: Proceedings of the 6th IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis (CODES/ISSS '08)
15. Hess B, Kutzner C, van der Spoel D, Lindahl E (2008) GROMACS 4: algorithms for highly efficient, load-balanced, and scalable molecular simulation. J Chem Theor Comput 4:435–447
16. Ho CR, Theobald M, Batson B, Grossman JP, Wang SC, Gagliardo J, Deneroff MM, Dror RO, Shaw DE (2009) Post-silicon debug using formal verification waypoints. In: Proceedings of the design and verification conference and exhibition (DVCon '09), San Jose
17. Khalili-Araghi F, Gumbart J, Wen P-C, Sotomayor M, Tajkhorshid E, Shulten K (2009) Molecular dynamics simulations of membrane channels and transporters. Curr Opin Struct Biol 19:128–137
18. Klepeis JL, Lindorff-Larsen K, Dror RO, Shaw DE (2009) Long-timescale molecular dynamics simulations of protein structure and function. Curr Opin Struct Biol 19:120–127
19. Kuskin JS, Young C, Grossman JP, Batson B, Deneroff MM, Dror RO, Shaw DE (2008) Incorporating flexibility in Anton, a specialized machine for molecular dynamics simulation. In: Proceedings of the 14th annual international symposium on high-performance computer architecture (HPCA '08). IEEE, New York
20. Larson RH, Salmon JK, Dror RO, Deneroff MM, Young C, Grossman JP, Shan Y, Klepeis JL, Shaw DE (2008) High-throughput pairwise point interactions in Anton, a specialized machine for molecular dynamics simulation. In: Proceedings of the 14th annual international symposium on high-performance computer architecture (HPCA '08). IEEE, New York
21. Luttmann E, Ensign DL, Vishal V, Houston M, Rimon N, Øland J, Jayachandran G, Friedrichs MS, Pande VS (2009) Accelerating molecular dynamic simulation on the cell processor and PlayStation 3. J Comput Chem 30:262–274
22. Martinez-Mayorga K, Pitman MC, Grossfield A, Feller SE, Brown MF (2006) Retinal counterion switch mechanism in vision evaluated by molecular simulations. J Am Chem Soc 128:16502–16503
23. McCammon JA, Gelin BR, Karplus M (1977) Dynamics of folded proteins. Nature 267:585–590
24. Pande VS, Baker I, Chapman J, Elmer SP, Khalil S, Larson SM, Rhee YM, Shirts MR, Snow CD, Sorin EJ, Zagrovic B (2003) Atomistic protein folding simulations on the submillisecond time scale using worldwide distributed computing. Biopolymers 68:91–109
25. Piana S, Sarkar K, Lindorff-Larsen K, Guo M, Gruebele M, Shaw DE (2011) Computational design and experimental testing of the fastest-folding β -sheet protein. J Mol Biol 405:43–48
26. Rosenbaum DM, Zhang C, Lyons JA, Holl R, Aragao D, Arlow DH, Rasmussen SGF, Choi H-J, DeVree BT, Sunahara RK, Chae PS, Gellman SH, Dror RO, Shaw DE, Weis WI, Caffrey M, Gmeiner P, Kobilka BK (2011) Structure and function of an irreversible agonist- β_2 adrenoceptor complex. Nature 469:236–240

27. Shan Y, Klepeis JL, Eastwood MP, Dror RO, Shaw DE (2005) Gaussian split Ewald: a fast Ewald mesh method for molecular simulation. *J Chem Phys* 122:054101
28. Shaw DE (2005) A fast, scalable method for the parallel evaluation of distance-limited pairwise particle interactions. *J Comput Chem* 26:1318–1328
29. Shaw DE, Deneroff MM, Dror RO, Kuskin JS, Larson RH, Salmon JK, Young C, Batson B, Bowers KJ, Chao JC, Eastwood MP, Gagliardo J, Grossman JP, Ho CR, Ierardi DJ, Kolossaváry I, Klepeis JL, Layman T, McLeavey C, Moraes MA, Mueller R, Priest EC, Shan Y, Spengler J, Theobald M, Towles B, Wang SC (2007) Anton: a special-purpose machine for molecular dynamics simulation. In: Proceedings of the 34th annual international symposium on computer architecture (ISCA '07). ACM, New York
30. Shaw DE, Dror RO, Salmon JK, Grossman JP, Mackenzie KM, Bank JA, Young C, Deneroff MM, Batson B, Bowers KJ, Chow E, Eastwood MP, Ierardi DJ, Klepeis JL, Kuskin JS, Larson RH, Lindorff-Larsen K, Maragakis P, Moraes MA, Piana S, Shan Y, Towles B (2009) Millisecond-scale molecular dynamics simulations on Anton. In: Proceedings of the conference for high performance computing, networking, storage and analysis (SC09). ACM, New York
31. Shaw DE, Maragakis P, Lindorff-Larsen K, Piana S, Dror RO, Eastwood MP, Bank JA, Jumper JM, Salmon JK, Shan Y, Wriggers W (2010) Atomic-level characterization of the structural dynamics of proteins. *Science* 330:341–346
32. Stone JE, Phillips JC, Freddolino PL, Hardy DJ, Trabuco LG, Schulten K (2007) Accelerating molecular modeling applications with graphics processors. *J Comput Chem* 28:2618–2640
33. Taiji M, Narumi T, Ohno Y, Futatsugi N, Suengaga A, Takada N, Konagaya A (2003) Protein Explorer: a petaflops special-purpose computer system for molecular dynamics simulations. In: Proceedings of the ACM/IEEE conference on supercomputing (SC '03), Phoenix, AZ. ACM, New York
34. Toyoda S, Miyagawa H, Kitamura K, Amisaki T, Hashimoto E, Ikeda H, Kusumi A, Miyakawa N (1999) Development of MD Engine: high-speed accelerator with parallel processor design for molecular dynamics simulations. *J Comput Chem* 2:185–199
35. Young C, Bank JA, Dror RO, Grossman JP, Salmon JK, Shaw DE (2009) A $32 \times 32 \times 32$, spatially distributed 3D FFT in four microseconds on Anton. In: Proceedings of the conference for high performance computing, networking, storage and analysis (SC09). ACM, New York

Application-Specific Integrated Circuits

- VLSI Computation

Applications and Parallelism

- Computational Sciences

Architecture Independence

A

- Network Obliviousness

Area-Universal Networks

- Universality in VLSI Computation

Array Languages

CALVIN LIN

University of Texas at Austin, Austin, TX, USA

Definition

An array language is a programming language that supports the manipulation of entire arrays – or portions of arrays – as a basic unit of operation.

Discussion

Array languages provide two primary benefits: (1) They raise the level of abstraction, providing conciseness and programming convenience; (2) they provide a natural source of data parallelism, because the multiple elements of an array can typically be manipulated concurrently. Both benefits derive from the removal of control flow. For example, the following array statement assigns each element of the B array to its corresponding element in the A array:

```
A := B;
```

This same expression could be expressed in a scalar language using an explicit looping construct:

```
for i := 1 to n
  for j := 1 to n
    A[i][j] := B[i][j];
```

The array statement is conceptually simpler because it removes the need to iterate over individual array elements, which includes the need to name individual elements. At the same time, the array expression admits more parallelism because it does not over-specify the order in which the pairs of elements are evaluated; thus,

the elements of the B array can be assigned to the elements of the A array in any order, provided that they obey array language semantics. (Standard array language semantics dictate that the righthand side of an assignment be fully evaluated before its value is assigned to the variable on the lefthand side of the assignment.)

For the above example, the convenience of array operators appears to be minimal, but in the context of a parallel computer, the benefits to programmer productivity can be substantial, because a compiler can translate the above array statement to efficient parallel code, freeing the programmer from having to deal with many low-level details that would be necessary if writing in a lower-level language, such as MPI. In particular, the compiler can partition the work and compute loop bounds, handling the messy cases where values do not divide evenly by the number of processors. Furthermore, if the compiler can statically identify locations where communication must take place, it can allocate extra memory to cache communicated values, and it can insert communication where necessary, including any necessary marshalling of noncontiguous data. Even for shared memory machines, the burden of partitioning work, inserting appropriate synchronization, etc., can be substantial.

Array Indexing

Array languages can be characterized by the mechanisms that they provide for referring to portions of an array. The first array language, APL, provided no method of accessing portions of an array. Instead, in APL, all operators are applied to all elements of their array operands.

Languages such as Fortran 90 use array *slices* to concisely specify indices for each dimension of an array. For example, the following statement assigns the upper left 3×3 corner of array A to the upper left corner of array B.

$$B(1:3, 1:3) = A(1:3, 1:3)$$

The problem with slices is that they introduce considerable redundancy, and they force the programmer to perform index calculations, which can be error prone. For example, consider the Jacobi iteration, which computes for each array element the average of its four nearest neighbors:

$$\begin{aligned} B(2:n, 2:n) &= (A(1:n-1, 2:n) + \\ &\quad A(3:n+1, 2:n) + \\ &\quad A(2:n, 1:n-1) + \\ &\quad A(2:n, 3:n+3) + \end{aligned}$$

The problem becomes much worse, of course, for higher-dimensional arrays.

The C* language [13], which was designed for the Connection machine, simplifies array indexing by defining indices that are relative to each array element. For example, the core of the Jacobi iteration can be expressed in C* as follows, where active is an array that specifies the elements of the array where the computation should take place:

```
where (active)
{
    B = ( [.-1] [.] A + [.+1] [.] A
          + [.] [-1] A + [.] [+1] A) /4;
}
```

C*'s relative indexing is less error prone than slices: Each relative index focuses attention on the differences among the array references, so it is clear that the first two array references refer to the neighbors above and below each element and that the last two array references refer to the neighbors to the left and right of each element.

The ZPL language [4] further raises the level of abstraction by introducing the notion of a *region* to represent index sets. Regions are a first-class language construct, so regions can be named and manipulated. The ability to name regions is important because – for software maintenance and readability reasons – descriptive names are preferable to constant values. The ability to manipulate regions is important because – like C*'s relative indexing – it defines new index sets relative to existing index sets, thereby highlighting the relationship between the new index set and the old index set.

To express the Jacobi iteration in ZPL, programmers would use the At operator (@) to translate an index set by a named vector:

```
[R] B := (A@north + A@south +
           A@west + A@east) /4;
```

The above code assumes that the programmer has defined the region R to represent the index set $[1:n][1:n]$, has defined north to be a vector whose value is $[-1, 0]$, has defined south to be a vector

whose value is $[+1,0]$, and so forth. Given these definitions, the region R provides a base index set $[1:n][1:n]$ for every reference to a two-dimensional array in this statement, so R applies to every occurrence of A in this statement. The direction `north` shifts this base index set by -1 in the first dimension, etc. Thus, the above statement has the same meaning as the Fortran 90 slice notation, but it uses named values instead of hard-coded constants.

ZPL provides other region operators to support other common cases, and it uses regions in other ways, for example, to declare array variables. More significantly, regions are quite general, as they can represent sparse index sets and hierarchical index sets.

More recently, the Chapel language from Cray [5] provides an elegant form of relative indexing that can take multiple forms. For example, the below Chapel code looks almost identical to the ZPL equivalent, except that it includes the base region, R , in each array index expression:

```
T[R] = (A[R+north] + A[R+south] +
        A[R+east] + A[R+west]) / 4.0;
```

Alternatively, the expression could be written from the perspective of a single element in the index set:

```
[ij in R] T(ij) = (A(ij+north) +
                      A(ij+south) +
                      A(ij+east) +
                      A(ij+west)) / 4.0;
```

The above formulation is significant because it allows the variable `ij` to represent an element of an arbitrary tuple, where the tuple (referred to as a *domain* in Chapel) could represent many different types of index sets, including sparse index sets, hierarchical index sets, or even nodes of a graph.

Finally, the following Chapel code fragment shows that the tuple can be decomposed into its constituent parts, which allows arbitrary arithmetic to be performed on each part separately:

```
[(i,j) in R] A(ij) = (A(i-1,j) +
                        A(i+1,j) +
                        A(i,j-1) +
                        A(i,j+1)) / 4.0;
```

The FIDIL language [9] represents an early attempt to raise the level of abstraction, providing support for

irregular grids. In particular, FIDIL supports index sets, known as *domains*, which are first-class objects that can be manipulated through union, intersection, and difference operators.

Array Operators

Array languages can also be characterized by the set of array operators that they provide. All array languages support elementwise operations, which are the natural extension of scalar operators to array operands. The Array Indexing section showed examples of the elementwise `+` and `=` operators.

While element-wise operators are useful, at some point, data parallel computations need to be combined or summarized, so most array languages provide reduction operators, which combine – or reduce – multiple values of an array into a single scalar value. For example, the values of an array can be reduced to a scalar by summing them or by computing their maximum or minimum value.

Some languages provide additional power by allowing reductions to be applied to a subset of an array's dimensions. For example, each row of values in a two-dimensional array can be reduced to produce a single column of values. In general, a partial reduction accepts an n -dimensional array of values and produces an m -dimensional array of values, where $m < n$. This construct can be further generalized by allowing the programmer to specify an arbitrary associative and commutative function as the reduction operator.

Parallel prefix operators – also known as scan operators – are an extension of reductions that produce an array of partial values instead of a single scalar value. For example, the prefix sum accepts as input n values and computes all sums, $x_0 + x_1 + x_2 + \dots + x_k$ for $0 \leq k \leq n$. Other parallel prefix operations are produced by replacing the `+` operator with some other associative operator. (When applied to multi-dimensional arrays, the array indices are linearized in some well-defined manner, e.g., using Row Major Order.)

The parallel prefix operator is quite powerful because it provides a general mechanism for parallelizing computations that might seem to require sequential iteration. In particular, sequential loop iterations that accumulate information as they iterate can typically be solved using a parallel prefix.

Given the ability to index an individual array element, programmers can directly implement their own reduction and scan code, but there are several benefits of language support. First, reduction and scan are common abstractions, so good linguistic support makes these abstractions easier to read and write. Second, language support allows their implementations to be customized to the target machine. Third, compiler support introduces nontrivial opportunities for optimization: When multiple reductions or scans are performed in sequence, their communication components can be combined to reduce communication costs.

Languages such as Fortran 90 and APL also provide additional operators for flattening, re-shaping, and manipulating arrays in other powerful ways. These languages also provide operators such as matrix multiplication and matrix transpose that treat arrays as matrices. The inclusion of matrix operations blurs the distinction between array languages and matrix languages, which are described in the next section.

Matrix Languages

Array languages should not be confused with matrix languages, such as Matlab [7]. An array is a programming language construct that has many uses. By contrast, a matrix is a mathematical concept that carries additional semantic meaning. To understand this distinction, consider the following statement:

```
A = B * C;
```

In an array language, the above statement assigns to A the element-wise product of B and C. In a matrix language, the statement multiples B and C.

The most popular matrix language, Matlab, was originally designed as a convenient interactive interface to numeric libraries, such as EISPACK and LINPACK, that encourages exploration. Thus, for example, there are no variable declarations. These interactive features make Matlab difficult to parallelize, because they inhibit the compiler's ability to carefully communication the computation and communication.

Future Directions

Array language support can be extended in two dimensions. First, the restriction to flat dense arrays is too limiting for many computations, so language support for sparsely populated arrays, hierarchical arrays, and

irregular pointer-based data structures are also needed. In principle, Chapel's domain construct supports all of these extensions of array languages, but further research is required to fully support these data structures and to provide good performance. Second, it is important to integrate task parallelism with data parallelism, as is being explored in languages such as Chapel and X10.

Related Entries

- ▶ [Chapel \(Cray Inc. HPCS Language\)](#)
- ▶ [Fortran 90 and Its Successors](#)
- ▶ [HPF \(High Performance Fortran\)](#)
- ▶ [NESL](#)
- ▶ [ZPL](#)

Bibliographic Notes and Further Reading

The first array language, APL [10], was developed in the early 1960s and has often been referred to as the first write-only language because of its terse, complex nature.

Subsequent array languages that extended more conventional languages began to appear in the late 1980s. For example, extensions of imperative languages include C* [13], FIDIL [9], Dataparallel C [8], and Fortran 90 [1]. NESL [3] is a functional language that includes support for nested one dimensional arrays.

In the early 1990s, High Performance Fortran (HPF) was a data parallel language that extended Fortran 90 and Fortran 77 to provide directives about data distribution. At about the same time, the ZPL language [12] showed that a more abstract notion of an array's index set could lead to clear and concise programs.

More recently, the DARPA-funded High-Productivity Computing Systems project led to the development of Chapel [5] and X10 [6], which both integrate array languages with support for task parallelism.

Ladner and Fischer [11] presented key ideas of the parallel prefix algorithm, and Blelloch [2] elegantly demonstrated the power of the scan operator for array languages.

Bibliography

1. Adams JC, Brainerd WS, Martin JT, Smith BT, Wagener JL (1992) Fortran 90 handbook. McGraw-Hill, New York
2. Blelloch G (1996) Programming parallel algorithms. Comm ACM 39(3):85–97

3. Blelloch GE (1992) NESL: a nested data-parallel language. Technical Report CMUCS-92-103, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1992
4. Chamberlain BL (2001) The design and implementation of a region-based parallel language. PhD thesis, University of Washington, Department of Computer Science and Engineering, Seattle, WA
5. Chamberlain BL, Callahan D, Zima HP (2007) Parallel programmability and the Chapel language. *Int J High Perform Comput Appl* 21(3):291–312
6. Ebcioglu K, Saraswat V, Sarkar V (2004) X10: programming for hierarchical parallelism and non-uniform data access. In: International Workshop on Language Runtimes, OOPSLA 2004, Vancouver, BC
7. Amos Gilat (2003) MATLAB: an introduction with applications, 2nd edn. Wiley, New York
8. Hatcher PJ, Quinn MJ (1991) Data-parallel programming on MIMD computers. MIT Press, Cambridge, MA
9. Hilfinger PN, Colella P (1988) FIDIL: a language for scientific programming. Technical Report UCRL-98057, Lawrence Livermore National Laboratory, Livermore, CA, January 1988
10. Iverson K (1962) A programming language. Wiley, New York
11. Ladner RE, Fischer MJ (1980) Parallel prefix computation. *JACM* 27(4):831–838
12. Lin C, Snyder L (1993) ZPL: an array sublanguage. In: Banerjee U, Gelernter D, Nicolau A, Padua D (eds) Languages and compilers for parallel computing. Springer-Verlag, New York, pp 96–114
13. Rose JR, Steele Jr GL (1987) C*: an extended C language for data parallel programming. In: 2nd International Conference on Supercomputing, Santa Clara, CA, March 1987

array operations and intrinsic functions to deliver performance for parallel architectures, such as vector processors, multi-processors and VLIW processors.

Discussion

Introduction to Array Languages

There are several programming languages providing a rich set of array operations and intrinsic functions along with array constructs to assist data-parallel programming. They include Fortran 90, High Performance Fortran (HPF), APL and MATLAB, etc. Most of them provide programmers array intrinsic functions and array operations to manipulate data elements of multidimensional arrays concurrently without requiring iterative statements. Among these array languages, Fortran 90 is a typical example, which consists of an extensive set of array operations and intrinsic functions as shown in [Table 1](#). In the following paragraphs, several examples will be provided to bring readers basic information about array operations and intrinsic functions supported by Fortran 90. Though in the examples only Fortran 90 array operations are used for illustration, the array programming concepts and compiler techniques are applicable to common array languages.

Fortran 90 extends former Fortran language features to allow a variety of scalar operations and intrinsic functions to be applied to arrays. These array operations and intrinsic functions take array objects as inputs, perform a specific operation on array elements concurrently, and return results in scalars or arrays. The code fragment below is an array accumulation example, which involves two two-dimensional arrays and one array-add operation. In this example, all array elements in the array *S* is going to be updated by accumulating by those of array *A* in corresponding positions. The accumulation result, also a two-dimensional 4×4 array, is at last stored back to array *S*, in which each data element contains element-wise sum of array *A* and array *S*.

```
integer S(4,4), A(4,4)
```

```
S = S + A
```

Besides primitive array operations, Fortran 90 also provides programmers a set of array intrinsic functions to manipulate array objects. These intrinsic functions listed in [Table 1](#) include functions for data movement,

Array Languages, Compiler Techniques for

JENQ-KUEN LEE¹, RONG-GUEY CHANG², CHI-BANG KUAN¹

¹National Tsing-Hua University, Hsin-Chu, Taiwan

²National Chung Cheng University, Chia-Yi, Taiwan

Synonyms

Compiler optimizations for array languages

Definition

Compiler techniques for array languages generally include compiler supports, optimizations and code generation for programs expressed or annotated by all kinds of array languages. These compiler techniques mainly take advantage of data-parallelism explicit in

Array Languages, Compiler Techniques for. Table 1 Array intrinsic functions in Fortran 90

Array intrinsics	Functionality
CSHIFT	Circular-shift elements of the input array along one specified dimension
DOT_PRODUCT	Compute dot-product of two input arrays as two vectors
EOSHIFT	End-off shift elements of the input array along one specified dimension
MATMUL	Matrix multiplication
MERGE	Combine two conforming arrays under the control of an input mask
PACK	Pack an array under the control of an input mask
Reduction	Reduce an array by one specified dimension and operator
RESHAPE	Construct an array of a specified shape from elements of the input array
SPREAD	Replicate an array by adding one dimension
Section move	Perform data movement over a region of the input array
TRANSPOSE	Matrix transposition
UNPACK	Unpack an array under the control of an input mask

matrix multiplication, array reduction, compaction, etc. In the following paragraphs, two examples using array intrinsic functions are provided: the first one with array reduction and the second one with array data movement.

The first example, shown in the code fragment below, presents a way to reduce an array with a specific operator. The SUM intrinsic function, one instance of array reduction functions in Fortran 90, sums up an input array along a specified dimension. In this example, a two-dimension 4×4 array A, composed of total 16 elements, is passed to SUM with its first dimension specified as the target for reduction. After reduction by SUM, it is expected that the SUM will return a one-dimension array consisting of four elements, each of them corresponds to a sum of the array elements in the first dimension.

```
integer A(4,4), S(4)
```

```
S = SUM(A,1)
```

The second example presents a way to reorganize data elements within an input array with array intrinsic functions. The circular-shift (CSHIFT) intrinsic function in Fortran 90 performs data movement over an input array along a given dimension. Given a two-dimensional array, it can be considered to shift data elements of the input array to the left or right, up or down in a circular manner. The first argument of CSHIFT indicates the input array to be shifted while the rest two arguments specify the shift amount and the dimension along which data are shifted. In the code fragment below, one two-dimensional array, A, is going to be shifted by one-element offset along the first dimension. If the initial contents of array A are labeled as the left-hand side of Fig. 1, after the circular-shift its data contents will be moved to new positions as shown in the right-hand side of Fig. 1, where the first row of A sinks to the bottom and the rest move upward by one-element offset.

```
integer A(4,4)
```

```
A = CSHIFT(A,1,1)
```

Compiler Techniques for Array Languages

So far, readers may have experienced the concise representation of array operations and intrinsic functions. The advantages brought by array operations mainly focus on exposing data parallelism to compilers for concurrent data processing. The exposed data parallelism can be used by compilers to generate efficient code to be executed on parallel architectures, such as vector processors, multi-processors and VLIW processors. Array operations provide abundant parallelism to be exploited by compilers for delivering performance on those parallel processors. In the following paragraphs, two compiler techniques on compiling Fortran 90 array operations will be elaborated to readers. Though these techniques are originally developed for Fortran 90, they are also applicable to common array languages, such as Matlab and APL.

The first technique covers array operation synthesis for consecutive array operations, which treats array intrinsic functions as mathematical functions. In the synthesis technique, each array intrinsic function has its data access function that specifies its data access pattern and the mapping relationship between its input and

11	12	13	14
21	22	23	24
31	32	33	34
41	42	43	44

$A =$

$A' =$

21	22	23	24
31	32	33	34
41	42	43	44
11	12	13	14

Array Languages, Compiler Techniques for. Fig. 1 Array contents change after CSHIFT (shift amount = 1, dimension = 1)

output arrays. Through providing data access functions for all array operations, this technique can synthesize multiple array operations into a composite data access function, which expresses data accesses and computation to generate the target arrays from source arrays. In this way, compiling programs directly by the composite data access function can greatly improve performance by reducing redundant data movement and temporary storages required for passing immediate results.

The second compiler technique concerns compiler supports and optimizations for sparse array programs. In contrast with dense arrays, sparse arrays consist of much more zero elements than nonzero elements. Having this characteristic, they are more applicable than dense arrays in many scientific applications. For example, sparse linear systems, such as Boeing–Harwell matrix, are with popular usages. Similar to dense arrays, there are demands for support of array operations and intrinsic functions to elaborate data parallelism in sparse matrices. Later on, we will have several paragraphs illustrating how to support sparse array operations in Fortran 90, and we will also cover some optimizing techniques for sparse programs.

Synthesis for Array Operations

In the next paragraphs, an array operation synthesis technique targeting on compiling Fortran 90 array operations is going to be elaborated. This synthesis technique can be applied to programs that contain consecutive array operations. Array operations here include not only those of Fortran 90 but all array operations that can be formalized into data access functions. With this technique, compilers can generate efficient codes by removing redundant data movement and temporary storages, which are often introduced in compiling array programs.

Consecutive Array Operations

As shown in the previous examples, array operations take one or more arrays as inputs, conduct a specific operation to them, and return results in an array or a scalar value. For more advanced usage, multiple array operations can be cascaded to express compound computation over arrays. An array operation within the consecutive array operations takes inputs either from input arrays or intermediate results from others, processing data elements and passing its results to the next. In this way, they conceptually describe a particular relationship between source arrays and target arrays.

The following code fragment is an example with consecutive array operations, which involves three array operations, TRANSPOSE, RESHAPE, and CSHIFT, and three arrays, A(4, 4), B(4, 4), and C(16). The cascaded array operations at first transpose array A and reshape array C into two 4 × 4 matrices, afterwards sum the two 4 × 4 intermediate results, and finally circular-shift the results along its first dimension. With this example, readers may experience the concise representation and power of compound array operations in the way manipulating arrays without iterative statements.

```
integer A(4, 4), B(4, 4), C(16)
```

```
B = CSHIFT((TRANSPOSE(A)
+ RESHAPE(C, /4, 4/)), 1, 1)
```

To compile programs with consecutive array operations, one straightforward compilation may translate each array operation into a parallel loop and create temporary arrays to pass intermediate results used by rest array operations. Take the previous code fragment as an example, at first, compilers will separate the TRANSPOSE function from the consecutive array operations and create a temporary array T1 to keep the transposed results. Similarly, another array T2 is created for the RESHAPE function, and these two temporary arrays, T1 and T2, are summed into another temporary array, T3. At last, T3 is taken by the CSHIFT and used to produce the final results in the target array B.

```
integer A(4, 4), B(4, 4), C(16)
```

```
integer T1(4, 4), T2(4, 4), T3(4, 4)
```

```
T1 = TRANSPOSE(A)
```

```
T2 = RESHAPE(C, /4, 4/)
```

```
T3 = T1 + T2
B = CSHIFT(T3, 1, 1)
```

This straightforward scheme is inefficient as it introduces unnecessary data movement between temporary arrays and temporary storages for passing intermediate results. To compile array programs in a more efficient way, array operation synthesis can be applied to obtain a function F at compile-time such that $B = F(A, C)$, which is the synthesized data access function of the compound operations and it is functionally identical to the original sequence of array operations. The synthesized data access function specifies a direct mapping from source arrays to target arrays, and can be used by compilers to generate efficient code without introducing temporary arrays.

Data Access Functions

The concept of data access functions needs to be further elaborated here. An array operation has its own data access function that specifies element-wise mapping between its input and output arrays. As array operations in Fortran 90 have various formats, there are different looks of data access functions. In the following paragraphs, total three types of data access functions for different types of array operations will be provided as the basis of array operation synthesis.

The first type of data access functions is for array operations that contain a single source array and comply with continuous data accessing, such as TRANSPOSE and SPREAD in Fortran 90. For an array operation with n -dimensional target array T and m -dimensional source array S , which can be annotated in $T = \text{Array_Operation}(S)$, its data access function can be represented as equation (1). In equation (1), f_i in the right hand side is an index function representing an array subscript of the source array S , where the array subscripts from i_1 to i_n are index variables of the target array T . For example, the data access function for $T = \text{TRANSPOSE}(S)$ is as follows: $T[i, j] = S[j, i]$ where the index functions by definition are $f_1(i, j) = j$ and $f_2(i, j) = i$.

$$T[i_1, i_2, \dots, i_n] = S[f_1(i_1, i_2, \dots, i_n), f_2(i_1, i_2, \dots, i_n), \dots, f_m(i_1, i_2, \dots, i_n)] \quad (1)$$

The second type of data access functions is for array operations that also have a single source array but with

segmented data accessing, which means data in the arrays are accessed in parts or with strides. Due to segmented data access, the data access functions cannot be represented in a single continuous form. Instead, they have to be described by multiple data access patterns, each of which covers an disjointed array index range. To represent an array index range, a notation called, segmentation descriptors, can be used, which are boolean predicates of the form:

$$\phi(/f_i(i_1, i_2, \dots, i_n), \dots, f_m(i_1, i_2, \dots, i_n), /l_1 : u_1 : s_1, l_2 : u_2 : s_2, \dots, l_m : u_m : s_m/)$$

where f_i is an index function and l_i , u_i , and s_i are the lower bound, upper bound, and stride of the index function $f_i(i_1, i_2, \dots, i_n)$. The stride s_i can be omitted if it is equal to one, representing contiguous data access. For example, the segmented descriptor $\phi(/i, j/, /4 : 4, 1 : 4/)$ delimits the range of $(i = 4, j = 1 : 4)$.

After providing a notation for segmented index ranges, let us go back to array operations with single source and segmented data access functions, such as CSHIFT and EOSHIFT in Fortran 90. For an array operation with an n -dimensional target array T and an m -dimensional source array S , annotated in $T = \text{Array_Operation}(S)$, its data access function can be represented as follows:

$$T[i_1, i_2, \dots, i_n] = \begin{cases} S[f_1(i_1, i_2, \dots, i_n), f_2(i_1, i_2, \dots, i_n), \dots, f_m(i_1, i_2, \dots, i_n)] | \gamma_1 \\ S[g_1(i_1, i_2, \dots, i_n), g_2(i_1, i_2, \dots, i_n), \dots, g_m(i_1, i_2, \dots, i_n)] | \gamma_2 \\ \dots \end{cases} \quad (2)$$

where γ_1 and γ_2 are two separate array index ranges for index function f_i and g_i respectively. Take CSHIFT as an example, for an array operation $B = \text{CSHIFT}(A, 1, 1)$ with the input A and output B are both 4×4 arrays, its data access function can be represented as equation (3). In equation (3), the data access function of $\text{CSHIFT}(A, 1, 1)$ is divided into two parts, describing different sections of the target array B will be computed by different formulas: for the index range, $i = 4, j = 1 \sim 4$, $B[i, j]$ is assigned by the value of $A[i-3, j]$; for the index

range, $i = 1 \sim 3$, $j = 1 \sim 4$, $B[i, j]$ is assigned by the value of $A[i + 1, j]$.

$$B[i, j] = \begin{cases} A[i - 3, j] | \phi(/i, j/, /4 : 4, 1 : 4/) \\ A[i + 1, j] | \phi(/i, j/, /1 : 3, 1 : 4/) \end{cases} \quad (3)$$

The third type of data access functions is for array operations with multiple source arrays and continuous data access. For an array operation with k source arrays, annotated as $T = \text{Array_Operation}(S_1, S_2, \dots, S_k)$, its data access function can be represented as equation (4), where F is a k -nary function used to describe how the desired output to be derived by k data elements from the input arrays. Each element from source arrays is associated with an index function of the form f_i^j , where i and j indicates its input array number and dimension, respectively. Take whole-array addition $C(:, :) = A(:, :) + B(:, :)$ as an example, its data access function can be represented in $C[i, j] = F(A[i, j], B[i, j])$, where $F(x, y) = x + y$.

$$\begin{aligned} T[i_1, i_2, \dots, i_n] &= F(S_1 [f_1^1(i_1, i_2, \dots, i_n), \\ &\quad f_1^2(i_1, i_2, \dots, i_n), \dots, f_1^{d1}(i_1, i_2, \dots, i_n)], \dots, \\ &\quad S_k [f_k^1(i_1, i_2, \dots, i_n), f_k^2(i_1, i_2, \dots, i_n), \\ &\quad \dots, f_k^{dk}(i_1, i_2, \dots, i_n)]) \end{aligned} \quad (4)$$

Synthesis of Array Operations

After providing the definition of data access functions, let us begin to elaborate the mechanism of array operation synthesis. For illustration, the synthesis framework can be roughly partitioned into three major steps:

1. Build a parse tree for consecutive array operations
2. Provide a data access function for each array operation
3. Synthesize collected data access functions into a composite one

Throughout the three steps, the consecutive array operations shown in the previous example will again be used as a running example to illustrate the synthesis flow. We replicate the code fragment here for reference.

```
integer A(4,4), B(4,4), C(16)
```

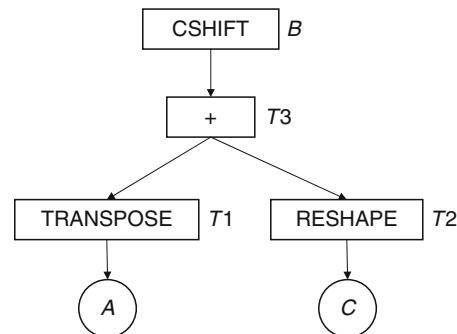
```
B = CSHIFT((TRANSPOSE(A)
+ RESHAPE(C, /4, 4/)), 1, 1)
```

In the first step, a parse tree is constructed for the consecutive array operations. In the parse tree, source arrays are leaf nodes while target arrays are roots, and each internal node corresponds to an array operation. The parse tree for the running example is provided in Fig. 2. All temporary arrays created in the straightforward compilation are arbitrarily given an unique name for identification. In the example, temporary arrays $T1$, $T2$ and $T3$ are attached to nodes of “+,” TRANSPOSE, and RESHAPE as their intermediate results. The root node is labeled with the target array, array B in this example, which will contain the final results of the consecutive array operations.

In the second step, data access functions are provided for all array operations. In the running example, the data access function of $T1 = \text{TRANSPOSE}(A)$ is $T1[i, j] = A[j, i]$; the data access function of $T2 = \text{RESHAPE}(C, /4, 4/)$ is $T2[i, j] = C[i + j * 4]$; the data access function of the $T3 = T1 + T2$ is $T3[i, j] = F_1(T1[i, j], T2[i, j])$, where $F_1(x, y) = x + y$; the data access function of $B = \text{CSHIFT}(T3)$ is as follows:

$$B[i, j] = \begin{cases} T3[i + 1, j] | \phi(/i, j/, /1 : 3, 1 : 4/) \\ T3[i - 3, j] | \phi(/i, j/, /4 : 4, 1 : 4/) \end{cases} \quad (5)$$

In the last step, the collected data access functions are going to be synthesized, starting from the data access



Array Languages, Compiler Techniques for. Fig. 2 The parse tree for consecutive array operations in the running example

function of array operation at the root node. Throughout the synthesis process, every temporary array at right-hand side (RHS) of access function is replaced by the data access function that defines the temporary array. During the substituting process, other temporary arrays will continually appear in the RHS of the updated data access function. This process repeats until all temporary arrays in the RHS are replaced with source arrays.

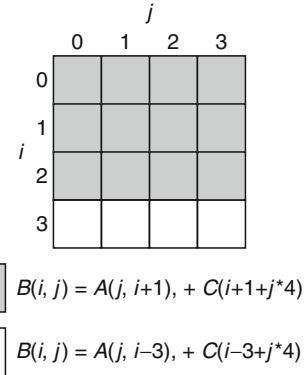
For the running example, the synthesis process begins with the data access function of CSHIFT for it is the root node in the parse tree. The data access function of CSHIFT is listed in Equation (5), in which temporary array $T3$ appears in the right-hand side. To substitute $T3$, the data access function of $T3 = T1 + T2$, which specifies $T3[i, j] = F_1(T1[i, j], T2[i, j])$ is of our interest. After the substitution, it may result in the following data access function:

$$B[i, j] = \begin{cases} F_1(T1[i+1, j], T2[i+1, j]) \mid \phi(/i, j/, /1 : 3, 1 : 4/) \\ F_1(T1[i-3, j], T2[i-3, j]) \mid \phi(/i, j/, /4 : 4, 1 : 4/) \end{cases} \quad (6)$$

In the updated data access function, come out two new temporary arrays $T1$ and $T2$. Therefore, the process continues to substitute $T1$ with $T1[i, j] = A[j, i]$ (the data access function of TRANSPOSE) and $T2$ with $T2[i, j] = C[i + j * 4]$ (the data access function of RESHAPE). At the end of synthesis process, a composite data access function will be derived as follows, containing only the target array B and the source array A and C without any temporary arrays.

$$B[i, j] = \begin{cases} F_1(A[j, i+1], C[i+1+j*4]) \mid \phi(/i, j/, /1 : 3, 1 : 4/) \\ F_1(A[j, i-3], C[i-3+j*4]) \mid \phi(/i, j/, /4 : 4, 1 : 4/) \end{cases} \quad (7)$$

The derived data access function describes a direct and element-wise mapping from data elements in the source arrays to those of the target array. The mapping described by the synthesized data access function can be visualized in Fig. 3. Since the synthesized data access function consists of two formulas with different segmented index ranges, it indicates that two portions of



Array Languages, Compiler Techniques for. Fig. 3
Constituents of the target array are described by the synthesized data access function

the target array B , gray blocks and white blocks in the Fig. 3, will be computed in different manners.

Code Generation with Data Access Functions

After deriving a composite data access function, compilers can generate a parallel loop with FORALL statements for the consecutive array operations. For the running example, compilers may generate the pseudo code below, with no redundant data movement between temporary arrays and therefore no additional storages required.

```
integer A(4,4), B(4,4), C(16)

FORALL i=1 to 4, j=1 to 4
  IF (i,j) ∈ ϕ(/i, j/, /1 : 3, 1 : 4/) THEN
    B[i, j] = A[j, i+1] + C[i+1+j*4]
  IF (i,j) ∈ ϕ(/i, j/, /4 : 4, 1 : 4/) THEN
    B[i, j] = A[j, i-3] + C[i-3+j*4]
END FORALL
```

Optimizations on Index Ranges

The code generated in the previous step consists of only one single nested loop. It is straightforward but inefficient because it includes two if-statements to ensure that different parts of the target array will be computed by different equations. These guarding statements incurred by segmented index ranges will be resolved at runtime and thus lead to performance degradation. To optimize programs with segmented index ranges the loop can be further divided into multiple loops, each of which has a

0	<0,0,0,5,3,0>	0	0	0	0
0	<2,0,1,0,0,0>	<0,0,9,3,0,0>	0	0	0
<9,0,0,0,6,0>	0	0	0	<1,0,6,2,3,5>	0
0	0	0	0	0	<0,2,0,3,4,0>
0	0	<1,0,0,0,7,2>	0	0	0
0	<2,4,2,8,0,0>	0	0	0	<0,0,0,7,3,4>

Array Languages, Compiler Techniques for. Fig. 4 A three-dimensional sparse array $A(6, 6, 6)$

bound to cover its array access range. By the index range optimization, the loop for the running example can be divided into two loops for two disjointed access ranges. At last, we conclude this section by providing efficient code as follows for the running example. It is efficient in array data accessing and promising to deliver performance. For more information about array operation synthesis, readers can refer to [2–4].

```

integer A(4,4), B(4,4), C(16)

FORALL i=1 to 3, j=1 to 4
    B[i,j]= A[j,i+1] + C[i+1+j*4]
END FORALL

FORALL i=4 to 4, j=1 to 4
    B[i,j]=A[j,i-3]+C[i-3+j*4]
END FORALL

```

Support and Optimizations for Sparse Array Operations

In the next section, another compiler technique is going to be introduced, which targets array operation supports and optimizations for sparse arrays. In contrast to dense arrays, sparse arrays consist of more zero elements than nonzero elements. With this characteristic, they are more applicable than dense arrays in many data specific applications, such as network theory and earthquake detection. Besides, they are also extensively used in scientific computation and numerical analysis. To help readers understand sparse arrays, a sparse array

$A(6, 6, 6)$ is depicted in Fig. 4. The sparse array A is a three-dimensional $6 \times 6 \times 6$ matrix with its first two dimensions containing plenty of zero elements, which are zero-vectors in the rows and columns.

Compression and Distribution Schemes for Sparse Arrays

For their sparse property, sparse arrays are usually represented in a compressed format to save computation and memory space. Besides, to process sparse workload on distributed memory environments efficiently sparse computation are often partitioned to be distributed to processors. For these reasons, both compression and distribution schemes have to be considered in supporting parallel sparse computation. Table 2 lists options of compression and distribution schemes for two-dimensional sparse arrays. There will be more detailed descriptions for these compression and distribution in upcoming paragraphs.

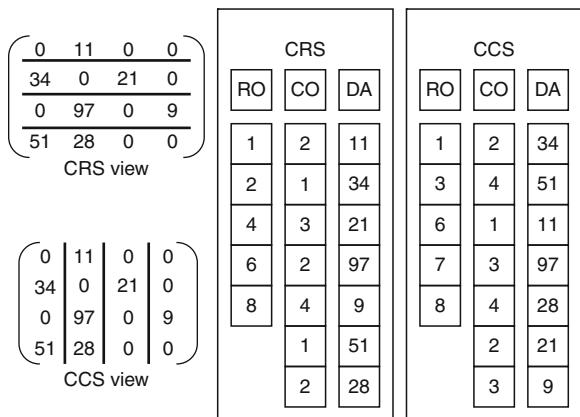
Compression Schemes

For a one-dimensional array, its compression can be either in a dense representation or in a pair-wise sparse representation. In the sparse representation, an array containing index and value pairs are used to record nonzero elements, with no space for keeping zero elements. For example, a one-dimensional array $\{0, 0, 0, 5, 3, 0\}$ can be compressed by the pair $(4, 5)$ and $(5, 3)$, representing two non-zero elements in the array, which are the fourth element equaling to five and the fifth element equaling to three.

Array Languages, Compiler Techniques for. Table 2

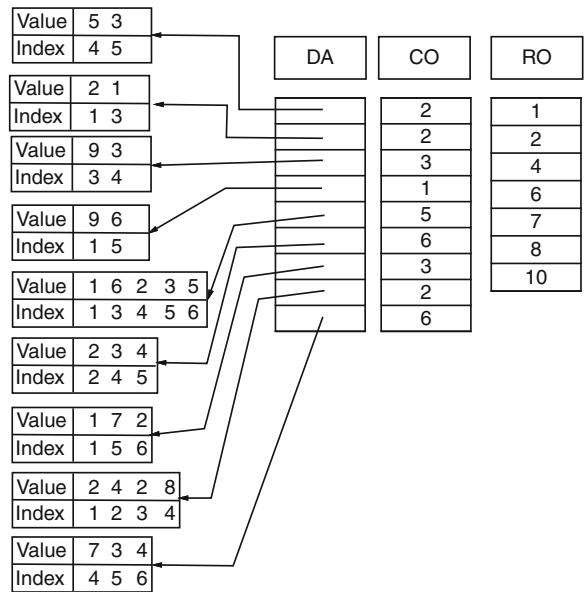
Compression and distribution schemes for two-dimensional arrays

Compression scheme	Distribution scheme
Compressed row storage (CRS)	(Block, *)
Compressed column storage (CCS)	(*, Block)
Dense representation	(Block, Block)



Array Languages, Compiler Techniques for. Fig. 5 CRS and CCS compression schemes for a two-dimensional array

For higher-dimensional arrays, there are two compression schemes for two-dimensional arrays, which are Compressed Row Storage (CRS) and Compressed Column Storage (CCS). They have different ways to compress a two-dimensional array, either by rows or by columns. The CCS scheme regards a two-dimensional array as a one-dimensional array of its rows, whereas the CRS scheme considers it as a one-dimensional array of its columns. Both of them use a CO and DA tuple, an index and data pair, to represent a nonzero element in a one-dimensional array, a row in CRS or a column in CCS. The CO fields correspond to index offsets of nonzero values in a CRS row or in a CCS column. In addition to CO and DA pairs, both CRS and CCS representation contain a RO list that keeps value indexes where each row or column starts. In Fig. 5, we show an example to encode a two-dimensional array with CRS and CCS schemes, where total 7 non-zero values are recorded with their values in the DA fields and CCS column or CRS row indexes in the CO fields.



Array Languages, Compiler Techniques for. Fig. 6 A compression scheme for the three-dimensional array in Fig. 4

Compression schemes for higher-dimensional arrays can be constructed by employing 1-d and 2-d sparse arrays as bases to construct higher-dimensional arrays. Figure 6 shows a compression scheme for the three-dimensional sparse array in Fig. 4. The three-dimensional sparse array is constructed by a two-dimensional CRS structure with each element in the first level is a one-dimensional sparse array. Similarly, one can employ 2-d sparse arrays as bases to build four-dimensional sparse arrays. The representation will be a two-dimensional compressed structure with each DA field in the structure also a two-dimensional compressed structure.

The code fragment below shows an instance to implement three-dimensional sparse arrays for both compression schemes, in which each data field contains a real number. This derived sparse2d_real data type can be used to declare sparse arrays with CRS or CCS compression schemes as the example shown in Fig. 5. All array primitive operations such as +, -, *, etc., and array intrinsic functions applied on the derived type can be overloaded to sparse implementations. In this way, data stored in sparse arrays can be manipulated as

they are in dense arrays. For example, one can conduct a matrix multiplication over two sparse arrays via a sparse matmul, in which only nonzero elements will be computed.

```
type sparse3d_real
  type(descriptor) :: d
  integer, pointer, dimension(:) :: RO
  integer, pointer, dimension(:) :: CO
  type(sparse1d_real), pointer,
    dimension(:) :: DA
end type sparse3d_real
```

Distribution Schemes

In distributed memory environments, data distribution of sparse arrays needs to be considered. The distribution schemes currently considered for sparse arrays are general block partitions based on the number of nonzero elements. For two-dimensional arrays, there are (Block, *), (*, Block) and (Block, Block) distributions. In the (*, Block) scheme a sparse array is distributed by rows, while in the (Block, *) distribution an array is distributed by columns. For the (Block, Block) distribution, an array is distributed both by rows and columns. Similarly, distribution schemes for higher-dimensional arrays can be realized by extending more dimensions in data distribution options.

The following code fragment is used to illustrate how to assign a distribution scheme to a sparse array on a distributed environment. The three-dimensional sparse array in Fig. 4 is again used for explanation. At first, the sparse array is declared as a 3-D sparse array by the derived type, sparse3d_real. Next, the *bound* function is used to specify the shape of the sparse array, specifying its size of each dimension. After shape binding, the sparse array is assigned to a (Block, Block, *) distribution scheme through the *distribution* function, by which the sparse array will be partitioned and distributed along its first two dimensions. Figure 7 shows the partition and distribution situation over a distributed environment with four-processors, where each partition covers data assigned to a target processor.

```
use sparse
type(sparse3d_real):: A

call bound(A, 6,6,6)
call distribution(A, Block, Block, *)
...
```

Programming with Sparse Arrays

With derived data types and operation overloading, sparse matrix computation can also be expressed as

		0 <0,0,0,5,3,0>	0	0	0	0
		0 <2,0,1,0,0,0>	<0,0,9,3,0,0>	0	0	0
	<9,0,0,0,6,0>	0	0	0 <1,0,6,2,3,5>	0	
		0	0	0	0	<0,2,0,3,4,0>
		0	0	<1,0,0,0,7,2>	0	0
		0	<2,4,2,8,0,0>	0	0	<0,0,0,7,3,4>

Array Languages, Compiler Techniques for. Fig. 7 A three-dimensional array is distributed on four processors by (Block, Block, *)

```

integer, parameter :: row = 1000
real, dimension (row,2*row-1) :: A
real, dimension (row) :: x, b
integer, dimension (2*row-1) :: shift

b = sum(A*eoshift (spread (x,dim=2,ncopies=2*row-1),
dim=1,shift=arth(-row+1,1,2*row-1)),dim=2)

```

Array Languages, Compiler Techniques for. Fig. 8
Numerical routines for banded matrix multiplication

concisely as dense computation. The code fragment in Fig. 8 is a part of a Fortran 90 program excerpted from the book, *Numerical Recipes in Fortran 90* [1] with its computation kernel named banded multiplication that calculates $b = Ax$, with the input, A , a matrix and the second input, x , a vector. (For more information about banded vector-matrix multiplication, readers can refer to the book [1].) In Fig. 8, both the input and output are declared as dense arrays, with three Fortran 90 array intrinsic functions, EOSHIFT, SPREAD, and SUM used to produce its results.

When the input and output are sparse matrices with plenty of zero elements, the dense representation and computation will be inefficient. With the support of sparse data types and array operations, the banmul kernel can be rewritten into a sparse version as shown in Fig. 9. Comparing Fig. 8 with Fig. 9, one can find that the sparse implementation is very similar to the dense version, with slight differences in array declaration and extra function calls for array shape binding. By changing inputs and outputs from dense to sparse arrays, a huge amount of space is saved by only recording non-zero elements. Besides, the computation kernel is unchanged thanks to operator overloading, and it tends to have better performance for only processing nonzero elements. One thing to be noted here, the compression and distribution scheme of the sparse implementation are not specified for input and output arrays, and thus the default settings are used. In advanced programming, these schemes can be configured via exposed routines to programmers, which can be used to optimize programs for compressed data organizations. We use following paragraphs to discuss how to select a proper compression and distribution scheme for sparse programs.

```

integer, parameter :: row = 1000
type(sparse2d_real) :: A
type(sparse1d_real) :: x, b
integer, dimension (2*row-1) :: shift

call bound(A, row, 2*row-1)
call bound(x, row)
call bound(b, row)

b = sum(A*eoshift (spread (x,dim=2,ncopies=2*row-1),
dim=1,shift=arth(-row+1,1,2*row-1)),dim=2)

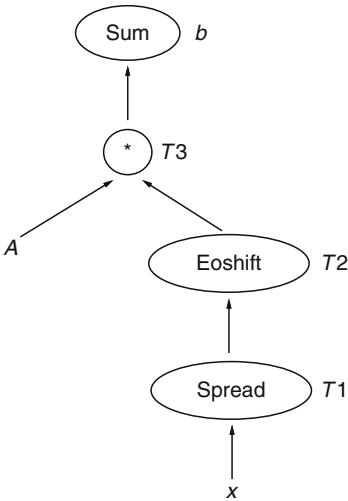
```

Array Languages, Compiler Techniques for. Fig. 9 Sparse implementation for the banmul routine

Selection of Compression and Distribution Schemes

Given a program with sparse arrays, there introduces an optimizing problem, how to select distribution and compression schemes for sparse arrays in the program. In the next paragraphs, two combinations of compression and distribution schemes for the banmul kernel are presented to show how scheme combinations make impact to performance. At first Fig. 10 shows the expression tree of the banmul kernel in Fig. 8, where T_1 , T_2 , and T_3 beside the internal nodes are temporary arrays introduced by compilers for passing intermediate results for array operations. As mentioned before, through exposed routines, programmers can choose compression and distribution schemes for input and output arrays. Nevertheless, it still has the need to figure out proper compression and distribution schemes for temporary arrays since they are introduced by compilers rather than programmers.

The reason why compression schemes for input, output and temporary arrays should be concerned is briefed as follows. When an array operation is conducted on arrays in different compression schemes, it requires data conversion that reorganizes compressed arrays from one compression scheme to another for a conforming format. That is because array operations are designed to process input arrays and return results in one conventional scheme, either in CRS or in CCS. The conversion cost implicit in array operations hurts performance a lot and needs to be avoided.

**Array Languages, Compiler Techniques for. Fig. 10**

Expression tree with intrinsic functions in banmul

Array Languages, Compiler Techniques for. Table 3

Assignments with extra compression conversions needed

Array	Compressed scheme	Distribution scheme
x	$1d$	(Block)
$T1$	CRS	(Block, *)
$T2$	CCS	(Block, *)
$T3$	CRS	(Block, *)
A	CRS	(Block, *)
b	$1d$	(Block)

In [Table 3](#) and [Table 4](#), we present two combinations of compression and distribution schemes for the banmul kernel. To focus on compression schemes, all arrays now have same distribution schemes, (Block) for one-dimensional arrays and (Block, *) for two-dimensional arrays. As shown in [Table 3](#), the input array A and three temporary arrays are in sparse representation, with only $T2$ assigned to CCS scheme and the rest assigned to CRS scheme. When the conversion cost is considered, we can tell that the scheme selection in [Table 4](#) is better than that in [Table 3](#) because it requires less conversions: one for converting the result of Spread to CCS and the other for converting the result of Eoshift to CRS.

Array Languages, Compiler Techniques for. Table 4

Assignments with less compression conversions

Array	Compressed scheme	Distribution scheme
x	$1d$	(Block)
$T1$	CCS	(Block, *)
$T2$	CCS	(Block, *)
$T3$	CCS	(Block, *)
A	CCS	(Block, *)
b	$1d$	(Block)

Sparsity, Cost Models, and the Optimal Selection

To compile sparse programs for distributed environments, in addition to conversion costs, communication costs need to be considered. In order to minimize the total execution time, an overall cost model needs be introduced, which covers three factors that have major impact to performance: computation, communication and compression conversion cost. The costs of computation and communication model time consumed on computing and transferring nonzero elements of sparse arrays, while the conversion cost mentioned before models time consumed on converting data in one compression schemes to another.

All these costs are closely related to the numbers of nonzero elements in arrays, defined as sparsity of arrays, which can be used to infer the best scheme combination for a sparse program. The sparsity informations can be provided by programmers that have knowledge about program behaviors, or they can be obtained through profiling or advanced probabilistic inference schemes discussed in [6]. With sparsity informations, the selection process can be formulated into a cost function that estimates overall execution time to run a sparse program on distributed memory environments. With this paragraph, we conclude the compiler supports and optimizations for sparse arrays. For more information about compiler techniques for sparse arrays, readers can refer to [5, 6].

Related Entries

► [Array Languages](#)

► [BLAS \(Basic Linear Algebra Subprograms\)](#)

- ▶ Data Distribution
- ▶ Dense Linear System Solvers
- ▶ Distributed-Memory Multiprocessor
- ▶ HPF (High Performance Fortran)
- ▶ Locality of Reference and Parallel Processing
- ▶ Metrics
- ▶ Reduce and Scan
- ▶ Shared-Memory Multiprocessors

Bibliographic Notes and Further Reading

For more information about compiler techniques of array operation synthesis, readers can refer to papers [2–4]. Among them, the work in [2, 3] focuses on compiling array operations supported by Fortran 90. Besides this, the synthesis framework, they also provide solutions to performance anomaly incurred by the presence of common subexpressions and one-to-many array operations. In their following work [4], the synthesis framework is extended to synthesize HPF array operations on distributed memory environments, in which communication issues are addressed.

For more information about compiler supports and optimizations for sparse programs, readers can refer to papers [5, 6]. The work in [5] puts lots of efforts to support Fortran 90 array operations for sparse arrays on parallel environments, in which both compression schemes and distribution schemes for multi-dimensional sparse arrays are considered. Besides this, the work also provides a complete complexity analysis for the sparse array implementations and report that the complexity is in proportion to the number of nonzero elements in sparse arrays, which is consistent with the conventional design criteria for sparse algorithms and data structures.

The support of sparse arrays and operations in [5] is divided into a two-level implementation: in the low-level implementation, a sparse array needs to be specified with compression and distribution schemes; in the high-level implementation, all intrinsic functions and operations are overloaded for sparse arrays, and compression and distribution details are hidden in implementations. In the work [6], a compilation scheme is proposed to transform high-level representations to low-level implementations with the three costs, computation, communication and conversion, considered.

Except the compiler techniques mentioned in this entry, there is a huge amount of literature discussing compiler techniques for dense and sparse array programs, which are not limited to Fortran 90. Among them there is a series of work on compiling ZPL [7–9], a language that defines a concise representation for describing data-parallel computations. In [9], another approach similar to array operations synthesis is proposed to achieve the same goal through loop fusion and array contraction. For other sparse program optimizations, there are also several research efforts on Matlab [10–12]. One design and implementation of sparse array support in Matlab is provided in [10], and there are related compiler techniques on Matlab sparse arrays discussed in [11, 12].

Bibliography

1. Press WH, Teukolsky SA, Vetterling WT, Flannery BP (1996) Numerical recipes in Fortran 90: the art of parallel scientific computing. Cambridge University Press, New York
2. Hwang GH, Lee JK, Ju DC (1995) An array operation synthesis scheme to optimize Fortran 90 programs. ACM SIGPLAN Notices (ACM PPoPP Issue)
3. Hwang GH, Lee JK, Ju DC (1998) A function-composition approach to synthesize Fortran 90 array operations. J Parallel Distrib Comput
4. Hwang GH, Lee JK, Ju DC (2001) Array operation synthesis to optimize HPF programs on distributed memory machines. J Parallel Distrib Comput
5. Chang RG, Chuang TR, Lee JK (2001) Parallel sparse supports for array intrinsic functions of Fortran 90. J Supercomputing 18(3):305–339
6. Chang RG, Chuang TR, Lee JK (2004) Support and optimization for parallel sparse programs with array intrinsics of Fortran 90. Parallel Comput
7. Lin C, Snyder L (1993) ZPL: an array sublanguage. 6th International Workshop on Languages and Compilers for Parallel Computing, Portland
8. Chamberlain BL, Choi S-E, Christopher Lewis E, Lin C, Snyder L, Weathersby D (1996) Factor-join: a unique approach to compiling array languages for parallel machines. 9th International Workshop on Languages and Compilers for Parallel Computing, San Jose, California
9. Christopher Lewis E, Lin C, Snyder L (1998) The implementation and evaluation of fusion and contraction in array languages. International Conference on Programming Language Design and Implementation, San Diego
10. Shah V, Gilbert JR (2004) Sparse matrices in Matlab*P: design and implementation. 11th International Conference on High Performance Computing, Springer, Heidelberg

11. Buluç A, Gilbert JR (2008) On the representation and multiplication of hypersparse matrices. 22nd IEEE International Symposium on Parallel and Distributed Processing, Miami, Florida
12. Buluç A, Gilbert JR (2008) Challenges and advances in parallel sparse matrix-matrix multiplication. International Conference on Parallel Processing, Portland

Asynchronous Iterations

► Asynchronous Iterative Algorithms

Asynchronous Iterative Algorithms

GIORGOS KOLLIAS, ANANTH Y. GRAMA, ZHIYUAN LI
Purdue University, West Lafayette, IN, USA

Synonyms

Asynchronous iterations; Asynchronous iterative computations

Definition

In iterative algorithms, a grid of data points are updated iteratively until some convergence criterion is met. The update of each data point depends on the latest updates of its neighboring points. Asynchronous iterative algorithms refer to a class of parallel iterative algorithms that are capable of relaxing strict data dependencies, hence not requiring the latest updates when they are not ready, while still ensuring convergence. Such relaxation may result in the use of inconsistent data which potentially may lead to an increased iteration count and hence increased computational operations. On the other hand, the time spent on waiting for the latest updates performed on remote processors may be reduced. Where waiting time dominates the computation, a parallel program based on an asynchronous algorithm may outperform its synchronous counterparts.

Discussion

Introduction

Scaling application performance to large number of processors must overcome challenges stemming from

high communication and synchronization costs. While these costs have improved over the years, a corresponding increase in processor speeds has tempered the impact of these improvements. Indeed, the speed gap between the arithmetic and logical operations and the memory access and message passing operations has a significant impact even on parallel programs executing on a tightly coupled shared-memory multiprocessor implemented on a single semiconductor chip.

System techniques such as prefetching and multithreading use concurrency to hide communication and synchronization costs. Application programmers can complement such techniques by reducing the number of communication and synchronization operations when they implement parallel algorithms. By analyzing the dependencies among computational tasks, the programmer can determine a minimal set of communication and synchronization points that are sufficient for maintaining all control and data dependencies embedded in the algorithm. Furthermore, there often exist several different algorithms to solve the same computational problem. Some may perform more arithmetic operations than others but require less communication and synchronization. A good understanding of the available computing system in terms of the tradeoff between arithmetic operations versus the communication and synchronization cost will help the programmer select the most appropriate algorithm.

Going beyond the implementation techniques mentioned above requires the programmer to find ways to relax the communication and synchronization requirement in specific algorithms. For example, an iterative algorithm may perform a convergence test in order to determine whether to start a new iteration. To perform such a test often requires gathering data which are scattered across different processors, incurring the communication overhead. If the programmer is familiar with the algorithm's convergence behavior, such a convergence test may be skipped until a certain number of iterations have been executed. To further reduce the communication between different processors, algorithm designers have also attempted to find ways to relax the data dependencies implied in conventional parallel algorithms such that the frequency of communication can be substantially reduced. The concept of asynchronous iterative algorithms, which dates back over 3 decades, is developed as a result of such attempts.

With the emergence of parallel systems with tens of thousands of processors and the deep memory hierarchy accessible to each processor, asynchronous iterative algorithms have recently generated a new level of interest.

Motivating Applications

Among the most computation-intensive applications currently solved using large-scale parallel platforms are iterative linear and nonlinear solvers, eigenvalue solvers, and particle simulations. Typical time-dependent simulations based on iterative solvers involve multiple levels at which asynchrony can be exploited. At the lowest level, the kernel operation in these solvers are sparse matrix-vector products and vector operations. In a graph-theoretic sense, a sparse matrix-vector product can be thought of as edge-weighted accumulations at nodes in a graph, where nodes correspond to rows and columns and edges correspond to matrix entries. Indeed, this view of a matrix-vector product forms the basis for parallelization using graph partitioners. Repeated matrix-vector products, say, to compute $A^{(n)}y$, while solving $Ax = b$, require synchronization between accumulations. However, it can be shown that within some tolerance, relaxing strict synchronization still maintains convergence guarantees of many solvers. Note that, however, the actual iteration count may be larger. Similarly, vector operations for computing residuals and intermediate norms can also relax synchronization requirements. At the next higher level, if the problem is nonlinear in nature, a quasi-Newton scheme is generally used. As before, convergence can be guaranteed even when the Jacobian solves are not exact. Finally, in time-dependent explicit schemes, some time-skew can be tolerated, provided global invariants are maintained, in typical simulations.

Particle systems exhibit similar behavior to one mentioned above. Spatial hierarchies have been well explored in this context to derive fast methods such as the Fast Multipole Method and Barnes-Hut method. Temporal hierarchies have also been explored, albeit to a lesser extent. Temporal hierarchies represent one form of relaxed synchrony, since the entire system does not evolve in lock steps. Informally, in a synchronous system, the state of a particle is determined by the state of the system in the immediately preceding time step(s) in explicit schemes. Under relaxed models of synchrony,

one can constrain the system state to be determined by the prior state of the system in a prescribed time window. So long as system invariants such as energy are maintained, this approach is valid for many applications. For example, in protein folding and molecular docking, the objective is to find minimum energy states. It is shown to be possible to converge to true minimum energy states under such relaxed models of synchrony.

While these are two representative examples from scientific domains, nonscientific algorithms lend themselves to relaxed synchrony just as well. For example, in information retrieval, PageRank algorithms can tolerate relaxed dependencies on iterative computations [35].

Iterative Algorithms

An iterative algorithm is typically organized as a series of steps essentially of the form

$$x(t+1) \leftarrow f(x(t)) \quad (1)$$

where operator $f(\cdot)$ is applied to some data $x(t)$ to produce new data $x(t+1)$. Here integer t counts the number of steps, assuming starting with $x(0)$, and captures the notion of time. Given certain properties on $f(\cdot)$, that it contracts (or pseudo-contracts) and it has a unique fixed point, and so on, an iterative algorithm is guaranteed to produce at least an approximation within a prescribed tolerance to the solution of the fixed-point equation $x = f(x)$, although the exact number of needed steps cannot be known in advance.

In the simplest computation scenario, data x reside in some Memory Entity (ME) and operator $f(\cdot)$ usually consists of both operations to be performed by some Processing Entity (PE) and parameters (i.e., data) hosted by some ME. For example, if the iteration step is a left matrix-vector multiplication $x \leftarrow Ax$, x is considered to be the “data” and the matrix itself with the set of incurred element-by-element multiplications and subsequent additions to be the “operator”, where the “operator” part also contains the matrix elements as its parameters.

However, in practice this picture can get complicated:

- Ideally x data and $f(\cdot)$ parameters should be readily available to the $f(\cdot)$ operations. This means that one

would like all data to fit in the register file of a typical PE, or even its cache files (data locality). This is not possible except for very small problems which, most of the time, are not of interest to researchers. Data typically reside either in the main memory or for larger problems in a slow secondary memory. Designing data access strategies which feed the PE at the fastest possible rate, in effect optimizing data flow across the memory hierarchy while preserving the semantics of the algorithm, is important in this aspect. Modern multicore architectures, in which many PEs share some levels of the memory hierarchy and compete for the interconnection paths, introduce new dimensions of complexity in the effective mapping of an iterative algorithm.

- There are cases in which data x are assigned to more than one PEs and $f(\cdot)$ is decomposed. This can be the result either of the sheer volume of the data or the parameters of $f(\cdot)$ or even the computational complexity of $f(\cdot)$ application which performs a large number of operations in each iteration on each data unit. In some rarer cases the data itself or the operator remains distributed by nature throughout the computation. Here, the PEs can be the cores of a single processor, the nodes (each consisting of multiple cores and processors) in a shared-memory machine or similar nodes in networked machines. The latter can be viewed as PEs accessing a network of memory hierarchies of multiple machines, i.e., an extra level of ME interconnection paths.

Synchronous Iterations and Their Problems

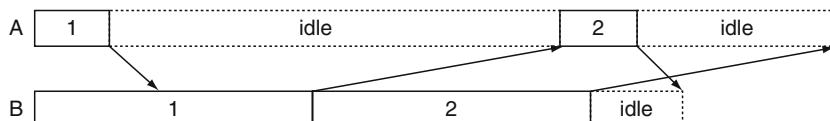
The decomposition of x and $f(\cdot)$, as mentioned above, necessitates the synchronization of all PEs involved at each step. Typically, each fragment $f_i(\cdot)$ of the decomposed operator may need many, if not all, of the

fragments $\{x_j\}$ of the newly computed data x for an iteration step. So, to preserve the exact semantics, it should wait for these data fragments to become available, in effect causing its PE to synchronize with all those PEs executing the operator fragments involved in updating these $\{x_j\}$ in need. Waiting for the availability of data at the end of each iteration make these iterations “synchronous” and this practice preserves the semantics. In networked environments, such synchronous iterations can be coupled with either “synchronous communications” (i.e., synchronous global communication at the end of each step) or “asynchronous communications” for overlapping computation and communication (i.e., asynchronously sending new data fragments as soon as they are locally produced and blocking the receipt of new data fragments at the end of each step) [5].

Synchronization between PEs is crucial to enforcing correctness of the iterative algorithm implementation but it also introduces idle synchronization phases between successive steps. For a moment consider the extreme case of a parallel iterative computation where for some PE the following two conditions happen to hold:

1. It completes its iteration step much faster than the other PEs.
2. Its input communication links from the other PEs are very slow.

It is evident that this PE will suffer a very long idle synchronization phase, i.e., time spent doing nothing (c.f. Fig. 1). Along the same line of thought, one could also devise the most unfavorable instances of computation/communication time assignments for the other PEs which suffer lengthy idle synchronization phases. It is clear that the synchronization penalty for iterative algorithms can be severe.



Asynchronous Iterative Algorithms. Fig. 1 Very long idle periods due to synchronization between A and B executing synchronous iterations. Arrows show exchange messages, the horizontal axis is time, boxes with numbers denote iteration steps, and dotted boxes denote idle time spent in the synchronization. Note that such boxes will be absent in asynchronous iterations

Asynchronous Iterations: Basic Idea and Convergence Issues

The essence of asynchronous iterative algorithms is to reduce synchronization penalty by simply eliminating the synchronization phases in iterative algorithms described above. In other words, each PE is permitted to proceed to its next iteration step without waiting for updating its data from other PEs. Use newly arriving data if available, but otherwise reuse the old data. This approach, however, fails to retain the temporal ordering of the operations in the original algorithm. The established convergence properties are usually altered as a consequence. The convergence behavior of the new asynchronous algorithm is much harder to analyze than the original [20], the main difficulty being the introduction of multiple independent time lines in practice, one for each PE. Even though the convergence analysis uses a global time line, the state space does not involve studying the behavior of the trajectory of only one point (i.e., the shared data at the end of each iteration as in synchronous iterations) but rather of a set of points, one for each PE. An extra complexity is injected by the flexibility of the local operator fragments f_i to be applied to rather arbitrary combinations of data components from the evolution history of their argument lists. Although some assumptions may underly a particular asynchronous algorithm so that certain scenarios of reusing old data components are excluded, a PE is free to use, in a later iteration, data components older than those of the current one. In other words, one can have non-FIFO communication links for local data updates between the PEs.

The most general strategy for establishing convergence of asynchronous iterations for a certain problem is to move along the lines of the Asynchronous Convergence Theorem (ACT) [13]. One tries to construct a sequence of boxes (cartesian products of intervals where data components are known to lie) with the property that the image of each box under $f(\cdot)$ will be contained in the next box in this sequence, given that the original (synchronous) iterative algorithm converges. This nested box structure ensures that as soon as all local states enter such a box, no communication scenario can make any of the states escape to an outer box (deconverge), since updating variables through communication is essentially a coordinate exchange. On the other

hand the synchronous convergence assumption guarantees that local computation also cannot produce any state escaping the current box, its only effect being a possible inclusion of some of the local data components in the respective intervals defining some smaller (nested) box. The argument here is that if communications are nothing but state transitions parallel to box facets and computations just drive some state coordinates in the facet of some smaller (nested) box, then their combination will continuously drive the set of local states to successively smaller boxes and ultimately toward convergence within a tolerance.

It follows that since a synchronous iteration is just a special case in this richer framework, its asynchronous counterpart will typically fail to converge for all asynchronous scenarios of minimal assumptions. Two broad classes of asynchronous scenarios are usually identified:

Totally asynchronous The only constraint here is that, in the local computations, data components are ultimately updated. Thus, no component ever becomes arbitrarily old as the global time line progresses, potentially indefinitely, assuming that any local computation marks a global clock tick. Under such a constraint, ACT can be used to prove, among other things, that the totally asynchronous execution of the classical iteration $x \leftarrow Ax + b$, where x and b are vectors and A a matrix distributed row-wise, converges to the correct solution provided that the spectral radius of the modulus matrix is less than unity ($\rho(|A|) < 1$). This is a very interesting result from a practical standpoint as it directly applies to classical stationary iterative algorithms (e.g., Jacobi) with nonnegative iteration matrix, those commonly used for benchmarking the asynchronous model implementations versus the synchronous ones.

Partially asynchronous The constraint here is stricter: each PE must perform at least one local iteration within the next B global clock ticks (for B a fixed integer) and it must not use components that are computed B ticks prior or older. Moreover, for the locally computed components, the most recent must always be used. Depending on how restricted the value of B may be, partially asynchronous algorithms can be classified in two types, Type I and Type II. The Type I is guaranteed to

converge for arbitrarily chosen B . An interesting case of Type I has the form $x \leftarrow Ax$ where x is a vector and A is a column-stochastic matrix. This case arises in computing stationary distributions of Markov chains [24] like the PageRank computation. Another interesting case is found in gossip-like algorithms for the distributed computation of statistical quantities, where $A = A(t)$ is a time-varying row-stochastic matrix [16]. For Type II partially asynchronous algorithms, B is restricted to be a function of the structure and the parameters of the linear operator itself to ensure convergence. Examples include gradient and gradient projection algorithms, e.g., those used in solving optimization and constrained optimization problems respectively. In these examples, unfortunately, the step parameter in the linearized operator must vary as inversely proportional to B in order to attain convergence. This implies that, although choosing large steps could accelerate the computation, the corresponding B may be too small to enforce in practice.

Implementation Issues

Termination Detection

In a synchronous iterative algorithm, global convergence can be easily decided. It can consist of a local convergence detection (e.g., to make sure a certain norm is below a local threshold in a PE) which triggers global convergence tests in all subsequent steps. The global tests can be performed by some PE acting as the monitor to check, at the end of each step, if the error (determined by substituting the current shared data x in the fixed point equation) gets below a global threshold. At that point, the monitor can signal all PEs to terminate. However, in an asynchronous setting, local convergence detection at some PE does not necessarily mean that the computation is near the global convergence, due to the fact that such a PE may compute too fast and not receive much input from its possibly slow communication in-links. Thus, in effect, there could also be the case in which each PE computes more or less in isolation from the others an approximation to a solution to the less-constrained fixed point problem concerning its respective local operator fragment $f_i(\cdot)$ but not $f(\cdot)$. Hence, local convergence detection may trigger

the global convergence tests too early. Indeed, experiments have shown long phases in which PEs continually enter and exit their “locally converged” status. With the introduction of an extra waiting period for the local convergence status to stabilize, one can avoid the premature trigger of the global convergence detection procedure which may be either centralized or distributed. Taking centralized global detection for example, there is a monitor PE which decides global convergence and notifies all iterating PEs accordingly. A practical strategy is to introduce two integer “persistence parameters” (a `localPersistence` and a `globalPersistence`). If local convergence is preserved for more than `localPersistence` iterations, the PE will notify the monitor. As soon as the monitor finds out that all the PEs remain in locally convergence status for more than `globalPersistence` of their respective checking cycles, it signals global convergence to all the PEs.

Another solution, with a simple proof of correctness, is to embed a conservative iteration number in messages, defined as the smallest iteration number of all those messages used to construct the argument list of the current local computation, incremented by one [9]. This is strongly reminiscent of the ideas in the ACT and practically the iteration number is the box counter in that context, where nested boxes are marked by successively greater numbers of such. In this way, one can precompute some bound for the target iteration number for a given tolerance, with simple local checks and minimal communication overhead. A more elaborate scheme for asynchronous computation termination detection in a distributed setting can be found in literature [1].

Asynchronous Communication

The essence of asynchronous algorithms is not to let local computations be blocked by communications with other PEs. This nonblocking feature is easier to implement on a shared memory system than on a distributed memory system. This is because on a shared memory system, computation is performed either by multiple threads or multiple processes which share the physical address space. In either case, data communication can be implemented by using shared buffers. When a PE sends out a piece of data, it locks the data buffer until the write completes. If another PE tries to receive the latest data but finds the data buffer locked, instead

of blocking itself, the PE can simply continue iterating the computation with its old, local copy of the data. On the other hand, to make sure the receive operation retrieves the data in the buffer in its entirety, the buffer must also be locked until the receive is complete. The sending PE hence may also find itself locked out of the buffer. Conceptually, one can give the send operation the privilege to preempt the receive operation, and the latter may erase the partial data just retrieved and let the send operation deposit the most up to date data. However, it may be easier to just let the send operation wait for the lock, especially because the expected waiting will be much shorter than what is typically experienced in a distributed memory system.

On distributed memory systems, the communication protocols inevitably perform certain blocking operations. For example, on some communication layer, a send operation will be considered incomplete until the receiver acknowledges the safe receipt of the data. Similarly, a receive operation on a certain layer will be considered incomplete until the arriving data become locally available. From this point of view, a communication operation in effect synchronizes the communicating partners, which is not quite compatible with the asynchronous model. For example, the so-called non-blocking send and probe operations assume the existence of a hardware device, namely, the network interface card that is independent of the PE. However, in order to avoid expensive copy operations between the application and the network layers, the send buffer is usually shared by both layers, which means that before the PE can reuse the buffer, e.g., while preparing for the next send in the application layer, it must wait until the buffer is emptied by the network layer. (Note that the application layer cannot force the network layer to undo its ongoing send operation.) This practically destroys the asynchronous semantics. To overcome this difficulty, computation and communication must be decoupled, e.g., implemented as two separate software modules, such that the blocking due to communication does not impede the computation. Mechanisms must be set up to facilitate fast data exchange between these two modules, e.g., by using another data buffer to copy data between them. Now, the computation module will act like a PE sending data in a shared memory system while the communication module acts like a PE receiving data. One

must also note that multiple probes by a receiving PE are necessary since multiple messages with the same “envelope” might have arrived during a local iteration. This again has a negative impact on the performance of an asynchronous algorithm, because the receiving PE must find the most recent of these messages and discard the rest.

The issues listed above have been discussed in the context of the MPI library [10]. Communication libraries such as Jace [7] implemented in Java or its C++ port, CRAC [18], address some of the shortcomings of “general-purpose” message passing libraries. Internally they use separate threads for the communication and computation activities with synchronized queues of buffers for the messages and automatic overwriting on the older ones.

Recent Developments and Potential Future Directions

Two-Stage Iterative Methods and Flexible Communications

There exist cases in which an algorithm can be restructured so as to introduce new opportunities for parallelization and thus yield more places to inject asynchronous semantics. Notable examples are two-stage iterative methods for solving linear systems of equations of the form $Ax = b$. Matrix A is written as $A = M - N$ with M being a block diagonal part. With such splitting, the iteration reads as $Mx(t + 1) \leftarrow (Nx(t) + b)$, and its computation can be decomposed row-wise and distributed to the PEs, i.e., a block Jacobi iteration. However, to get a new data fragment $x_i(t + 1)$ at each iteration step at some PE, a new linear system of equations must be solved, which can be done by a new splitting local M_i part, resulting in a nested iteration. In a synchronous version, new data fragments exchange only at the end of each iteration to coordinate execution. The asynchronous model applied in this context [22] not only relaxes the timing in $x_i(t + 1)$ exchanges in the synchronous model, but it also introduces asynchronous exchanges even during the nested iterations (toward $x_i(t + 1)$) and their immediate use in the computation and not at the end of the respective computational phases. This idea was initially used for solving systems of nonlinear equations [25].

Asynchronous Tiled Iterations (or Parallelizing Data Locality)

As mentioned in Sect. 3, the PEs executing the operator and the MEs hosting its parameters and data must have fast interconnection paths. Restructuring the computation so as to maximize reuse of cached data across iterations have been studied in the past [27]. These are tiling techniques [34] applied to chunks of iterations (during which convergence is not tested) and coupled with strategies for breaking the so induced dependencies. In this way data locality is considerably increased but opportunities for parallelization are confined only within the current tile data and not the whole data set as is the general case, e.g., in iterative stencil computations. Furthermore, additional synchronization barriers, scaling with the number of the tiles in number, are introduced. In a very recent work [23], the asynchronous execution of tiled chunks is proposed for regaining the parallelization degree of nontiled iterations: each PE is assigned a set of tiles (its sub-grid) and performs the corresponding loops without synchronizing with the other PEs. Only the convergence test at the end of such a phase enforces synchronization. So on the one hand, locality is preserved since each PE traverses its current tile data only and on the other hand all available PEs execute concurrently in a similar fashion without synchronizing, resulting in a large degree of parallelization.

When to Use Asynchronous Iterations?

Asynchronism can enter an iteration in both natural and artificial ways. In naturally occurring asynchronous iterations, PEs are either asynchronous by default (or it is unacceptable to synchronize) computation over sensor networks or distributed routing over data networks being such examples.

However, the asynchronous execution of a parallelized algorithm enters artificially, in the sense that most of the times it comes as a variation of the synchronous parallel port of a sequential one. Typically, there is a need to accelerate the sequential algorithm and as a first step it is parallelized, albeit in synchronous mode in order to preserve semantics. Next, in the presence of large synchronization penalties (when PEs are heterogeneous both in terms of computation and communication as in some Grid installations) or extra flexibility needs (such as asynchronous computation starts,

dynamic changes in data or topology, non-FIFO communication channels), asynchronous implementations are evaluated. Note that in all those cases of practical interest, networked PEs are implied. The interesting aspect of [23, 36] is that it broadens the applicability of the asynchronous paradigm in shared memory setups for yet another purpose, which is to preserve locality but without losing parallelism itself as a performance boosting strategy.

Related Entries

- [Memory Models](#)
- [Synchronization](#)

Bibliographic Notes and Further Reading

The asynchronous computation model, as an alternative to the synchronous one, has a life span of almost 4 decades. It started with its formal description in the pioneering work of Chazan and Miranker [17] and its first experimental investigations by Baudet [8] back in the 1970s. Perhaps the most extensive and systematic treatment of the subject is contained in a book by Bertsekas and Tsitsiklis in the 1980s [14], particularly in its closing three chapters. During the last 2 decades an extensive literature has been accumulated [4, 11, 12, 15, 19, 21, 26, 29, 30, 32, 33]. Most of these works explore theoretical extensions and variations of the asynchronous model coupled with very specific applications. However in the most recent ones, focus has shifted to more practical, implementation-level aspects [28, 31, 36], since the asynchronous model seems appropriate for the realization of highly heterogeneous, Internet-scale computations [2, 3, 6].

Bibliography

1. Bahi JM, Contassot-Vivier S, Couturier R, Vernier F (2005) A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms. *IEEE Trans Parallel Distrib Syst* 16(1):4–13
2. Bahi JM, Contassot-Vivier S, Couturier R (2002) Asynchronism for iterative algorithms in a global computing environment. In: 16th Annual International Symposium on high performance computing systems and applications (HPCS'02). IEEE, Moncton, Canada, pp 90–97

3. Bahi JM, Contassot-Vivier S, Couturier R (2003) Coupling dynamic load balancing with asynchronism in iterative algorithms on the computational Grid. In: 17th International Parallel and Distributed Processing Symposium (IPDPS'03), p 40. IEEE, Nice, France
4. Bahi JM, Contassot-Vivier S, Couturier R (2004) Performance comparison of parallel programming environments for implementing AIAC algorithms. In: 18th International Parallel and Distributed Processing Symposium (IPDPS'04). IEEE, Santa Fe, USA
5. Bahi JM, Contassot-Vivier S, Couturier R (2007) Parallel iterative algorithms: from sequential to Grid computing. Chapman & Hall/CRC, Boca Raton, FL
6. Bahi JM, Domas S, Mazouzi K (2004) Combination of Java and asynchronism for the Grid: a comparative study based on a parallel power method. In: 18th International Parallel and Distributed Processing Symposium (IPDPS '04), pp 158a, 8. IEEE, Santa Fe, USA, April 2004
7. Bahi JM, Domas S, Mazouzi K (2004). Jace: a Java environment for distributed asynchronous iterative computations. In: 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (EUROMICRO-PDP'04), pp 350–357. IEEE, Coruna, Spain
8. Baudet GM (1978) Asynchronous iterative methods for multiprocessors. JACM 25(2):226–244
9. El Baz D (1996) A method of terminating asynchronous iterative algorithms on message passing systems. Parallel Algor Appl 9:153–158
10. El Baz D (2007) Communication study and implementation analysis of parallel asynchronous iterative algorithms on message passing architectures. In: Parallel, distributed and network-based processing, 2007. PDP '07. 15th EUROMICRO International Conference, pp 77–83. Weimar, Germany
11. El Baz D, Gazen D, Jarraya M, Spiteri P, Miellou JC (1998) Flexible communication for parallel asynchronous methods with application to a nonlinear optimization problem. D'Hollander E, Joubert G et al (eds). In: Advances in Parallel Computing: Fundamentals, Application, and New Directions. North Holland, vol 12, pp 429–436
12. El Baz D, Spiteri P, Miellou JC, Gazen D (1996) Asynchronous iterative algorithms with flexible communication for nonlinear network flow problems. J Parallel Distrib Comput 38(1):1–15
13. Bertsekas DP (1983) Distributed asynchronous computation of fixed points. Math Program 27(1):107–120
14. Bertsekas DP, Tsitsiklis JN (1989) Parallel and distributed computation. Prentice-Hall, Englewood Cliffs, NJ
15. Blathras K, Szyld DB, Shi Y (1999) Timing models and local stopping criteria for asynchronous iterative algorithms. J Parallel Distrib Comput 58(3):446–465
16. Blondel VD, Hendrickx JM, Olshevsky A, Tsitsiklis JN (2006) Convergence in multiagent coordination, consensus, and flocking. In: Decision and Control, 2005 and 2005 European Control Conference. CDC-ECC'05. 44th IEEE Conference on, pp 2996–3000
17. Chazan D, Miranker WL (1969) Chaotic relaxation. J Linear Algebra Appl 2:199–222
18. Couturier R, Domas S (2007) CRAC: a Grid environment to solve scientific applications with asynchronous iterative algorithms. In: Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, p 1–8
19. Elsner L, Koltracht I, Neumann M (1990) On the convergence of asynchronous paracontractions with application to tomographic reconstruction from incomplete data. Linear Algebra Appl 130:65–82
20. Frommer A, Szyld DB (2000) On asynchronous iterations. J Comput Appl Math 123(1–2):201–216
21. Frommer A, Schwandt H, Szyld DB (1997) Asynchronous weighted additive schwarz methods. ETNA 5:48–61
22. Frommer A, Szyld DB (1994) Asynchronous two-stage iterative methods. Numer Math 69(2):141–153
23. Liu L, Li Z (2010) Improving parallelism and locality with asynchronous algorithms. In: 15th ACM SIGPLAN Symposium on principles and practice of parallel programming (PPoPP), pp 213–222, Bangalore, India
24. Lubachevsky B, Mitra D (1986) A chaotic, asynchronous algorithm for computing the fixed point of a nonnegative matrix of unit spectral radius. JACM 33(1):130–150
25. Miellou JC, El Baz D, Spiteri P (1998) A new class of asynchronous iterative algorithms with order intervals. Mathematics of Computation, 67(221):237–255
26. Moga AC, Dubois M (1996) Performance of asynchronous linear iterations with random delays. In: Proceedings of the 10th International Parallel Processing Symposium (IPPS '96), pp 625–629
27. Song Y, Li Z (1999) New tiling techniques to improve cache temporal locality. ACM SIGPLAN Notices ACM SIGPLAN Conf Program Lang Design Implement 34(5):215–228
28. Spiteri P, Chau M (2002) Parallel asynchronous Richardson method for the solution of obstacle problem. In: Proceedings of 16th Annual International Symposium on High Performance Computing Systems and Applications, Moncton, Canada, pp 133–138
29. Strikwerda JC (2002) A probabilistic analysis of asynchronous iteration. Linear Algebra Appl 349(1–3):125–154
30. Su Y, Bhaya A, Kaszkurewicz E, Kozyakin VS (1998) Further results on convergence of asynchronous linear iterations. Linear Algebra Appl 281(1–3):11–24
31. Szyld DB (2001) Perspectives on asynchronous computations for fluid flow problems. First MIT Conference on Computational Fluid and Solid Mechanics, pp 977–980
32. Szyld DB, Xu JJ (2000) Convergence of some asynchronous nonlinear multisplitting methods. Num Algor 25(1–4):347–361
33. Uresin A, Dubois M (1996) Effects of asynchronism on the convergence rate of iterative algorithms. J Parallel Distrib Comput 34(1):66–81
34. Wolfe M (1989) More iteration space tiling. In: Proceedings of the 1989 ACM/IEEE conference on Supercomputing, p 664. ACM, Reno, NV

35. Kollia G, Gallopoulos E, Szyld DB (2006) Asynchronous iterative computations with Web information retrieval structures: The PageRank case. In: Joubert GR, Nagel WE, et al (Eds). Parallel Computing: Current and Future issues of High-End computing, NIC Series. John von Neumann-Institut für Computing, Jülich, Germany, vol 33, pp 309–316
36. Kollia G, Gallopoulos E (2007) Asynchronous Computation of PageRank computation in an interactive multithreading environment. In: Frommer A, Mahoney MW, Szyld DB (eds) Web Information Retrieval and Linear Algebra Algorithms, Dagstuhl Seminar Proceedings. IBFI, Schloss Dagstuhl, Germany, ISSN: 1862–4405

Asynchronous Iterative Computations

► [Asynchronous Iterative Algorithms](#)

ATLAS (Automatically Tuned Linear Algebra Software)

R. CLINT WHALEY
University of Texas at San Antonio, San Antonio, TX, USA

Synonyms

[Numerical libraries](#)

Definition

ATLAS [20–22, 24, 25] is an ongoing research project that uses empirical tuning to optimize dense linear algebra software. The fruits of this research are embodied in an empirical tuning framework available as an open source/free software package (also referred to as “ATLAS”), which can be downloaded from the ATLAS homepage [23]. ATLAS generates optimized libraries which are also often collectively referred to as “ATLAS,” “ATLAS libraries,” or more precisely, “ATLAS-tuned libraries.” In particular, ATLAS provides a full implementation of the BLAS [6, 7, 10, 12] (Basic Linear Algebra Subprograms) API, and a subset of optimized LAPACK [1] (Linear Algebra PACKage) routines. Because dense linear algebra is rich in operand reuse, many routines can run tens or hundreds of times

faster when tuned for the hardware than when written naively. Unfortunately, highly tuned codes are usually not performance portable (i.e., a code transformation that helps performance on architecture A may reduce performance on architecture B).

The BLAS API provides basic building block linear algebra operations, and was designed to help ease the performance portability problem. The idea was to design an API that provides the basic computational needs for most dense linear algebra algorithms, so that when this API has been tuned for the hardware, all higher-level codes that rely on it for computation automatically get the associated speedup. Thus, the job of optimizing a vast library such as LAPACK can be largely handled by optimizing the much smaller code base involved in supporting the BLAS API. The BLAS are split into three “levels” based on how much cache reuse they enjoy, and thus how computationally efficient they can be made to be. In order of efficiency, the BLAS levels are: Level 3 BLAS [6], which involve matrix–matrix operations that can run near machine peak, Level 2 BLAS [7, 8] which involve matrix–vector operations and Level 1 BLAS [10, 12], which involve vector–vector operations. The Level 1 and 2 BLAS have the same order of memory references as floating point operations (FLOPS), and so will run at roughly the speed of memory for out-of-cache operation.

The BLAS were extremely successful as an API, allowing dense linear algebra to run at near-peak rates of execution on many architectures. However, with hardware changing at the frantic pace dictated by Moore’s Law, it was an almost impossible task for hand tuners to keep BLAS libraries up-to-date on even those fortunate families of machines that enjoyed their attentions. Even worse, many architectures did not have anyone willing and able to provide tuned BLAS, which left investigators with codes that literally ran orders of magnitude slower than they should, representing huge missed opportunities for research. Even on systems where a vendor provided BLAS implementations, license issues often prevented their use (e.g., SUN provided an optimized BLAS for the SPARC, but only licensed its use for their own compilers, which left researchers using other languages such as High Performance Fortran without BLAS; this was one of the original motivations to build ATLAS).

Empirical tuning arose as a response to this need for performance portability. The idea is simple enough in principle: Rather than hand-tune operations to the architecture, write a software framework that can vary the implementation being optimized (through techniques such as code generation) so that thousands of inter-related transformation combinations can be empirically evaluated on the actual machine in question. The framework uses actual timings to discover which combinations of transformations lead to high performance on this particular machine, resulting in portably efficient implementations regardless of architecture. Therefore, instead of waiting months (or even years) for a hand tuner to do the same thing, the user need only install the empirical tuning package, which will produce a highly tuned library in a matter of hours.

ATLAS Software Releases and Version Numbering

ATLAS almost always has two current software releases available at any one time. The first is the *stable release*, which is the safest version to use. The stable release has undergone extensive testing, and is known to work on many different platforms. Further, every known bug in the stable release is tracked (along with associated fixes) in the ATLAS errata file [17]. When errors affecting answer accuracy are discovered in the stable release, a message is sent to the ATLAS error list [14], which any user can sign up for. In this way, users get updates anytime the library they are using might have an error, and they can update the software with the supplied patch if the error affects them. Stable releases happen relatively rarely (say once every year or two).

The second available package is the *developer release*, which is meant to be used by ATLAS developers, contributors, and people happy to live on the bleeding edge. Developer releases typically contain a host of features and performance improvements not available in the stable release, but many of these features will have been exposed to minimal testing (a new developer release may have only been crudely tested on a single platform, whereas a new stable release will have been extensively tested on dozens of platforms). Developer releases happen relatively often (it is not uncommon to release two in the same week).

Each ATLAS release comes with a version number, which is comprised of: <major number>.<minor

number>.<update number>. The meaning of these terms is:

Major number: Major release numbers are changed only when fairly large, sweeping changes are made. Changes in the API are the most likely to cause a major release number to increment. For example, when ATLAS went from supporting only matrix multiply to all the Level 3 BLAS, the major number changed; the same happened when ATLAS went from supporting only Level 3 BLAS to all BLAS.

Minor number: Minor release numbers are changed at each official release. Even numbers represent stable releases, while odd minor numbers are reserved for developer releases.

Update number: Update numbers are essentially patches on a particular release. For instance, stable ATLAS releases only occur roughly once per year or two. As errors are discovered, they are errata-ed, so that a user can apply the fixes by hand. When enough errata are built up that it becomes impractical to apply the important ones by hand, an update release is issued. So, stable updates are typically bug fixes, or important system workarounds, while developer updates often involve substantial new code. A typical number of updates to a stable release might be something like 4. A developer release may have any number of updates.

So, 3.8.1 would be a stable release, with one group of fixes already applied. 3.9.12 would be the 12th update (13th release) of the associated developer release.

Essentials of Empirical Tuning

Any package that adapts software based on timings falls into a classification that ATLAS shares, which we call AEOS (Automated Empirical Optimization of Software). These packages can vary strongly on details, but they must have some commonalities:

1. The search must be *automated* in some way, so that an expert hand-tuner is not required.
→ ATLAS has a variety of searches for different operations, all of which can be found in the ATLAS/tune directory.
2. The decision of whether a transformation is useful or not must be *empirical*, in that an actual timing

measurement on the specific architecture in question is performed, as opposed to the traditional application of transformations using static heuristics or profile counts.

- ATLAS has a plethora of timers and testers, which can be found in ATLAS/tune and ATLAS/bin. These timers must be much more accurate and context-sensitive than typical timers, since optimization decisions are based on them. ATLAS uses the methods described in [19] to ensure high-quality timings.
3. These methods must have some way to vary/adapt the software being tuned. ATLAS currently uses *parameterized adaptation*, *multiple implementation*, and *source generation* (see Methods of Software Adaption for details).

Methods of Software Adaptation

Parameterized adaptation: The simplest method is having runtime or compile-time variables that cause different behaviors depending on input values. In linear algebra, the most important of such parameters is probably the blocking factor(s) used in blocked algorithms, which, when varied, varies the data cache utilization. Other parameterized adaptations in ATLAS include a large number of crossover points (empirically found points in some parameter space where a second algorithm becomes superior to a first). Important crossover points in ATLAS include: whether problem size is large enough to withstand a data copy, whether problem is large enough to utilize parallelism, whether a problem dimension is close enough to degenerate that a special-case algorithm should be used, etc.

Not all important tuning variables can be handled by parameterized adaptation (simple examples include instruction cache size, choice of combined or separate multiply and add instructions, length of floating point and fetch pipelines, etc.), since varying them actually requires changing the underlying source code. This then brings in the need for the second method of software adaptation, *source code adaptation*, which involves actually generating differing implementations of the same operation.

ATLAS presently uses two methods of source code adaptation, which are discussed in greater detail below.

1. **Multiple implementation:** Formalized search of multiple hand-written implementations of the kernel in question. ATLAS uses multiple implementation in the tuning of all levels of the BLAS.
2. **Source generation:** Write a program that can generate differing implementations of a given algorithm based on a series of input parameters. ATLAS presently uses source generation in tuning matrix multiply (and hence the entire Level 3 BLAS).

Multiple implementation: Perhaps the simplest approach for source code adaptation is for an empirical tuning package to supply various hand-tuned implementations, and then the search heuristic can be as simple as trying each implementation in turn until the best is found. At first glance, one might suspect that supplying these multiple implementations would make even this approach to source code adaptation much more difficult than the traditional hand-tuning of libraries. However, traditional hand-tuning is not the mere application of known techniques it may appear when examined casually. Knowing the size and properties of your level 1 cache is not sufficient to choose the best blocking factor, for instance, as this depends on a host of interlocking factors which often defy a priori understanding in the real world. Therefore, it is common in hand-tuned optimizations to utilize the known characteristics of the machine to narrow the search, but then the programmer writes various implementations and chooses the best.

For multiple implementation, this process remains the same, but the programmer adds a search and timing layer to accomplish what would otherwise be done by hand. In the simplest cases, the time to write this layer may not be much if any more than the time the implementer would have spent doing the same process in a less formal way by hand, while at the same time capturing at least some of the flexibility inherent in empirical tuning. Due to its obvious simplicity, this method is highly parallelizable, in the sense that multiple authors can meaningfully contribute without having to understand the entire package. In particular, various specialists on given architectures can provide hand-tuned routines without needing to understand other architectures, the higher level codes (e.g., timers, search heuristics, higher-level routines which utilize these basic kernels, etc.). Therefore, writing a multiple

implementation framework can allow for outside contribution of hand-tuned kernels in an open source/free software framework such as ATLAS.

Source generation: In *source generation*, a source generator (i.e., a program that writes other programs) is produced. This source generator takes as parameters the various source code adaptations to be made. As before, simple examples include loop unrolling factors, choice of combined or separate multiply and add instructions, length of floating point and fetch pipelines, and so on. Depending on the parameters, the source generator produces a routine with the requisite characteristics. The great strength of source generators is their ultimate flexibility, which can allow for far greater tunings than could be produced by all but the best hand-coders. However, generator complexity tends to go up along with flexibility, so that these programs rapidly become almost insurmountable barriers to outside contribution.

ATLAS therefore combines these two methods of source adaptation, where the GEMM kernel source generator produces strict ANSI/ISO C for maximal architectural portability. Multiple implementation is utilized to encourage outside contribution, and allows for extreme architectural specialization via assembly implementations. Parameterized adaptation is then combined with these two techniques to fully tune the library.

Both multiple implementation and code generation are specific to the kernel being tuned, and can be either platform independent (if written to use portable languages such as C) or platform dependent (as when assembly is used or generated). Empirically tuned compilers can relax the kernel-specific tuning requirement, and there has been some initial work on utilizing this third method of software adaptation [26, 27] for ATLAS, but this has not yet been incorporated into any ATLAS release.

Search and Software Adaptation for the Level 3 BLAS

ATLAS's Level 3 BLAS are implemented as GEMM-based BLAS, and so ATLAS's empirical tuning is all done for matrix multiply (see [24] and [25] for descriptions of ATLAS's GEMM-based BLAS). As of this writing, the most current stable release is 3.8.3, and the newest developer release is 3.9.16. Some features

discussed below are available only in the later developer releases; this is noted whenever true.

GEMM (GEneral rectangular Matrix Multiply) is empirically tuned using all discussed methods (parameterized adaption, multiple implementation, and source generation). Parameterization is mainly used for crossover points and cache blocking, and ATLAS uses both its methods of source code adaptation in order to optimize GEMM:

1. **Code generation:** ATLAS's main code generator produces ANSI C implementations of ATLAS's matrix multiply kernel [25]. The code generator can be found in `ATLAS/tune/blas/gemm/emit_mm.c`. `ATLAS/tune/blas/gemm/mmsearch.c` is the master GEMM search that not only exercises the options to `emit_mm.c`, but also invokes all subservient searches. With `emit_mm.c` and `mmsearch.c`, ATLAS has a general-purpose code generator that can work on any platform with an ANSI C compiler. However, most compilers are generally unsuccessful in vectorizing these types of code (and vectorization has become critically important, especially on the $\times 86$), and so from version 3.9.15 and later, ATLAS has a code generator written by Chad Zalkin that generates SSE vectorized code using gcc's implementation of the Intel SSE intrinsics. The vectorized source generator is `ATLAS/tune/blas/gemm/mmgen_sse.c`, and the search that exercises its options is `ATLAS/tune/blas/gemm/mmkssearch_sse.c`.
2. **Multiple implementation:** ATLAS also tunes GEMM using multiple implementation, and this search can be found in `ATLAS/tune/blas/gemm/ummsearch.c`, and all the hand-tuned kernel implementations which are searched can be found in `ATLAS/tune/blas/gemm/CASES/`.

Empirical Tuning in the Rest of the Package

Currently, the the Level 1 and 2 BLAS are tuned only via parameterization and multiple implementation searches. ATLAS 3.9.15 and greater has some prototype SSE generators for matrix vector multiply and rank-1 update (most important of the Level 2 BLAS routines) available in `ATLAS/tune/blas/gemv/mvgen_sse.c` & `ATLAS/tune/blas/ger/r1gen_sse.c`, respectively. These generators are currently not

debugged, and lack a search, but may be used in later releases.

ATLAS can also autotune LAPACK's blocking factor, as discussed in [18]. ATLAS currently autotunes the QR factorization for both parallel and serial implementations.

Parallelism in ATLAS

ATLAS is currently used in two main ways in parallel programming. Parallel programmers call ATLAS's serial routines directly in algorithms they themselves have parallelized. The other main avenue of parallel ATLAS use is programmers that write serial algorithms which get implicit parallelism by calling ATLAS's parallelized BLAS implementation. ATLAS currently parallelizes only the Level 3 BLAS (the Level 1 and 2 are typically bus-bound, so parallelization of these operations can sometimes lead to *slowdown* due to increased bus contention). In the current stable (3.8.3), the parallel BLAS use pthreads. However, recent research [3] forced a complete rewrite of the threading system for the developer release which results in as much as a doubling of parallel application performance. The developer release also supports Windows threads and OpenMP in addition to pthreads. Ongoing work involves improving parallelism at the LAPACK level [4], and empirically tuning parallel crossover points, which may lead to the safe parallelization of the Level 1 and 2 BLAS.

Discussion

History of the Project

The research that eventually grew into ATLAS was undertaken by R. Clint Whaley in early 1997, as a direct response to a problem from ongoing research on parallel programming and High Performance Fortran (HPF). The Innovative Computer Laboratory (ICL) of the University of Tennessee at Knoxville (UTK) had two small clusters, one consisting of SPARC chips, and the other PentiumPROs. For the PentiumPROs, no optimized BLAS were available. For the SPARC cluster, SUN provided an optimized BLAS, but licensed them so that they could not be used with non-SUN compilers, such as the HPF compiler. This led to the embarrassment of having a 32-processor parallel algorithm run slower than a serial code on the same chips, due to lack of a portably optimal BLAS.

The PHiPAC effort [2] from Berkeley comprised the first systematic attempt to harness automated empirical tuning in this area. PHiPAC did not deliver the required performance on the platforms being used, but the idea was obviously sound. Whaley began working on this idea on nights and weekends in an attempt to make the HPF results more credible. Eventually, a working prototype was produced, and was demonstrated to the director of ICL (Jack Dongarra), and it became the full-time project for Whaley, who was eventually joined on the project by Antoine Petitet. ATLAS has been under continuous development since that time, and has followed Whaley to a variety of institutions, as shown in the ATLAS timeline below. Please note that ATLAS is an open source project, so many people have substantially contributed to ATLAS that are not mentioned here. Please see `ATLAS/doc/AtlasCredits.txt` for a rough description of developer contribution to ATLAS.

Rough ATLAS Timeline Including Stable Releases

Early 1997: Development of prototype by Whaley in spare time.

Mid 1997: Whaley full time on ATLAS development at ICL/UTK.

Dec 1997: Technical report describing ATLAS published. ATLAS v 0.1 released, provides S/D GEMM only.

Sep 1998: Version 1.0 released, using SuperScalar GEMM-based BLAS [11] to provide entire real Level 3 BLAS in both precisions.

Mid 1998: Antoine Petitet joins ATLAS group at ICL/ UTK.

Feb 1999: Version 2.0 released. Automated install and configure steps, all Level 3 BLAS supported in all four types/precisions, C interface to BLAS added.

Dec 1999: Version 3.0Beta released. ATLAS provides complete BLAS support with C and F77 interfaces. GEMM generator can generate all transpose cases, saving the need to copy small matrices. Addition of LU, Cholesky, and associated LAPACK routines.

Dec 2000: Version 3.2 released. Pthreads support for parallel Level 3 BLAS. Support for user-contribution of kernels, and associated multiple implementation search.

- Mid 2001:** Antoine leaves to take job at SUN/France.
- Jan 2002:** Whaley begins PhD work in iterative compilation at FSU.
- Jun 2002:** Version 3.4 released. Level 1 BLAS optimized. Addition of LAPACK inversion and related routines. Addition of sanity check after install.
- Dec 2003:** Version 3.6 released. Numerous optimizations, but no new API coverage.
- Jul 2005:** Whaley begins as assistant professor at UTSA.
- Mar 2006:** NSF/CRI funding for ATLAS obtained.
- Dec 2007:** Version 3.8 released. Complete rewrite of configure and install, as part of overall modernization of package (brought from 2002 to 2006, where there was no financial support, and so ATLAS work only done on volunteer basis). Numerous fixes and optimizations, but API support unchanged. Addition of ATLAS install guide. Extended LAPACK timing/testing available.

Ongoing Research and Development

There are a host of areas in ATLAS that require significant research in order to improve. Since 2005, there has been continuous ATLAS R&D coming from Whaley's group at the University of Texas at San Antonio. Here, we discuss only those areas that the ATLAS team is actually currently investigating (time of writing: November 2009). Initial work on these ideas is already in the newest developer releases, and at least some of the results will be available in the next stable release (3.10.0).

Kernel Tuning

Improving ATLAS's kernel tuning is an ongoing focus. One area of interest involves exploiting empirical compilation [26, 27] for all kernels, and adding additional vectorized source generators, particularly for the Level 2 and 3 BLAS. Additionally, it is important to add support for tuning the Level 1 and 2 BLAS to different cache states, as discussed in [26]. There is also an ongoing effort to rewrite the ATLAS timing and tuning frameworks so that others can easily plug in their own searches or empirical tuning frameworks for ATLAS to automatically use.

Improvements in Parallel Performance

There is ongoing work aimed at extending [3], both to cover more OSes, to even further improve overheads, and to empirically tune a host of crossover points

between degrees of parallelism, which should improve current performance and enable opportunities to safely parallelize additional operations. Further work along the lines of [4] is being pursued in order to more effectively parallelize LAPACK. Finally, use of massively parallel GPUs is being investigated based on the impressive initial work of [13, 15, 16].

LAPACK

In addition to the parallel work mentioned in the previous section, the ATLAS group is expanding the coverage of routines to include all Householder factorization-related routines, based on the ideas presented in [9] (this will result in ATLAS providing a native implementation, with full C and Fortran interfaces, for all dense factorizations). Another ongoing investigation involves extending the LAPACK tuning discussed in [18] to handle more routines more efficiently. Finally, the ATLAS group is researching error control [5] with an eye to keeping error bounds low while using faster algorithms.

Bibliography

- Anderson E, Bai Z, Bischof C, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Ostrouchov S, Sorensen D (1999) LAPACK users' guide, 3rd edn. SIAM, Philadelphia, PA
- Bilmes J, Asanović K, Chin CW, Demmel J (1997) Optimizing matrix multiply using PHIPAC: a portable, high-performance, ANSI C coding methodology. In: Proceedings of the ACM SIGARC International Conference on SuperComputing, Vienna, Austria, July 1997
- Castaldo AM, Whaley RC (2009) Minimizing startup costs for performance-critical threading. In: Proceedings of the IEEE international parallel and distributed processing symposium, Rome, Italy, May 2009
- Castaldo AM, Whaley RC (2010) Scaling LAPACK panel operations using parallel cache assignment. In: Accepted for publication in 15th AMC SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, Bangalore, India, January 2010
- Castaldo AM, Whaley RC, Chronopoulos AT (2008) Reducing floating point error in dot product using the superblock family of algorithms. SIAM J Sci Comput 31(2):1156–1174
- Dongarra J, Du Croz J, Duff I, Hammarling S (1990) A set of level 3 basic linear algebra subprograms. ACM Trans Math Softw 16(1): 1–17
- Dongarra J, Du Croz J, Hammarling S, Hanson R (1988) Algorithm 656: an extended set of basic linear algebra subprograms: model implementation and test programs. ACM Trans Math Softw 14(1):18–32

8. Dongarra J, Du Croz J, Hammarling S, Hanson R (1988) An extended set of FORTRAN basic linear algebra subprograms. ACM Trans Math Softw 14(1):1–17
9. Elmroth E, Gustavson F (2000) Applying recursion to serial and parallel qr factorization leads to better performance. IBM J Res Dev 44(4):605–624
10. Hanson R, Krogh F, Lawson C (1973) A proposal for standard linear algebra subprograms. ACM SIGNUM Newsl 8(16):47–69
11. Kågström B, Ling P, van Loan C (1998) Gemm-based level 3 blas: high performance model implementations and performance evaluation benchmark. ACM Trans Math Softw 24(3):268–302
12. Lawson C, Hanson R, Kincaid D, Krogh F (1979) Basic linear algebra subprograms for fortran usage. ACM Trans Math Softw 5(3):308–323
13. Li Y, Dongarra J, Tomov S (2009) A note on autotuning GEMM for GPUs. Technical Report UT-CS-09-635, University of Tennessee, January 2009
14. Whaley TC et al. Atlas mailing lists. <http://math-atlas.sourceforge.net/faq.html#lists>
15. Volkov V, Demmel J (2008) Benchmarking GPUs to tune dense linear algebra. In: Supercomputing 2008. Los Alamitos, November 2008
16. Volkov V, Demmel J (2008) LU, QR and Cholesky factorizations using vector capabilities of GPUs. Technical report, University of California, Berkeley, CA, May 2008
17. Whaley RC (2009) Atlas errata file. <http://math-atlas.sourceforge.net/errata.html>
18. Whaley RC (2008) Empirically tuning lapack’s blocking factor for increased performance. In: Proceedings of the International Multiconference on Computer Science and Information Technology, Wisla, Poland, October 2008
19. Whaley RC, Castaldo AM (2008) Achieving accurate and context-sensitive timing for code optimization. Softw Practice Exp 38(15):1621–1642
20. Whaley RC, Dongarra J (1997) Automatically tuned linear algebra software. Technical Report UT-CS-97-366, University of Tennessee, TN, December 1997. <http://www.netlib.org/lapack/lawns/lawn131.ps>
21. Whaley RC, Dongarra J (1998) Automatically tuned linear algebra software. In: SuperComputing 1998: high performance networking and computing, San Antonio, TX, USA, 1998. CD-ROM proceedings. Winner, best paper in the systems category. http://www.cs.utsa.edu/~whaley/papers/atlas_sc98.ps
22. Whaley RC, Dongarra J (1999) Automatically tuned linear algebra software. In: Ninth SIAM conference on parallel processing for scientific computing, 1999. CD-ROM proceedings.
23. Whaley RC, Petitet A (2009) Atlas homepage. <http://math-atlas.sourceforge.net/>
24. Whaley RC, Petitet A (2005) Minimizing development and maintenance costs in supporting persistently optimized BLAS. Softw Practice Exp 35(2):101–121, February 2005. <http://www.cs.utsa.edu/~whaley/papers/spercw04.ps>
25. Whaley RC, Petitet A, Dongarra JJ (2001) Automated empirical optimization of software and the ATLAS project. Parallel Comput 27(1–2):3–35
26. Whaley RC, Whalley DB (2005) Tuning high performance kernels through empirical compilation. In: The 2005 international conference on parallel processing, Oslo, Norway, June 2005, pp 89–98
27. Yi Q, Whaley RC (2007) Automated transformation for performance-critical kernels. In: ACM SIGPLAN symposium on library-centric software design, Montreal, Canada, October 2007

Atomic Operations

- ▶ Synchronization
- ▶ Transactional Memories

Automated Empirical Optimization

- ▶ Autotuning

Automated Empirical Tuning

- ▶ Autotuning

Automated Performance Tuning

- ▶ Autotuning

Automated Tuning

- ▶ Autotuning

Automatically Tuned Linear Algebra Software (ATLAS)

- ▶ ATLAS (Automatically Tuned Linear Algebra Software)

Autotuning

RICHARD W. VUDUC

Georgia Institute of Technology, Atlanta,
GA, USA

Synonyms

[Automated empirical optimization](#); [Automated empirical tuning](#); [Automated performance tuning](#); [Automated tuning](#); [Software autotuning](#)

Definition

Automated performance tuning, or *autotuning*, is an automated process, guided by experiments, of selecting one from among a set of candidate program implementations to achieve some performance goal. “Performance goal” may mean, for instance, the minimization of execution time, energy delay, storage, or approximation error. An “experiment” is the execution of a benchmark and observation of its results with respect to the performance goal. A system that implements an autotuning process is referred to as an *autotuner*. An autotuner may be a stand-alone code generation system or may be part of a compiler.

Discussion

Introduction: From Manual to Automated Tuning

When tuning a code by hand, a human programmer typically engages in the following iterative process. Given an implementation, the programmer develops or modifies the program implementation, performs an experiment to measure the implementation’s performance, and then analyzes the results to decide whether the implementation has met the desired performance goal; if not, he or she repeats these steps. The use of an iterative, experiment-driven approach is typically necessary when the program and hardware performance behavior is too complex to model explicitly in another way.

Autotuning attempts to automate this process. More specifically, the modern notion of autotuning is a process consisting of the following components, any or all

of which are automated in a given autotuning approach or system:

- *Identification* of a space of candidate implementations. That is, the computation is associated with some space of possible implementations that may be defined implicitly through parameterization.
- *Generation* of these implementations. That is, an autotuner typically possesses some facility for producing (generating) the actual code that corresponds to any given point in the space of candidates.
- *Search* for the best implementation, where best is defined relative to the performance goal. This search may be guided by an empirically derived model and/or actual experiments, that is, benchmarking candidate implementations.

Bilmes et al. first described this particular notion of autotuning in the context of an autotuner for dense matrix–matrix multiplication [4].

Intellectual Genesis

The modern notion of autotuning can be traced historically to several major movements in program generation.

The first is the work by Rice on *polyalgorithms*. A polyalgorithm is a type of algorithm that, given some a particular input instance, selects from among several candidate algorithms to carry out the desired computation [14]. Rice’s later work included his attempt to formalize algorithm selection mathematically, along with an approximation theory approach to its solution, as well as applications in both numerical algorithms and operating system scheduler selection [15]. The key influence on autotuning is the notion of multiple candidates and the formalization of the selection problem as an approximation and search problem.

A second body of work is in the area of *profile-and feedback-directed compilation*. In this work, the compiler instruments the program to gather and store data about program behavior at runtime, and then uses this data to guide subsequent compilation of the same program. This area began with the introduction of detailed program performance measurement [11] and measurement tools [9]. Soon after, measurement became an integral component in the compilation process itself in several experimental compilers, including Massalin’s *superoptimizer* for

exploring candidate instruction schedules, as well as the peephole optimizers of Chang et al. [6]. Dynamic or just-in-time compilers employ similar ideas; see Smith’s survey of the state-of-the-art in this area as of 2000 [16]. The key influence on modern autotuning is the idea of automated measurement-driven transformation.

The third body of work that has exerted a strong influence on current work in autotuning is that of formalized *domain-driven code generation systems*. The first systems were developed for signal processing and featured high-level transformation/rewrite systems that manipulated symbolic formulas and translated these formulas into code [7, 10]. The key influence on modern autotuning is the notion of high-level mathematical representations of computations and the automated transformations of these representations.

Autotuning as it is largely studied today began with the simultaneous independent development of the PHiPAC autotuner for dense matrix–matrix multiplication [4], FFTW for the fast Fourier transform [8], and the OCEANS iterative compiler for embedded systems [3]. These were soon followed by additional systems for dense linear algebra (ATLAS) [18] and signal transforms (SPIRAL) [13].

Contemporary Issues and Approaches

A developer of an autotuner must, in implementing his or her autotuning system or approach, consider a host of issues for each of the three major autotuning components, that is, identification of candidate implementations, generation, and search. The issues cut across components, though for simplicity the following exposition treats them separately.

Identification of Candidates

The identification of candidates may involve characterization of the target computations and/or anticipation of the likely target hardware platforms. Key design points include what candidates are possible, who specifies the candidates, and how the candidates are represented for subsequent code generation and/or transformation.

For example, for a given computation there may be multiple candidate algorithms and data structures, for example, which linear solver, what type of graph data structure. If the computation is expressed as code, the space may be defined through some parameterization

of the candidate transformations, for example, depth of loop unrolling, cache blocking/tiling sizes. There may be numerous other implementation issues, such as where to place data or how to schedule tasks and communication.

Regarding *who* identifies these candidates, possibilities include the autotuner developer; the autotuner itself, for example, through preprogrammed and possibly extensible rewrite rules; or even the end-user programmer, for example, through a meta-language or directives.

Among target architectures, autotuning researchers have considered all of the different categories of sequential and parallel architectures. These include everything from superscalar cache-based single- and multicore systems, vector-enabled systems, and shared and distributed memory multiprocessor systems.

Code Generation

Autotuners employ a variety of techniques for producing the actual code.

One approach is to build a specialized code generator that can only produce code for a particular computation or family of computations. The generator itself might be as conceptually simple as a script that, given a few parameter settings, produces an output implementation. It could also be as sophisticated as a program that takes an abstract mathematical formula as input, using symbolic algebra and rewriting to transform the formula to an equivalent formula, and translating the formula to an implementation. In a compiler-based autotuner, the input could be code in a general-purpose language, with conventional compiler technology enabling the transformation or rewriting of that input to some implementation output. In other words, there is a large design space for the autotuner code generator component. Prominent examples exist using combinations of these techniques.

Search

Given a space of implementations, selecting the best implementation is, generically, a combinatorial search problem. Key questions are what type of search to use, when to search, and how to evaluate the candidate implementations during search.

Among types of search, the simplest is an exhaustive approach, in which the autotuner enumerates all

possible candidates and experimentally evaluates each one. To reduce the potential infeasibility of exhaustive search, numerous pruning heuristics are possible. These include random search, simulated annealing, statistical experimental design approaches, machine-learning guided search, genetic algorithms, and dynamic programming, among others.

On the question of when to search, the main categories are off-line and at runtime. An off-line search would typically occur once per architecture or architectural families, prior to use by an application. A runtime search, by contrast, occurs when the computation is invoked by the end-user application. Hybrid approaches are of course possible. For example, a series of off-line benchmarks could be incorporated into a runtime model that selects an implementation. Furthermore, tuning could occur across multiple application invocations through historical recording and analysis mechanisms, as is the case in profile and feedback-directed compilation.

The question of evaluation is one of the methodologies for carrying out the experiment. One major issue is whether this evaluation is purely experimental or guided by some predictive model (or both). The model itself may be parameterized and parameter values learned (in the statistical sense) during tuning. A second major issue is under what context evaluation occurs. That is, tuning may depend on features of the input data, so conducting an experiment in the right context could be critical to effective tuning.

Additional Pointers

The literature on work related to autotuning is large and growing. At the time of this writing, the last major comprehensive surveys of autotuning projects had appeared in the Proceedings of the IEEE special issue edited by Moura et al. [12] and the article by Vuduc et al. [17, Section 5], which include many of the methodological aspects autotuning described here.

There are numerous community-driven efforts to assemble autotuning researchers and disseminate their results. These include: the U.S. Department of Energy (DOE) sponsored workshop on autotuning, organized under the auspices of CScADS [5]; the International Workshop on Automatic Performance Tuning [2]; and the Collective Tuning (cTuning) wiki [1], to name a few.

There are numerous debates and issues within this community at present, a few examples of which follow:

- How will we measure the success of autotuners, in terms of performance, productivity, and/or other measures?
- To what extent can entire programs be autotuned, or will large successes be largely limited to relatively small library routines and small kernels extracted from programs?
- For what applications is data-dependent tuning really necessary?
- Is there a common infrastructure (tools, languages, intermediate representations) that could support autotuning broadly, across application domains?
- Where does the line between an autotuner and a “traditional” compiler lie?
- When is search necessary, rather than analytical models? Always? Never? Or only sometimes, perhaps as a “stopgap” measure when porting to new platforms? How does an autotuner know when to stop?

Related Entries

- ▶ [Algorithm Engineering](#)
- ▶ [ATLAS \(Automatically Tuned Linear Algebra Software\)](#)
- ▶ [Benchmarks](#)
- ▶ [Code Generation](#)
- ▶ [FFTW](#)
- ▶ [Profiling](#)
- ▶ [Spiral](#)

Bibliography

1. (2009). Collective Tuning Wiki. <http://ctuning.org>
2. (2010). International Workshop on Automatic Performance Tuning. <http://iwapt.org>
3. Aarts B, Barreteau M, Bodin F, Brinkhaus P, Chamski Z, Charles H-P, Eisenbeis C, Gurd J, Hoogerbrugge J, Hu P, Jalby W, Krijnenburg PMW, O’Boyle MFP, Rohou E, Sakellariou R, Schepers H, Seznec A, Stöhr E, Verhoeven M, Wijshoff HAG (1997) OCEANS: Optimizing Compilers for Embedded Applications. In: *Proc. Euro-Par*, vol 1300 of LNCS, Passau, Germany. Springer, Berlin / Heidelberg
4. Bilmes J, Asanovic K, Chin C-W, Demmel J (1997) Optimizing matrix multiply using PHIPAC: A portable, highperformance, ANSI C coding methodology. In Proc. ACM Int’l Conf Supercomputing (ICS), Vienna, Austria, pp 340–347

5. Center for Scalable Application Development Software (2009) Workshop on Libraries and Autotuning for Petascale Applications. <http://cscads.rice.edu/workshops/summer09/autotuning>
6. Chang PP, Mahlke SA, Mei W, Hwu W (1991) Using profile information to assist classic code optimizations. *Software: Pract Exp* 21(12):1301–1321
7. Covell MM, Myers CS, Oppenheim AV (1992) Symbolic and knowledge-based signal processing, chapter 2: computer-aided algorithm design and rearrangement. *Signal Processing Series*. Prentice-Hall, pp 30–87
8. Frigo M, Johnson SG (1999) A fast Fourier transform compiler. *ACM SIGPLAN Notices* 34(5):169–180. Origin: Proc. ACM Conf. Programming Language Design and Implementation (PLDI)
9. Graham SL, Kessler PB, McKusick MK (1982) gprof: A call graph execution profiler. *ACM SIGPLAN Notices* 17(6):120–126. Origin: Proc. ACM Conf. Programming Language Design and Implementation (PLDI)
10. Johnson JR, Johnson RW, Rodriguez D, Tolimieri R (1990) A methodology for designing, modifying, and implementing Fourier Transform algorithms on various architectures. *Circuits, Syst Signal Process* 9(4):449–500
11. Knuth DE (1971) An empirical study of FORTRAN programs. *Software: Practice Exp* 1(2):105–133
12. Moura JMF, Püschel M, Dongarra J, Padua D, (eds) (2005) Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Platform Adaptation, vol 93. IEEE Comp Soc. <http://ieeexplore.ieee.org/xpl/tocresult.jsp?isNumber=30187&puNumber=5>
13. Püschel M, Moura JMF, Johnson J, Padua D, Veloso M, Singer B, Xiong J, Franchetti F, Gacic A, Voronenko Y, Chen K, Johnson RW, Rizzolo N (2005) SPIRAL: Code generation for DSP transforms. *Proc. IEEE: Special issue on “Program Generation, Optimization, and Platform Adaptation”* 93(2):232–275
14. Rice JR (1969) A polyalgorithm for the automatic solution of nonlinear equations. In: Proc. ACM Annual Conf./Annual Mtg. New York, pp 179–83
15. Rice JR (1976) The algorithm selection problem. In: Alt F, Rubinoff M, Yovits MC (eds) *Adv Comp* 15:65–117
16. Smith MD (2000) Overcoming the challenges to feedback-directed optimization. *ACM SIGPLAN Notices* 35(7):1–11
17. Vuduc R, Demmel J, Bilmes J (2004) Statistical models for empirical search-based performance tuning. *Int'l J High Performance Comp Appl (IJHPCA)* 18(1):65–94
18. Whaley RC, Petitet A, Dongarra J (2001) Automated empirical optimizations of software and the ATLAS project. *Parallel Comp (ParCo)* 27(1–2):3–35

B

Backpressure

► [Flow Control](#)

Bandwidth-Latency Models (BSP, LogP)

THILO KIELMANN¹, SERGEI GORLATCH²

¹Vrije Universiteit, Amsterdam, The Netherlands

²Westfälische Wilhelms-Universität Münster, Münster, Germany

Synonyms

Message-passing performance models; Parallel communication models

Definition

Bandwidth-latency models are a group of performance models for parallel programs that focus on modeling the communication between the processes in terms of network bandwidth and latency, allowing quite precise performance estimations. While originally developed for distributed-memory architectures, these models also apply to machines with nonuniform memory access (NUMA), like the modern multi-core architectures.

Discussion

Introduction

The foremost goal of parallel programming is to speed up algorithms that would be too slow when executed sequentially. Achieving this so-called speedup requires a deep understanding of the performance of the inter-process communication and synchronization, together with the algorithm's computation. Both computation and communication/synchronization performance strongly depend on properties of the machine

architecture in use. The strength of the bandwidth-latency models is that they model quite precisely the communication and synchronization operations. When the parameters of a given machine are fed into the performance analysis of a parallel algorithm, bandwidth-latency models lead to rather precise and expressive performance evaluations. The most important models of this class are bulk synchronous parallel (BSP) and LogP, as discussed below.

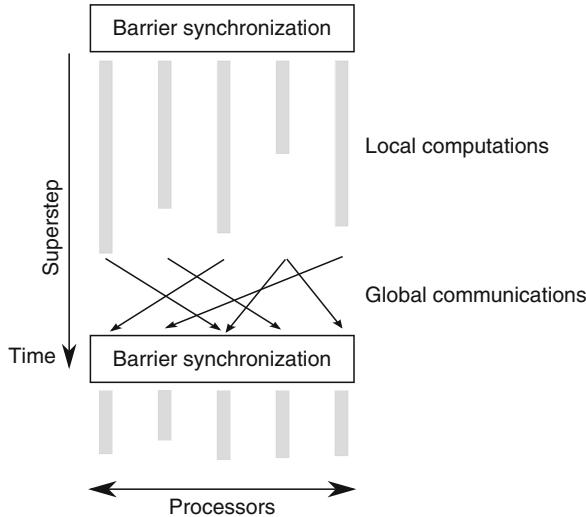
The BSP Model

The *bulk synchronous parallel* (BSP) model was proposed by Valiant [11] to overcome the shortcomings of the traditional PRAM (Parallel Random Access Machine) model, while keeping its simplicity. None of the suggested PRAM models offers a satisfying forecast of the behavior of parallel machines for a wide range of applications. The BSP model was developed as a bridge between software and hardware developers: if the architecture of parallel machines is designed as prescribed by the BSP model, then software developers can rely on the BSP-like behavior of the hardware. Furthermore it should not be necessary to customize perpetually the model of applications to new hardware details in order to benefit from a higher efficiency of emerging architectures.

The BSP model is an abstraction of a machine with physically distributed memory that uses a presentation of communication as a global bundle instead of single point-to-point transfers. A BSP model machine consists of a number of processors equipped with memory, a connection network for point-to-point messages between processors, and a synchronization mechanism, which allows a barrier synchronization of all processors.

Calculations on a BSP machine are organized as a sequence of supersteps as shown in Fig. 1:

- In each superstep, each processor executes local computations and may perform communication operations.



Bandwidth-Latency Models (BSP, LogP). Fig. 1 The BSP model executes calculations in supersteps. Each superstep comprises three phases: (1) simultaneous local computations performed by each process, (2) global communication operations for exchanging data between processes, (3) barrier synchronization for finalizing the communication operation and enabling access to received data by the receiving processes

- A local computation can be executed in every time unit (step).
- The result of a communication operation does not become effective before the next superstep begins, i.e., the receiver cannot use received data until the current superstep is finished.
- At the end of every superstep, a barrier synchronization using the synchronization mechanism of the machine takes place.

The BSP model machine can be viewed as a MIMD (Multiple Instruction Multiple Data) system, because the processes can execute different instructions simultaneously. It is loosely synchronous at the superstep level, compared to the instruction-level tight synchrony in the PRAM model: within a superstep, different processes execute asynchronously at their own pace. There is a single address space, and a processor can access not only its local memory but also any remote memory in another processor, the latter imposing communication.

Within a superstep, each computation operation uses only data in the processor's local memory. These

data are put into the local memory either at the program start-up time or by the communication operations of previous supersteps. Therefore, the computation operations of a process are independent of other processes, while it is not allowed for multiple processes to read or write the same memory location in the same step. Because of the barrier synchronization, all memory and communication operations in a superstep must completely finish before any operation of the next superstep begins. These restrictions imply that a BSP computer has a sequential consistency memory model.

A program execution on a BSP machine is characterized using the following four parameters [6]:

- p : the number of processors.
- s : the computation speed of processors expressed as the number of computation steps that can be executed by a processor per second. In each step, one arithmetic operation on local data can be executed.
- l : the number of steps needed for the barrier synchronization.
- g : the average number of steps needed for transporting a memory word between two processors of the machine.

In a real parallel machine, there are many different patterns of communication between processors. For simplicity, the BSP model abstracts the communication operations using the h relation concept: an h relation is an abstraction of any communication operation, where each node sends at most h words to various nodes and each node receives at most h words. On a BSP computer, the time to realize any h relation is not longer than $g \cdot h$.

The BSP model is more realistic than the PRAM model, because it accounts for several overheads ignored by PRAM:

- To account for load imbalance, the computation time w is defined as the maximum number of steps spent on computation operations by any processor.
- The synchronization overhead is l , which has a lower bound equal to the communication network latency (i.e., the time for a word to propagate through the physical network) and is always greater than zero.
- The communication overhead is $g \cdot h$ steps, i.e., $g \cdot h$ is the time to execute the most time-consuming h relation. The value of g is platform-dependent: it is

smaller on a computer with more efficient communication support.

For a real parallel machine, the value of g depends on the bandwidth of the communication network, the communication protocols, and the communication library. The value of l depends on the diameter of the network, as well as on the communication library. Both parameters are usually estimated using benchmark programs. Since the value s is used for normalizing the values l and g , only p , l and g are independent parameters. The execution time of a BSP-program is a sum of the execution time of all supersteps. The execution time of each superstep comprises three terms: (1) maximum local computation time of all processes, w , (2) costs of global communication realizing a h -relation, and (3) costs for the barrier synchronization finalizing the superstep:

$$T_{\text{superstep}} = w + g \cdot h + l \quad (1)$$

BSP allows for overlapping of the computation, the communication, and the synchronization operations within a superstep. If all three types of operations are fully overlapped, then the time for a superstep becomes $\max(w, g \cdot h, l)$. However, usually the more conservative $w + g \cdot h + l$ is used.

The BSP model was implemented in a so-called BSplib library [5, 6] that provides operations for the initialization of supersteps, execution of communication operations, and barrier synchronizations.

The LogP Model

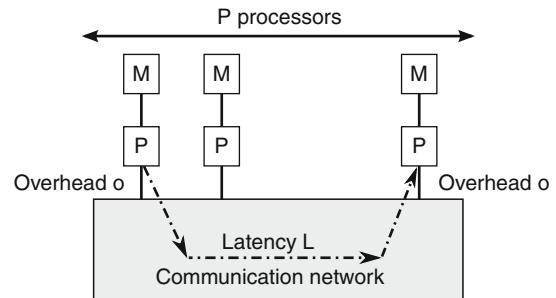
In [4], Culler et al. criticize BSP's assumption that the length of supersteps must allow to realize arbitrary h -relations, which means that the granularity has a lower bound. Also, the messages that have been sent during a superstep are not available for a recipient before the following superstep begins, even if a message is sent and received within the same superstep. Furthermore, the BSP model assumes hardware support for the synchronization mechanism, although most existing parallel machines do not provide such a support. Because of these problems, the LogP model [4] has been devised, which is arguably more closely related to the modern hardware used in parallel machines.

In analogy to BSP, the LogP model assumes that a parallel machine consists of a number of processors

equipped with memory. The processors can communicate using point-to-point messages through a communication network. The behavior of the communication is described using four parameters L , o , g , and P , which give the model the name LogP:

- L (latency) is an upper bound for the network's latency, i.e., the maximum delay between sending and receiving a message.
- o (overhead) describes the period of time in which a processor is busy with sending or receiving a message; during this time, no other work can be performed by that processor.
- g (gap) denotes the minimal timespan between sending or receiving two messages back-to-back.
- P is the number of processors of the parallel machine.

[Figure 2](#) shows a visualization of the LogP parameters [3]. Except P , all parameters are measured in time units or multiple machine cycle units. The model assumes a network with finite capacity. From the definitions of L and g , it follows that, for a given pair of source and destination nodes, the number of messages that can be on their way through the network simultaneously is limited by L/g . A processor trying to send a message that would exceed this limitation will be blocked until the network can accept the next message. This property models the network bandwidth where the parameter g reflects the bottleneck bandwidth, independent of the bottleneck's location, be it the network link, or be it the processing time spent on sending or receiving. g thus denotes an upper bound for o .



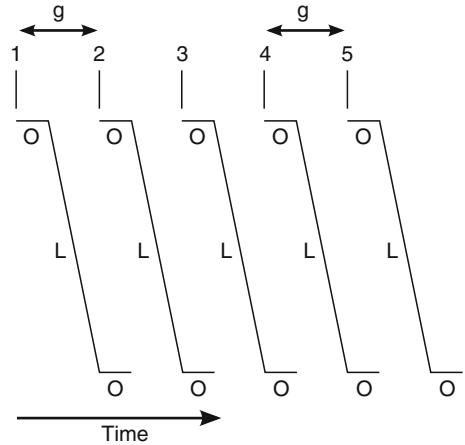
Bandwidth-Latency Models (BSP, LogP). [Fig. 2](#) Parameter visualization of the LogP model

For this capacity constraint, LogP both gets praised and criticized. The advantage of this constraint is a very realistic modeling of communication performance, as the bandwidth capacity of a communication path can get limited by any entity between the sender's memory and the receiver's memory. Due to this focus on point-to-point communication, LogP (variants) have been successfully used for modeling various computer communication problems, like the performance of the Network File System (NFS) [9] or collective communication operations from the Message Passing Interface (MPI) [8]. The disadvantage of the capacity constraint is that LogP exposes the sensitivity of a parallel computation to the communication performance of individual processors. This way, analytic performance modeling is much harder with LogP, as compared to models with a higher abstraction level like BSP [2].

While LogP assumes that the processors are working asynchronously, it requires that no message may exceed a preset size, i.e., larger messages must be fragmented. The latency of a single message is not predictable, but it is limited by L . This means, in particular, that messages can overtake each other such that the recipient may potentially receive the messages out of order. The values of the parameters L , o , and g depend on hardware characteristics, the used communication software, and the underlying communication protocols.

The time to communicate a message from one node to another (i.e., the start-up time t_0) consists of three terms: $t_0 = o + L + o$. The first o is called the *send overhead*, which is the time at the sending node to execute a message send operation in order to inject a message into the network. The second o is called the *receive overhead*, which is the time at the receiving node to execute a message receive operation. For simplicity, the two overheads are assumed equal and called the *overhead* o , i.e., o is the length of a time period that a node is engaged in sending or receiving a message. During this time, the node cannot perform other operations (e.g., overlapping computations).

In the LogP model, the runtime of an algorithm is determined as the maximum runtime across all processors. A consequence of the LogP model is that the access to a data element in memory of another processor costs $T_m = 2L + 4o$ time units (a message round-trip), of

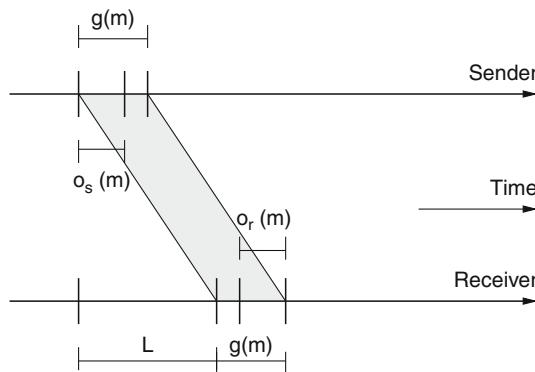


Bandwidth-Latency Models (BSP, LogP). Fig. 3 Modeling the transfer of a large message in n segments using the LogP model. The last message segment is sent at time $T_s = (n - 1) \cdot g$ and will be received at time $T_r = T_s + 2o + L$

which one half is used for reading, and the other half for writing. A sequence of n pipelined messages can be delivered in $T_n = L + 2o + (n - 1)g$ time units, as shown in Fig. 3.

A strong point of LogP is its simplicity. This simplicity, however, can, at times, lead to inaccurate performance modeling. The most obvious limitation is the restriction to small messages only. This was overcome by introducing a LogP variant, called LogGP [1]. It contains an additional parameter G (Gap per Byte) as the time needed to send a byte from a large message. $1/G$ is the bandwidth per processor. The time for sending a message consisting of n Bytes is then $T_n = o + (n - 1)G + L + o$.

A more radical extension is the *parameterized LogP* model [7], PLogP, for short. PLogP has been designed taking observations about communication software, like the Message Passing Interface (MPI), into account. These observations are (1) Overhead and gap strongly depend on the message size; some communication libraries even switch between different transfer implementations, for short, medium, and large messages. (2) Send and receive overhead can strongly differ, as the handling of asynchronously incoming messages needs a fundamentally different implementation than a synchronously invoked send operation. In the PLogP model (see Fig. 4), the original parameters o and g have been replaced by the *send overhead* $o_s(m)$, the *receive*



Bandwidth-Latency Models (BSP, LogP). Fig. 4

Visualization of a message transport with m bytes using the PLogP model. The sender is busy for $o_s(m)$ time. The message has been received at $T = L + g(m)$, out of which the receiver had been busy for $o_r(m)$ time

overhead $o_r(m)$, and the gap $g(m)$, where m is the message size. L and P remain the same as in LogP.

PLogP allows precise performance modeling when parameters for the relevant message sizes of an application are used. In [8], this has been demonstrated for MPI's collective communication operations, even for hierarchical communication networks with different sets of performance parameters. Pješivac-Grbović et al. [10] have shown that PLogP provides flexible and accurate performance modeling.

Concluding Remarks

Historically, the major focus of parallel algorithm development had been the PRAM model, which ignores data access and communication cost and considers only load balance and extra work. PRAM is very useful in understanding the inherent concurrency of an application, which is the first conceptual step in developing a parallel program; however, it does not take into account important realities of particular systems, such as the fact that data access and communication costs are often the dominant components of execution time.

The bandwidth-latency models described in this chapter articulate the performance issues against which software must be designed. Based on a clearer understanding of the importance of communication costs on modern machines, models like BSP and LogP help analyze communication cost and hence improve the structure of communication. These models expose

the important costs associated with a communication event, such as latency, bandwidth, or overhead, allowing algorithm designers to factor them into the comparative analysis of parallel algorithms. Even more, the emphasis on modeling communication cost has shifted to the cost at the nodes that are the endpoints of the communication message, such that the number of messages and contention at the endpoints have become more important than mapping to network technologies. In fact, both the BSP and LogP models ignore network topology, modeling network delay as a constant value.

An in-depth comparison of BSP and LogP has been performed in [2], showing that both models are roughly equivalent in terms of expressiveness, slightly favoring BSP for its higher-level abstraction. But it was exactly this model that was found to be too restrictive by the designers of LogP [4]. Both models have their advantages and disadvantages. LogP is better suited for modeling applications that actually use point-to-point communication, while BSP is better and simpler for data-parallel applications that fit the superstep model. The BSP model also provides an elegant framework that can be used to reason about communication and parallel performance. The major contribution of both models is the explicit acknowledgement of communication costs that are dependent on properties of the underlying machine architecture.

The BSP and LogP models are important steps toward a realistic architectural model for designing and analyzing parallel algorithms. By experimenting with the values of the key parameters in the models, it is possible to determine how an algorithm will perform across a range of architectures and how it should be structured for different architectures or for portable performance.

Related Entries

- ▶ [Amdahl's Law](#)
- ▶ [BSP \(Bulk Synchronous Parallelism\)](#)
- ▶ [Collective Communication](#)
- ▶ [Gustafson's Law](#)
- ▶ [Models of Computation, Theoretical](#)
- ▶ [PRAM \(Parallel Random Access Machines\)](#)

Bibliographic Notes and Further Reading

The presented models and their extensions were originally introduced in papers [1, 4, 11] and described in

textbooks [12–14] which were partially used for writing this entry. A number of various models similar to BSP and LogP have been proposed: Queuing Shared Memory (QSM) [15], LPRAM [16], Decomposable BSP [17], etc. Further papers in the list of references deal with particular applications of the models and their classification and standardization.

Bibliography

1. Alexandrov A, Ionescu M, Schausler KE, Scheiman C (1995) LogGP: incorporating long messages into the LogP model – one step closer towards a realistic model for parallel computation. In: 7th ACM Symposium on Parallel Algorithms and Architectures (SPAA'95), Santa Barbara, California, pp 95–105, July 1995
2. Bilardi G, Herley KT, Pietracaprina A, Pucci G, Spirakis P (1999) BSP versus LogP. Algorithmica 24:405–422
3. Culler DE, Dusseau AC, Martin RP, Schausler KE (1994) Fast parallel sorting under LogP: from theory to practice. In: Portability and Performance for Parallel Processing, Wiley, Southampton, pp 71–98, 1994
4. Culler DE, Karp R, Sahay A, Schausler KE, Santos E, Subramonian R, von Eicken T (1993) LogP: towards a realistic model of parallel computation. In: 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'93), pp 1–12, 1993
5. Goudreau MW, Hill JM, Lang K, McColl WF, Rao SD, Stefanescu DC, Suel T, Tsantilas T (1996) A proposal for a BSP Worldwide standard. Technical Report, BSP Worldwide, www.bsp-worldwide.org, 1996
6. Hill M, McColl W, Skillicorn D (1997) Questions and answers about BSP. Scientific Programming 6(3):249–274
7. Kielmann T, Bal HE, Verstoep K (2000) Fast measurement of LogP parameters for message passing platforms. In: 4th Workshop on Runtime Systems for Parallel Programming (RTSPP), held in conjunction with IPDPS 2000, May 2000
8. Kielmann T, Bal HE, Gorlatch S, Verstoep K, Hofman RFH (2001) Network performance-aware collective communication for clustered wide area systems. Parallel Computing 27(11): 1431–1456
9. Martin RP, Culler DE (1999) NFS sensitivity to high performance networks. In: ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, pp 71–82, 1999
10. Pješivac-Grbović J, Angskun T, Bosilca G, Fagg GE, Gabriel E, Dongarra JJ (2004) Performance analysis of MPI collective operations. In: 4th International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS'05), April 2004
11. Valiant LG (1990) A bridging model for parallel computation. Communications of the ACM 33(8):103–111
12. Rauber T, Rünger G (2010) Parallel programming: for multicore and cluster systems. Springer, New York
13. Hwang K, Xu Z (1998) Scalable parallel computing. WCB/McGraw-Hill, New York
14. Culler DE, Singh JP, Gupta A (1999) Parallel computer architecture – a hardware/software approach. Morgan Kaufmann, San Francisco
15. Gibbons PB, Matias Y, Ramachandran V (1999) Can a shared-memory model serve as a bridging-model for parallel computation? Theor Comput Syst 32(3):327–359
16. Aggarwal A, Chandra AK, Snir M (1990) Communication complexity of PRAMs. Theor Comput Sci 71:3–28
17. De la Torre P, Kruskal CP (1996) Submachine locality in the bulk synchronous setting. In: Proceedings of the EUROPAR'96, LNCS 1124, pp 352–358, Springer, Berlin, August 1996

Banerjee's Dependence Test

UTPAL BANERJEE

University of California at Irvine, Irvine, CA, USA

Definition

Banerjee's Test is a simple and effective data dependence test widely used in automatic vectorization and parallelization of loops. It detects dependence between statements caused by subscripted variables by analyzing the subscripts.

Discussion

Introduction

A restructuring compiler transforms a sequential program into a parallel form that can run efficiently on a parallel machine. The task of the compiler is to discover the parallelism that may be hiding in the program and bring it out into the open. The first step in this process is to compute the *dependence structure* imposed on the program operations by the sequential execution order. The parallel version of the sequential program must obey the same dependence constraints, so that it computes the same final values as the original program. (If an operation B depends on an operation A in the sequential program, then A must be executed before B in the parallel program.) To detect possible dependence between two operations, one needs to know if

they access the same memory location during sequential execution and in which order. Banerjee's Test provides a simple mechanism for dependence detection in loops when subscripted variables are involved.

In this essay, Banerjee's test is developed for one-dimensional array variables in assignment statements within a perfect loop nest. Pointers are given for extensions to more complicated situations. The first section below is on mathematical preliminaries, where certain concepts and results are presented that are essential for an understanding of the test. After that the relevant dependence concepts are explained, and then the test itself is discussed.

Mathematical Preliminaries

Linear Diophantine Equations

Let \mathbf{Z} denote the set of all integers. An integer b divides an integer a , if there exists an integer c such that $a = bc$. For a list of integers a_1, a_2, \dots, a_m , not all zero, the *greatest common divisor* or gcd is the largest positive integer that divides each member of the list. It is denoted by $\gcd(a_1, a_2, \dots, a_m)$. The gcd of a list of zeros is defined to be 0.

A *linear diophantine equation* in m variables is an equation of the form

$$a_1x_1 + a_2x_2 + \dots + a_mx_m = c$$

where the coefficients a_1, a_2, \dots, a_m are integers not all zero, c is an integer, and x_1, x_2, \dots, x_m are integer variables. A *solution* to this equation is a sequence of integers (i_1, i_2, \dots, i_m) such that $\sum_{k=1}^m a_k i_k = c$. The following theorem is a well-known result in Number Theory.

Theorem 1 *The linear diophantine equation*

$$a_1x_1 + a_2x_2 + \dots + a_mx_m = c$$

has a solution if and only if $\gcd(a_1, a_2, \dots, a_m)$ divides c .

Proof The “only if” Part is easy to prove. If the equation has a solution, then there are integers i_1, i_2, \dots, i_m such that $\sum_{k=1}^m a_k i_k = c$. Since $\gcd(a_1, a_2, \dots, a_m)$ divides each a_k , it must also divide c . To get the “if” Part (and derive the general solution), see the proof of Theorem 3.5 in [4]. \square

Lexicographic Order

For any positive integer m , the set of all integer m -vectors (i_1, i_2, \dots, i_m) is denoted by \mathbf{Z}^m . The zero vector $(0, 0, \dots, 0)$ is abbreviated as $\mathbf{0}$. Addition and subtraction of members of \mathbf{Z}^m are defined coordinate-wise in the usual way.

For $1 \leq \ell \leq m$, a relation \prec_ℓ in \mathbf{Z}^m is defined as follows: If $\mathbf{i} = (i_1, i_2, \dots, i_m)$ and $\mathbf{j} = (j_1, j_2, \dots, j_m)$ are vectors in \mathbf{Z}^m , then $\mathbf{i} \prec_\ell \mathbf{j}$ if

$$i_1 = j_1, i_2 = j_2, \dots, i_{\ell-1} = j_{\ell-1}, \text{ and } i_\ell < j_\ell.$$

The *lexicographic order* \prec in \mathbf{Z}^m is then defined by requiring that $\mathbf{i} \prec \mathbf{j}$, if $\mathbf{i} \prec_\ell \mathbf{j}$ for some ℓ in $1 \leq \ell \leq m$.

It is often convenient to write $\mathbf{i} \prec_{m+1} \mathbf{j}$ when $i_1 = j_1, i_2 = j_2, \dots, i_m = j_m$, that is, $\mathbf{i} = \mathbf{j}$. The notation $\mathbf{i} \leq \mathbf{j}$ means either $\mathbf{i} \prec \mathbf{j}$ or $\mathbf{i} = \mathbf{j}$, that is, $\mathbf{i} \prec_\ell \mathbf{j}$ for some ℓ in $1 \leq \ell \leq m+1$. Note that \leq is a total order in \mathbf{Z}^m .

The associated relations $>$ and \geq are defined in the usual way: $\mathbf{j} > \mathbf{i}$ means $\mathbf{i} \prec \mathbf{j}$, and $\mathbf{j} \geq \mathbf{i}$ means $\mathbf{i} \leq \mathbf{j}$. (\geq is also a total order in \mathbf{Z}^m .) An integer vector \mathbf{i} is *positive* if $\mathbf{i} > \mathbf{0}$, *nonnegative* if $\mathbf{i} \geq \mathbf{0}$, and *negative* if $\mathbf{i} < \mathbf{0}$.

Let \mathbf{R} denote the field of real numbers. The *sign function* $\operatorname{sgn} : \mathbf{R} \rightarrow \mathbf{Z}$ is defined by

$$\operatorname{sgn}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0, \end{cases}$$

for each x in \mathbf{R} . The *direction vector* of any vector (i_1, i_2, \dots, i_m) in \mathbf{Z}^m is $(\operatorname{sgn}(i_1), \operatorname{sgn}(i_2), \dots, \operatorname{sgn}(i_m))$. Note that a vector is positive (negative) if and only if its direction vector is positive (negative).

Extreme Values of Linear Functions

For any positive integer m , let \mathbf{R}^m denote the m -dimensional Euclidean space consisting of all real m -vectors. It is a real vector space where vector addition and scalar multiplication are defined coordinate-wise. The concepts of the previous subsection stated in terms of integer vectors can be trivially extended to the realm of real vectors. For a real number a , we define the *positive part* a^+ and the *negative part* a^- as in [2]:

$$a^+ = \frac{|a| + a}{2} = \max(a, 0) \quad \text{and}$$

$$a^- = \frac{|a| - a}{2} = \max(-a, 0).$$

Thus, $a^+ = a$ and $a^- = 0$ for $a \geq 0$, while $a^+ = 0$ and $a^- = -a$ for $a \leq 0$. For example, $4^+ = 4$, $4^- = 0$, $(-4)^+ = 0$, and $(-4)^- = 4$. The following lemma lists the basic properties of positive and negative parts of a number (Lemma 2.2 in [2], Lemma 3.1 in [4]).

Lemma 2 *For any real number a , the following statements hold:*

1. $a^+ \geq 0$, $a^- \geq 0$,
2. $a = a^+ - a^-$, $|a| = a^+ + a^-$,
3. $(-a)^+ = a^-$, $(-a)^- = a^+$,
4. $(a^+)^+ = a^+$, $(a^+)^- = 0$,
5. $(a^-)^+ = a^-$, $(a^-)^- = 0$,
6. $-a^- \leq a \leq a^+$.

The next lemma gives convenient expressions for the extreme values of a simple function. (This is Lemma 3.2 in [4]; it generalizes Lemma 2.3 in [2].)

Lemma 3 *Let a, p, q denote real constants, where $p < q$. The minimum and maximum values of the function $f(x) = ax$ on the interval $p \leq x \leq q$ are $(a^+p - a^-q)$ and $(a^+q - a^-p)$, respectively.*

Proof For $p \leq x \leq q$, the following hold:

$$a^+p \leq a^+x \leq a^+q$$

$$-a^-q \leq -a^-x \leq -a^-p,$$

since $a^+ \geq 0$ and $-a^- \leq 0$. Adding these two sets of inequalities, one gets

$$a^+p - a^-q \leq ax \leq a^+q - a^-p,$$

since $a = a^+ - a^-$. To complete the proof, note that these bounds for $f(x)$ are actually attained at the end points $x = p$ and $x = q$, and therefore they are the extreme values of f . For example, when $a \geq 0$, it follows that

$$a^+p - a^-q = ap = f(p)$$

$$a^+q - a^-p = aq = f(q).$$

1. If $x = y$, then

$$-(a - b)^-q \leq ax - by \leq (a - b)^+q.$$
2. If $x \leq y - 1$, then

$$-b - (a^- + b)^+(q - 1) \leq ax - by \leq -b + (a^+ - b)^+(q - 1).$$
3. If $x \geq y + 1$, then

$$a - (b^+ - a)^+(q - 1) \leq ax - by \leq a + (b^- + a)^+(q - 1).$$
4. If (x, y) varies freely in A , then

$$-(a^- + b^+)q \leq ax - by \leq (a^+ + b^-)q.$$

Proof By Lemma 3, $0 \leq x \leq q$ implies $-a^-q \leq ax \leq a^+q$. This result and Lemma 2 are used repeatedly in the following proof.

Case 1. Let $x = y$. Then $ax - by = (a - b)x$. Since $0 \leq x \leq q$, one gets

$$-(a - b)^-q \leq ax - by \leq (a - b)^+q.$$

Case 2. Let $0 \leq x \leq y - 1 \leq q - 1$. Then $ax \leq a^+(y - 1)$. Hence,

$$ax - by = -b + [ax - b(y - 1)] \leq -b + (a^+ - b)(y - 1).$$

Since $0 \leq y - 1 \leq q - 1$, an upper bound for $(ax - by)$ is given by:

$$ax - by \leq -b + (a^+ - b)^+(q - 1).$$

To derive a lower bound, replace a with $-a$ and b with $-b$ to get

$$-ax + by \leq b + (a^- + b)^+(q - 1),$$

so that

$$-b - (a^- + b)^+(q - 1) \leq ax - by.$$

Case 3. Let $x \geq y + 1$. Then $0 \leq y \leq x - 1 \leq q - 1$. In Case 2, replace x by y , y by x , a by $-b$, and b by $-a$, to get

$$\begin{aligned} a - ((-b)^- - a)^+(q - 1) &\leq (-b)y - (-a)x \\ &\leq a + ((-b)^+ + a)^+(q - 1), \end{aligned}$$

or

$$a - (b^+ - a)^+(q - 1) \leq ax - by \leq a + (b^- + a)^+(q - 1).$$

Case 4. Since $0 \leq x \leq q$, one gets $-a^-q \leq ax \leq a^+q$. Also, since $0 \leq y \leq q$, it follows that $-b^-q \leq by \leq b^+q$, that is, $-b^+q \leq -by \leq b^-q$. Hence,

$$-a^-q - b^+q \leq ax - by \leq a^+q + b^-q.$$

This gives the bounds for Case 4.

As in the proof of Lemma 3, it can be shown in each case that each bound is actually attained at some point of the corresponding domain. Hence, the bounds represent the extreme values of f in each case. \square

The Euclidean space \mathbf{R}^m is an inner product space, where the inner product of two vectors $\mathbf{x} = (x_1, x_2, \dots, x_m)$ and $\mathbf{y} = (y_1, y_2, \dots, y_m)$ is defined by $\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{k=1}^m x_k y_k$. The inner product defines a norm (length), the norm defines a metric (distance), and the metric defines a topology. In this topological vector space, one can talk about bounded sets, open and closed sets, compact sets, and connected sets. The following theorem is easily derived from well-known results in topology. (See the topology book by Kelley [8] for details.)

Theorem 5 *Let $f : \mathbf{R}^m \rightarrow \mathbf{R}$ be a continuous function. If a set $A \subset \mathbf{R}^m$ is closed, bounded, and connected, then $f(A)$ is a finite closed interval of \mathbf{R} .*

Proof Note that R^m and \mathbf{R} are both Euclidean spaces. A subset of a Euclidean space is compact if and only if it is closed and bounded. Thus, the set A is compact and connected. Since f is continuous, it maps a compact set onto a compact set, and a connected set onto a connected set. Hence, the set $f(A)$ is compact and connected, that is, closed, bounded, and connected. Therefore, $f(A)$ must be a finite closed interval of \mathbf{R} . \square

Corollary 1 *Let $f : \mathbf{R}^m \rightarrow \mathbf{R}$ denote a continuous function, and let A be a closed, bounded, and connected subset of \mathbf{R}^m . Then f assumes a minimum and a maximum value on A . And for any real number c satisfying the inequalities*

$$\min_{x \in A} f(x) \leq c \leq \max_{x \in A} f(x),$$

the equation $f(x) = c$ has a solution $x_0 \in A$.

Proof By Theorem 5, the image $f(A)$ of A is a finite closed interval $[\alpha, \beta]$ of \mathbf{R} . Then for each $x \in A$, one has $\alpha \leq f(x) \leq \beta$. So, α is a lower bound and β an upper bound for f on A . Since $\alpha \in f(A)$ and $\beta \in f(A)$, there are points $x_1, x_2 \in A$ such that $f(x_1) = \alpha$ and $f(x_2) = \beta$. Thus, f assumes a minimum and a maximum value on A , and $\alpha = \min_{x \in A} f(x)$ and $\beta = \max_{x \in A} f(x)$. If $\alpha \leq c \leq \beta$, then $c \in f(A)$, that is, $f(x_0) = c$ for some $x_0 \in A$. \square

Dependence Concepts

For more details on the material of this section, see Chapter 4 in [4]. An *assignment statement* has the form

$$S : \quad x = E$$

where S is a label, x a variable, and E an expression. Such a statement reads the memory locations specified in E , and writes the location x . The *output variable* of S is x and its *input variables* are the variables in E . Let $\text{Out}(S) = \{x\}$ and denote by $\text{In}(S)$ the set of all input variables of S .

The basic program model is a perfect nest of loops $\mathbf{L} = (L_1, L_2, \dots, L_m)$ shown in Fig. 1. For $1 \leq k \leq m$, the index variable I_k of L_k runs from 0 to some positive integer N_k in steps of 1. The *index vector* of the loop nest is $\mathbf{I} = (I_1, I_2, \dots, I_m)$. An *index point* or *index value* of the nest is a possible value of the index vector, that is, a vector $\mathbf{i} = (i_1, i_2, \dots, i_m) \in \mathbf{Z}^m$ such that $0 \leq i_k \leq N_k$ for $1 \leq k \leq m$. The subset of \mathbf{Z}^m consisting of all index points is the *index space* of the loop nest. During sequential execution of the nest, the index vector starts at the index point $\mathbf{0} = (0, 0, \dots, 0)$ and ends at the point (N_1, N_2, \dots, N_m) after traversing the entire index space in the lexicographic order.

The *body* of the loop nest \mathbf{L} is denoted by $H(\mathbf{I})$ or H ; it is assumed to be a sequence of assignment statements. A given index value \mathbf{i} defines a particular *instance* $H(\mathbf{i})$ of $H(\mathbf{I})$, which is an *iteration* of \mathbf{L} . An iteration $H(\mathbf{i})$ is executed before an iteration $H(\mathbf{j})$ if and only if $\mathbf{i} < \mathbf{j}$.

A typical statement in the body of the loop nest is denoted by $S(\mathbf{I})$ or S , and its instance for an index value \mathbf{i} is written as $S(\mathbf{i})$. Let S and T denote any two (not necessarily distinct) statements in the body. Statement T depends on statement S , if there exist a memory location M , and two index points \mathbf{i} and \mathbf{j} , such that

```

 $L_1 : \quad \text{do } I_1 = 0, N_1, 1$ 
 $L_2 : \quad \text{do } I_2 = 0, N_2, 1$ 
 $\vdots \quad \vdots$ 
 $L_m : \quad \text{do } I_m = 0, N_m, 1$ 
 $H(I_1, I_2, \dots, I_m)$ 
 $\text{enddo}$ 
 $\vdots$ 
 $\text{enddo}$ 

```

Banerjee's Dependence Test. Fig. 1 A perfect loop nest

1. The instances $S(\mathbf{i})$ of S and $T(\mathbf{j})$ of T both reference (read or write) \mathcal{M}
2. In the sequential execution of the program, $S(\mathbf{i})$ is executed before $T(\mathbf{j})$.

If \mathbf{i} and \mathbf{j} are a pair of index points that satisfy these two conditions, then it is convenient to say that $T(\mathbf{j})$ depends on $S(\mathbf{i})$. Thus, T depends on S if and only if at least one instance of T depends on at least one instance of S . The concept of dependence can have various attributes as described below.

Let $\mathbf{i} = (i_1, i_2, \dots, i_m)$ and $\mathbf{j} = (j_1, j_2, \dots, j_m)$ denote a pair of index points, such that $T(\mathbf{j})$ depends on $S(\mathbf{i})$. If S and T are distinct and S appears lexically before T in H , then $S(\mathbf{i})$ is executed before $T(\mathbf{j})$ if and only if $\mathbf{i} \leq \mathbf{j}$. Otherwise, $S(\mathbf{i})$ is executed before $T(\mathbf{j})$ if and only if $\mathbf{i} < \mathbf{j}$. If $\mathbf{i} \leq \mathbf{j}$, there is a unique integer ℓ in $1 \leq \ell \leq m+1$ such that $\mathbf{i} <_{\ell} \mathbf{j}$. If $\mathbf{i} < \mathbf{j}$, there is a unique integer ℓ in $1 \leq \ell \leq m$ such that $\mathbf{i} <_{\ell} \mathbf{j}$. This integer ℓ is a *dependence level* for the dependence of T on S . The vector $\mathbf{d} \in \mathbb{Z}^m$, defined by $\mathbf{d} = \mathbf{j} - \mathbf{i}$, is a *distance vector* for the same dependence. Also, the direction vector σ of \mathbf{d} , defined by

$$\sigma = (\operatorname{sgn}(j_1 - i_1), \operatorname{sgn}(j_2 - i_2), \dots, \operatorname{sgn}(j_m - i_m)),$$

is a *direction vector* for this dependence. Note that both \mathbf{d} and σ are nonnegative vectors if S precedes T in H , and both are positive vectors otherwise. The dependence of $T(\mathbf{j})$ on $S(\mathbf{i})$ is *carried* by the loop L_{ℓ} if the dependence level ℓ is between 1 and m . Otherwise, the dependence is *loop-independent*.

A statement can reference a memory location only through one of its variables. The definition of dependence is now extended to make explicit the role played by the variables in the statements under consideration. A variable $u(\mathbf{I})$ of the statement S and a variable $v(\mathbf{I})$ of the statement T cause a dependence of T on S , if there are index points \mathbf{i} and \mathbf{j} , such that

1. The instance $u(\mathbf{i})$ of $u(\mathbf{I})$ and the instance $v(\mathbf{j})$ of $v(\mathbf{I})$ both represent the same memory location;
2. In the sequential execution of the program, $S(\mathbf{i})$ is executed before $T(\mathbf{j})$.

If these two conditions hold, then the dependence caused by $u(\mathbf{I})$ and $v(\mathbf{I})$ is characterized as

1. A *flow dependence* if

$$u(\mathbf{I}) \in \operatorname{Out}(S) \text{ and } v(\mathbf{I}) \in \operatorname{In}(T);$$

2. An *anti-dependence* if $u(\mathbf{I}) \in \operatorname{In}(S)$ and $v(\mathbf{I}) \in \operatorname{Out}(T)$;
3. An *output dependence* if $u(\mathbf{I}) \in \operatorname{Out}(S)$ and $v(\mathbf{I}) \in \operatorname{Out}(T)$;
4. An *input dependence* if $u(\mathbf{I}) \in \operatorname{In}(S)$ and $v(\mathbf{I}) \in \operatorname{In}(T)$.

Banerjee's Test

In its simplest form, Banerjee's Test is a necessary condition for the existence of dependence of one statement on another at a given level, caused by a pair of one-dimensional array variables. This form of the test is described in [Theorem 6](#). [Theorem 7](#) gives the version of the test that deals with dependence with a fixed direction vector. Later on, pointers are given for extending the test in different directions.

Theorem 6 Consider any two assignment statements S and T in the body H of the loop nest of [Fig. 1](#). Let $X(f(\mathbf{I}))$ denote a variable of S and $X(g(\mathbf{I}))$ a variable of T , where X is a one-dimensional array, $f(\mathbf{I}) = a_0 + \sum_{k=1}^m a_k I_k$, $g(\mathbf{I}) = b_0 + \sum_{k=1}^m b_k I_k$, and the a 's and the b 's are all integer constants. If $X(f(\mathbf{I}))$ and $X(g(\mathbf{I}))$ cause a dependence of T on S at a level ℓ , then the following two conditions hold:

- (A) $\gcd(a_1 - b_1, \dots, a_{\ell-1} - b_{\ell-1}, a_{\ell}, \dots, a_m, b_{\ell}, \dots, b_m)$ divides $(b_0 - a_0)$;
 - (B) $\alpha \leq b_0 - a_{\ell-1} \leq \beta$, where
- $$\begin{aligned} \alpha &= -b_{\ell} - \sum_{k=1}^{\ell-1} (a_k - b_k)^- N_k - (a_{\ell}^- + b_{\ell})^+ (N_{\ell} - 1) \\ &\quad - \sum_{k=\ell+1}^m (a_k^- + b_k^+) N_k \\ \beta &= -b_{\ell} + \sum_{k=1}^{\ell-1} (a_k - b_k)^+ N_k + (a_{\ell}^+ - b_{\ell})^+ (N_{\ell} - 1) \\ &\quad + \sum_{k=\ell+1}^m (a_k^+ + b_k^-) N_k. \end{aligned}$$

Proof Assume that the variables $X(f(\mathbf{I}))$ and $X(g(\mathbf{I}))$ do cause a dependence of statement T on statement S at a level ℓ . If S precedes T in H , the dependence level ℓ could be any integer in $1 \leq \ell \leq m+1$. Otherwise, ℓ must be in the range $1 \leq \ell \leq m$. By hypothesis, there exist two index points $\mathbf{i} = (i_1, i_2, \dots, i_m)$ and $\mathbf{j} = (j_1, j_2, \dots, j_m)$ such that $\mathbf{i} <_{\ell} \mathbf{j}$, and $X(f(\mathbf{i}))$ and $X(g(\mathbf{j}))$ represent the same memory location.

The restrictions on $i_1, i_2, \dots, i_m, j_1, j_2, \dots, j_m$ are as follows:

$$\left. \begin{array}{l} 0 \leq i_k \leq N_k \\ 0 \leq j_k \leq N_k \end{array} \right\} \quad (1 \leq k \leq m) \quad (1)$$

$$i_k = j_k \quad (1 \leq k \leq \ell - 1) \quad (2)$$

$$i_\ell \leq j_\ell - 1. \quad (3)$$

(As i_ℓ and j_ℓ are integers, $i_\ell < j_\ell$ means $i_\ell \leq j_\ell - 1$.)

Since $f(\mathbf{i})$ and $g(\mathbf{j})$ must be identical, it follows that

$$\begin{aligned} b_0 - a_0 &= \sum_{k=1}^{\ell-1} (a_k i_k - b_k j_k) + (a_\ell i_\ell - b_\ell j_\ell) \\ &\quad + \sum_{k=\ell+1}^m (a_k i_k - b_k j_k). \end{aligned} \quad (4)$$

Because of (2), this equation is equivalent to

$$\begin{aligned} b_0 - a_0 &= \sum_{k=1}^{\ell-1} (a_k - b_k) i_k + (a_\ell i_\ell - b_\ell j_\ell) \\ &\quad + \sum_{k=\ell+1}^m (a_k i_k - b_k j_k). \end{aligned} \quad (5)$$

First, think of $i_1, i_2, \dots, i_m, j_\ell, j_{\ell+1}, \dots, j_m$ as integer variables. Since the linear diophantine equation (5) has a solution, condition (A) of the theorem is implied by [Theorem 1](#).

Next, think of $i_1, i_2, \dots, i_m, j_\ell, j_{\ell+1}, \dots, j_m$ as real variables. Note that for $1 \leq k < t \leq m$, there is no relation between the pair of variables (i_k, j_k) and the pair of variables (i_t, j_t) . Hence, the minimum (maximum) value of the right-hand side of (4) can be computed by summing the minimum (maximum) values of all the individual terms $(a_k i_k - b_k j_k)$. For each k in $1 \leq k \leq m$, one can compute the extreme values of the term $(a_k i_k - b_k j_k)$ by using a suitable case of Theorem 4. It is then clear that α is the minimum value of the right-hand side of (4), and β is its maximum value. Hence, $(b_0 - a_0)$ must lie between α and β . This is condition (B) of the theorem. \square

For the dependence problem of Theorem 6, equation (4) or (5) is the *dependence equation*, condition (A) is the *gcd Test*, and condition (B) is *Banerjee's Test*.

Remarks

1. In general, the two conditions of Theorem 6 are necessary for existence of dependence, but they are not sufficient. Suppose both conditions hold. First, think in terms of real variables. Let P denote the subset of \mathbf{R}^{2m} defined by the inequalities (1)–(3). Then P is a closed, bounded, and connected set. The right-hand side of (5) represents a real-valued continuous function on \mathbf{R}^{2m} . Its extreme values on P are given by α and β . Condition (B) implies that there is a *real* solution to equation (5) with the constraints (1)–(3) (by Corollary 1 to [Theorem 5](#)). On the other hand, condition (A) implies that there is an *integer* solution to equation (5) without any further constraints (by Theorem 1). Theoretically, the two conditions together do not quite imply the existence of an *integer* solution with the constraints needed to guarantee the dependence of $T(\mathbf{j})$ on $S(\mathbf{i})$. However, note that using Theorem 6, one would never falsely conclude that a dependence does not exist when in fact it does.
2. If one of the two conditions of Theorem 6 fails to hold, then there is no dependence. If both of them hold, there may or may not be dependence. In practice, however, it usually turns out that when the conditions hold, there is dependence. This can be explained by the fact that there are certain types of array subscripts for which the conditions are indeed sufficient, and these are the types most commonly encountered in practice. Theorem 4.3 in [2] shows that the conditions are sufficient, if there is an integer $t > 0$ such that $a_k, b_k \in \{-t, 0, t\}$ for $1 \leq k \leq m$. Psarris et al. [11] prove the sufficiency for another large class of subscripts commonly found in real programs.

Example 1 Consider the loop nest of [Fig. 2](#), where X is a one-dimensional array, and the constant terms a_0, b_0 in the subscripts are integers unspecified for the moment. Suppose one needs to check if T is output-dependent on S at level 2. For this problem, $m = 3$, $\ell = 2$, and

$$(N_1, N_2, N_3) = (100, 50, 40),$$

$$(a_1, a_2, a_3) = (1, 2, -1), (b_1, b_2, b_3) = (3, -1, 2).$$

Since $\text{gcd}(a_1 - b_1, a_2, a_3, b_2, b_3) = \text{gcd}(-2, 2, -1, -1, 2) = 1$, condition (A) of Theorem 6 is always satisfied. To test

```

 $L_1 : \text{do } I_1 = 0, 100, 1$ 
 $L_2 : \text{do } I_2 = 0, 50, 1$ 
 $L_3 : \text{do } I_3 = 0, 40, 1$ 
 $S: \quad X(a_0 + I_1 + 2I_2 - I_3) = \dots$ 
 $T: \quad X(b_0 + 3I_1 - I_2 + 2I_3) = \dots$ 
    enddo
  enddo
enddo

```

Banerjee's Dependence Test. Fig. 2 Loop nest of example 1

condition (B), evaluate α and β :

$$\begin{aligned}\alpha &= -b_2 - (a_1 - b_1)^- N_1 - (a_2^- + b_2)^+ (N_2 - 1) \\ &\quad - (a_3^- + b_3^+) N_3 = -319 \\ \beta &= -b_2 + (a_1 - b_1)^+ N_1 + (a_2^+ - b_2)^+ (N_2 - 1) \\ &\quad + (a_3^+ + b_3^-) N_3 = 148.\end{aligned}$$

Condition (B) then becomes

$$-319 \leq b_0 - a_0 \leq 148.$$

First, take $a_0 = -50$ and $b_0 = 100$. Then $(b_0 - a_0)$ is outside this range, and hence Banerjee's Test is not satisfied. By Theorem 6, statement T is not output-dependent on statement S at level 2.

Next, take $a_0 = b_0 = 0$. Then $b_0 - a_0$ is within the range, and Banerjee's Test is satisfied. Theorem 6 cannot guarantee the existence of the dependence under question. However, there exist two index points $\mathbf{i} = (20, 10, 10)$ and $\mathbf{j} = (20, 40, 5)$ of the loop nest, such that $\mathbf{i} <_2 \mathbf{j}$, the instance $S(\mathbf{i})$ of statement S is executed before the instance $T(\mathbf{j})$ of statement T , and both instances write the memory location $X(30)$. Thus, statement T is indeed output-dependent on statement S at level 2.

Theorem 6 dealt with dependence at a level. It is now extended to a result that deals with dependence with a given direction vector. A direction vector here has the general form $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_m)$, where each σ_k is either unspecified, or has one of the values 0, 1, -1. When σ_k is unspecified, one writes $\sigma_k = *$. (As components of a direction vector, many authors use ' $=$ ', ' $<$ ', ' $>$ ' in place of 0, 1, -1, respectively.)

Theorem 7 Consider any two assignment statements S and T in the body H of the loop nest of Fig. 1. Let $X(f(\mathbf{I}))$ denote a variable of S and $X(g(\mathbf{I}))$ a variable of T , where X is a one-dimensional array, $f(\mathbf{I}) = a_0 + \sum_{k=1}^m a_k I_k$, and the a 's and the b 's are all

integer constants. If these variables cause a dependence of T on S with a direction vector $(\sigma_1, \sigma_2, \dots, \sigma_m)$, then the following two conditions hold:

(A) The gcd of all integers in the three lists

$$\{a_k - b_k : \sigma_k = 0\}, \{a_k : \sigma_k \neq 0\}, \{b_k : \sigma_k \neq 0\}$$

divides $(b_0 - a_0)$;

(B) $\alpha \leq b_0 - a_0 \leq \beta$, where

$$\begin{aligned}\alpha &= - \sum_{\sigma_k=0} (a_k - b_k)^- N_k \\ &\quad - \sum_{\sigma_k=1} [b_k + (a_k^- + b_k)^+ (N_k - 1)] \\ &\quad + \sum_{\sigma_k=-1} [a_k - (b_k^+ - a_k)^+ (N_k - 1)] \\ &\quad - \sum_{\sigma_k= *} (a_k^- + b_k^+) N_k \\ \beta &= \sum_{\sigma_k=0} (a_k - b_k)^+ N_k \\ &\quad + \sum_{\sigma_k=1} [-b_k + (a_k^+ - b_k)^+ (N_k - 1)] \\ &\quad + \sum_{\sigma_k=-1} [a_k + (b_k^- + a_k)^+ (N_k - 1)] \\ &\quad + \sum_{\sigma_k= *} (a_k^+ + b_k^-) N_k.\end{aligned}$$

Proof All expressions of the form $(a_k - b_k)$, where $1 \leq k \leq m$ and $\sigma_k = 0$, constitute the list $\{a_k - b_k : \sigma_k = 0\}$. The other two lists have similar meanings. The sum $\sum_{\sigma_k=0}$ is taken over all values of k in $1 \leq k \leq m$ such that $\sigma_k = 0$. The other three sums have similar meanings. As in the case of Theorem 6, the proof of this theorem follows directly from Theorem 1 and Theorem 4. \square

For the dependence problem with a direction vector, condition (A) of Theorem 7 is the gcd Test and condition (B) is Banerjee's Test. Comments similar to those given in Remarks 1 also apply to Theorem 7.

The expressions for α and β in Banerjee's Test may look complicated even though their numerical evaluation is quite straightforward. With the goal of developing the test quickly while avoiding complicated formulas, the perfect loop nest of Fig. 1 was taken as the model program, and the variables were assumed to be one-dimensional array elements. However, Banerjee's Test can be extended to cover multidimensional array elements in a much more general program. Detailed references are given in the section on further reading.

Related Entries

- ▶ [Code Generation](#)
- ▶ [Dependence Abstractions](#)
- ▶ [Dependences](#)
- ▶ [Loop Nest Parallelization](#)
- ▶ [Omega Test](#)
- ▶ [Parallelization, Automatic](#)
- ▶ [Parallelism Detection in Nested Loops, Optimal](#)
- ▶ [Unimodular Transformations](#)

Bibliographic Notes and Further Reading

Banerjee's Test first appeared in Utpal Banerjee's MS thesis [2] at the University of Illinois, Urbana-Champaign, in 1976. In that thesis, the test is developed for checking dependence at any level when the subscript functions $f(\mathbf{I})$ and $g(\mathbf{I})$ are polynomials in I_1, I_2, \dots, I_m . The case where $f(\mathbf{I})$ and $g(\mathbf{I})$ are linear (affine) functions of the index variables is then derived as a special case (Theorem 4.1). Theorem 4.1 of [2] appeared as Theorem 5 in the 1979 paper by Banerjee, Chen, Kuck, and Towle [7]. Theorem 6 presented here is essentially the same theorem, but has a stronger gcd Test. (The stronger gcd Test was pointed out by Kennedy in [9].)

Banerjee's Test for a direction vector of the form $\sigma = (0, \dots, 0, 1, -1, *, \dots, *)$ was given by Kennedy in [9]. It is a special case of Theorem 7 presented here. See also the comprehensive paper by Allen and Kennedy [1]. Wolfe and Banerjee [12] give an extensive treatment of the dependence problem involving direction vectors. (The definition of the negative part of a number used in [12] is slightly different from that used here and in most other publications.)

The first book on dependence analysis [3] was published in 1988; it gave the earliest coverage of Banerjee's Test in a book form. *Dependence Analysis* [6] published in 1997 is a completely new work that subsumes the material in [3].

It is straightforward to extend theorems 6 and 7 to the case where we allow the loop limits to be arbitrary integer constants, as long as the stride of each loop is kept at 1. See theorems 6.2 and 6.3 in [6]. This model can be further extended to test the dependence of a statement T on a statement S when the nest of loops enclosing S is different from the nest enclosing T . See theorems 6.4 and 6.5 in [6].

A loop of the form “**do** $I = p, q, \theta$,” where p, q, θ are integers and $\theta \neq 0$, can be converted into the loop “**do** $\hat{I} = 0, N, 1$,” where $I = p + \hat{I}\theta$ and $N = \lfloor (q - p)/\theta \rfloor$. The new variable \hat{I} is the *iteration variable* of the loop. Using this process of loop normalization, one can convert any nest of loops to a standard form if the loop limits and strides are integer constants. (See [6].)

Consider now the dependence problem posed by variables that come from a multidimensional array, where each subscript is a linear (affine) function of the index variables. The gcd Test can be generalized to handle this case; see Section 5.2 of [4] and Section 5.4 of [6]. Also, Banerjee's Test can be applied separately to the subscripts in each dimension; see Theorem 6.6 in [6]. If the test is not satisfied in one particular dimension, then there is no dependence as a whole. But, if the test is satisfied in each dimension, there is no definite conclusion. Another alternative is array linearization; see Section 6.5 in [3]. Zhiyuan Li et al. [10] did a study of dependence analysis for multidimensional array elements, that did not involve subscript-by-subscript testing, nor array linearization. The general method presented in Chapter 6 of [6] includes Li's λ -test.

The dependence problem is quite complex when one has an arbitrary loop nest with loop limits that are linear functions of index variables, and multidimensional array elements with linear subscripts. To understand the general problem and some methods of solution, see the developments in [4] and [6].

Many researchers have studied Banerjee's Test over the years; the test in various forms can be found in many publications. *Dependence Analysis* [6] covers this test quite extensively. For a systematic development of the dependence problem, descriptions of the needed mathematical tools, and applications of dependence analysis to program transformations, see the books [4–6] in the series on Loop Transformations for Restructuring compilers.

Bibliography

1. Allen JR, Kennedy K (Oct 1987) Automatic translation of FORTRAN programs to vector form. *ACM Trans Program Lang Syst* 9(4):491–542
2. Banerjee U (Nov 1976) Data dependence in ordinary programs. MS Thesis, Report 76-837, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois

3. Banerjee U (1988) Dependence analysis for supercomputing. Kluwer, Norwell
4. Banerjee U (1993) Loop transformations for restructuring compilers: the foundations. Kluwer, Norwell
5. Banerjee U (1994) Loop transformations for restructuring compilers: loop parallelization. Kluwer, Norwell
6. Banerjee U (1997) Loop transformations for restructuring compilers: dependence analysis. Kluwer, Norwell
7. Banerjee U, Chen S-C, Kuck DJ, Towle RA (Sept 1979) Time and parallel processor bounds for FORTRAN-like loops. IEEE Trans Comput C-28(9):660–670
8. Kelley JL (1955) General topology. D. Van Nostrand Co., New York
9. Kennedy K (Oct 1980) Automatic translation of FORTRAN programs to vector form. Rice Technical Report 476-029-4, Department of Mathematical Sciences, Rice University, Houston, Texas
10. Li Z, Yew P-C, Zhu C-Q (Jan 1990) An efficient data dependence analysis for parallelizing compilers. IEEE Trans Parallel Distrib Syst 1(1):26–34
11. Psarris K, Klappholz D, Kong X (June 1991) On the accuracy of the Banerjee test. J Parallel Distrib Comput 12(2):152–157
12. Wolfe M, Banerjee U (Apr 1987) Data dependence and its application to parallel processing. Int J Parallel Programming 16(2):137–178

Barnes-Hut

- [N-Body Computational Methods](#)

Barriers

- [NVIDIA GPU](#)
- [Synchronization](#)

Basic Linear Algebra Subprograms (BLAS)

- [BLAS \(Basic Linear Algebra Subprograms\)](#)

Behavioral Equivalences

ROCCO DE NICOLA
Università degli Studi di Firenze, Firenze, Italy

Synonyms

[Behavioral relations](#); [Extensional equivalences](#)

Definition

Behavioral equivalences serve to establish in which cases two reactive (possible concurrent) systems offer similar interaction capabilities relatively to other systems representing their operating environment. Behavioral equivalences have been mainly developed in the context of *process algebras*, mathematically rigorous languages that have been used for describing and verifying properties of concurrent communicating systems. By relying on the so-called structural operational semantics (SOS), labeled transition systems are associated to each term of a process algebra. Behavioral equivalences are used to abstract from unwanted details and identify those labeled transition systems that react “similarly” to external experiments. Due to the large number of properties which may be relevant in the analysis of concurrent systems, many different theories of equivalences have been proposed in the literature. The main contenders consider those systems equivalent that (1) perform the same sequences of actions, or (2) perform the same sequences of actions and after each sequence are ready to accept the same sets of actions, or (3) perform the same sequences of actions and after each sequence exhibit, recursively, the same behavior. This approach leads to many different equivalences that preserve significantly different properties of systems.

Introduction

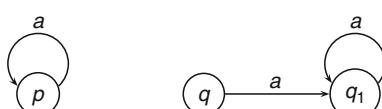
In many cases, it is useful to have theories which can be used to establish whether two systems are equivalent or whether one is a satisfactory “approximation” of another. It can be said that a system S_1 is equivalent to a system S_2 whenever “some” aspects of the externally observable behavior of the two systems are compatible. If the same formalism is used to model what is required of a system (its *specification*) and how it can actually be built (its *implementation*), then it is possible to use theories based on equivalences to prove that a particular concrete description is correct with respect to a given abstract one. If a step-wise development method is used, equivalences may permit substituting large specifications with equivalent concise ones. In general it is useful to be able to interchange subsystems proved behaviorally equivalent, in the sense that one subsystem may replace another as part of a larger system without affecting the behavior of the overall system.

The kind of equivalences, or approximations, involved depends very heavily on how the systems under consideration will be used. In fact, the way a system is used determines the behavioral aspects which must be taken into account and those which can be ignored. It is then important to know, for the considered equivalence, the systems properties it preserves.

In spite of the general agreement on taking an extensional approach for defining the equivalence of concurrent or nondeterministic systems, there is still disagreement on what “reasonable” observations are and how their outcomes can be used to distinguish or identify systems. Many different theories of equivalences have been proposed in the literature for models which are intended to be used to describe and reason about concurrent or nondeterministic systems. This is mainly due to the large number of properties which may be relevant in the analysis of such systems. Almost all the proposed equivalences are based on the idea that two systems are equivalent whenever no external observation can distinguish them. In fact, for any given system it is not its internal structure which is of interest but its behavior with respect to the outside world, i.e., its effect on the environment and its reactions to stimuli from the environment.

One of the most successful approaches for describing the formal, precise, behavior of concurrent systems is the so-called *operational semantics*. Within this approach, concurrent programs or systems are modeled as labeled transition systems (LTSs) that consist of a set of states, a set of transition labels, and a transition relation. The states of the transition systems are programs, while the labels of the transitions between states represent the actions (instructions) or the interactions that are possible in a given state.

When defining behavioral equivalence of concurrent systems described as LTSs, one might think that it is possible to consider systems equivalent if they give rise to the same (isomorphic) LTSs. Unfortunately, this would lead to unwanted distinctions, e.g., it would consider the two LTSs below different



in spite of the fact that their behavior is the same; they can (only) execute infinitely many a -actions, and they should thus be considered equivalent.

The basic principles for any reasonable equivalence can be summarized as follows. It should:

- Abstract from states (consider only the actions)
- Abstract from internal behavior
- Identify processes whose LTSs are isomorphic
- Consider two processes equivalent only if both can execute the same actions sequences
- Allow to replace a subprocess by an equivalent counterpart without changing the overall semantics of the system

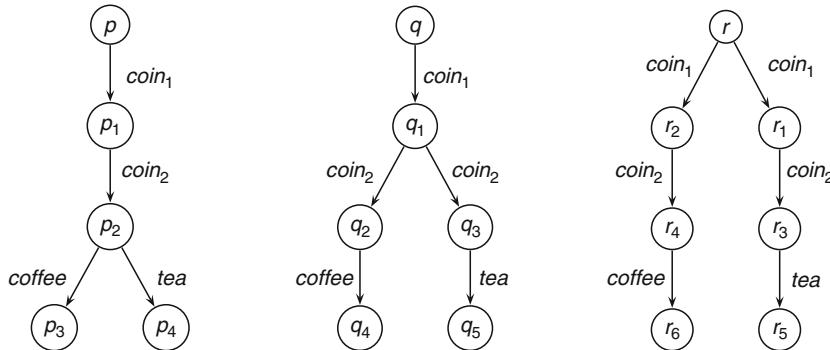
However, these criteria are not sufficiently insightful and discriminative, and the above adequacy requirements turn out to be still too loose. They have given rise to many different kinds of equivalences, even when all actions are considered visible.

The main equivalences over LTSs introduced in the literature consider as equivalent those systems that:

1. Perform the same sequences of actions
2. Perform the same sequences of actions and after each sequence are ready to accept the same sets of actions
3. Perform the same sequences of actions and after each sequence exhibit, recursively, the same behavior

These three different criteria lead to three groups of equivalences that are known as *traces* equivalences, *decorated-traces* equivalences, and *bisimulation-based* equivalences. Equivalences in different classes behave differently relatively to the three-labeled transition systems in Fig. 1. The three systems represent the specifications of three vending machines that accept two coins and deliver coffee or tea. The trace-based equivalences equate all of them, the bisimulation-based equivalences distinguish all of them, and the decorated traces distinguish the leftmost system from the other two, but equate the central and the rightmost one.

Many of these equivalences have been reviewed [11]; here, only the main ones are presented. First, equivalences that consider invisible (τ) actions just normal actions will be presented, then their variants that abstract from internal actions will be introduced.



Behavioral Equivalences. Fig. 1 Three vending machines

The equivalences will be formally defined on states of LTSs of the form $\langle Q, A_\tau, \xrightarrow{\mu} \rangle$, where Q is a set of states, ranging over p, q, p', q_1, \dots , A_τ is the set of labels, ranging over a, b, c, \dots , that also contains the distinct silent action τ , and $\xrightarrow{\mu}$ is the set of transitions. In the following, s will denote a generic element of A_τ^* , the set of all sequences of actions that a process might perform.

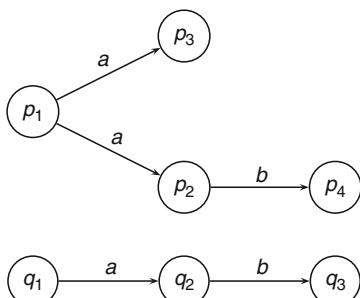
Traces Equivalence

The first equivalence is known as *traces equivalence* and is perhaps the simplest of all; it is imported from automata theory that considers those automata equivalent that generate the same language. Intuitively, two processes are deemed traces equivalent if and only if they can perform exactly the same sequences of actions. In the formal definition, $p \xrightarrow{s} p'$, with $s = \mu_1 \mu_2 \dots \mu_n$, denotes the sequence $p \xrightarrow{\mu_1} p_1 \xrightarrow{\mu_2} p_2 \dots \xrightarrow{\mu_n} p'$ of transitions.

Two states p and q are *traces equivalent* ($p \simeq_T q$) if:

1. $p \xrightarrow{s} p'$ implies $q \xrightarrow{s} q'$ for some q' and
2. $q \xrightarrow{s} q'$ implies $p \xrightarrow{s} p'$ for some p' .

A drawback of \simeq_T is that it is not sensitive to deadlocks. For example, if we consider the two LTSs below:



we have that $P_1 \simeq_T Q_1$, but P_1 , unlike Q_1 , after performing action a , can reach a state in which it cannot perform any action, i.e., a deadlocked state.

Traces equivalence identifies all of the three LTSs of Fig. 1. Indeed, it is not difficult to see that the three vending machines can perform the same sequences of visible actions. Nevertheless, a customer with definite preferences for coffee who is offered to choose between the three machines would definitely select to interact with the leftmost one since the others do not let him choose what to drink.

Bisimulation Equivalence

The classical alternative to traces equivalence is bisimilarity (also known as observational equivalence) that considers equivalent two systems that can simulate each other step after step [8]. Bisimilarity is based on the notion of bisimulation:

A relation $R \subseteq Q \times Q$ is a *bisimulation* if, for any pair of states p and q such that $\langle p, q \rangle \in R$, the following holds:

1. For all $\mu \in A_\tau$ and $p' \in Q$, if $p \xrightarrow{\mu} p'$ then $q \xrightarrow{\mu} q'$ for some $q' \in Q$ s.t. $\langle p', q' \rangle \in R$
2. For all $\mu \in A_\tau$ and $q' \in Q$, if $q \xrightarrow{\mu} q'$ then $p \xrightarrow{\mu} p'$ for some $p' \in Q$ s.t. $\langle p', q' \rangle \in R$

Two states p, q are *bisimilar* ($p \sim q$) if there exists a bisimulation R such that $\langle p, q \rangle \in R$.

This definition corresponds to the circular definition below that more clearly shows that two systems are bisimilar (observationally equivalent) if they can perform the same action and reach bisimilar states. This

recursive definition can be solved with the usual fixed points techniques.

Two states $p, q \in Q$ are *bisimilar*, written $p \sim q$, if and only if for each $\mu \in A_\tau$:

1. if $p \xrightarrow{\mu} p'$ then $q \xrightarrow{\mu} q'$ for some q' such that $p' \sim q'$;
2. if $q \xrightarrow{\mu} q'$ then $p \xrightarrow{\mu} p'$ for some p' such that $p' \sim q'$.

Bisimilarity distinguishes all machines of Fig. 1. This is because the basic idea behind bisimilarity is that two states are considered equivalent if by performing the same sequences of actions from these states it is possible to reach equivalent states. It is not difficult to see that bisimilarity distinguishes the first and the second machine of Fig. 1 because after receiving two coins (*coin*₁ and *coin*₂) the first machine still offers the user the possibility of choosing between having *coffee* or *tea* while the second does not. To see that also the second and the third machine are distinguished, it is sufficient to consider only the states reachable after just inserting *coin*₁ because already after this insertion the user loses his control of the third machine. Indeed, there is no way for this machine to reach a state bisimilar to the one that the second machine reaches after accepting *coin*₁.

Testing Equivalence

The formulation of bisimilarity is mathematically very elegant and has received much attention also in other fields of computer science [10]. However, some researchers do consider it too discriminating: two processes may be deemed unrelated even though there is no practical way of ascertaining it. As an example, consider the two rightmost vending machines of Fig. 1. They are not bisimilar because after inserting the first coin in one case there is still the illusion of having the possibility of choosing what to drink. Nevertheless, a customer would not be able to appreciate their differences since there is no possibility of deciding what to drink with both machines.

Testing equivalence has been proposed [4] (see also [5]) as an alternative to bisimilarity; it takes to the extreme the claim that when defining behavioral equivalences, one does not want to distinguish between systems that cannot be taken apart by external observers and bases the definition of the equivalences

on the notions of *observers*, *observations*, and *successful observations*. Equivalences are defined that consider equivalent those systems that satisfy (lead to successful observations by) the same sets of observers. An *observer* is an LTS with actions in $A_{\tau,w} \triangleq A_\tau \cup \{w\}$, with $w \notin A$. To determine whether a state q satisfies an observer with initial state o , the set $OBS(q, o)$ of all *computations* from $\langle q, o \rangle$ is considered.

Given an LTS $\langle Q, A_\tau, \xrightarrow{\mu} \rangle$ and an observer $\langle O, A_{\tau,w}, \xrightarrow{\mu} \rangle$, and a state $q \in Q$ and the initial state $o \in O$, an *observation* c from $\langle q, o \rangle$ is a maximal sequence of pairs $\langle q_i, o_i \rangle$, such that $\langle q_0, o_0 \rangle = \langle q, o \rangle$. The transition $\langle q_i, o_i \rangle \xrightarrow{\mu} \langle q_{i+1}, o_{i+1} \rangle$ can be proved using the following inference rule:

$$\frac{E \xrightarrow{\mu} E' \quad F \xrightarrow{\mu} F'}{\langle E, F \rangle \xrightarrow{\mu} \langle E', F' \rangle} \mu \in A_\tau$$

An observation from $\langle q, o \rangle$ is *successful* if it contains a configuration $\langle q_n, o_n \rangle \in c$, with $n \geq 0$, such that $o_n \xrightarrow{w} o$ for some o .

When analyzing the outcome of observations, one has to take into account that, due to nondeterminism, a process satisfies an observer **sometimes** or a process satisfies an observer **always**. This leads to the following definitions:

1. q **MAY SATISFY** o if there exists an observation from $\langle q, o \rangle$ that is successful.
2. q **MUST SATISFY** o if all observations from $\langle q, o \rangle$ are successful.

These notions can be used to define *may*, *must* and, *testing* equivalence.

May equivalence: p is *may* equivalent to q ($p \simeq_m q$) if, for all possible observers o :

p **MAY SATISFY** o if and only if q **MAY SATISFY** o ;

Must equivalence: p is *must* equivalent to q ($p \simeq_M q$) if, for all possible observers o :

p **MUST SATISFY** o if and only if q **MUST SATISFY** o .

Testing equivalence: p is *testing* equivalent to q ($p \simeq_{test} q$) if $p \simeq_m q$ and $p \simeq_M q$.

The three vending machines of Fig. 1 are may equivalent, but only the two rightmost ones are must equivalent and testing equivalent. Indeed, in most cases must equivalence implies may equivalence, and thus in most cases must and testing do coincide. The two leftmost

machines are not must equivalent because one after receiving the two coins the machine cannot refuse to (must) deliver the drink chosen by the customer while the other can.

May and must equivalences have nice alternative characterizations. It has been shown that may equivalence coincides with traces equivalence and that must equivalence coincides with *failures equivalence*, another well-studied relation that is inspired by traces equivalence but takes into account the possible interactions (*failures*) after each trace and is thus more discriminative than trace equivalence [7]. Failures equivalence relies on pairs of the form $\langle s, F \rangle$, where s is a trace and F is a set of labels. Intuitively, $\langle s, F \rangle$ is a failure for a process if it can perform the sequence of actions s to evolve into a state from which no action in F is possible. This equivalence can be formulated on LTS as follows:

Failures equivalence: Two states p and q are *failures-equivalent*, written $p \simeq_F q$, if and only if they possess the same failures, i.e., if for any $s \in A_\tau^*$ and for any $F \subseteq A_\tau$:

1. $p \xrightarrow{s} p'$ and $Init(p') \cap F = \emptyset$ implies $q \xrightarrow{s} q'$ for some q' and $Init(q') \cap F = \emptyset$
2. $q \xrightarrow{s} q'$ and $Init(q') \cap F = \emptyset$ implies $p \xrightarrow{s} p'$ for some p' and $Init(p') \cap F = \emptyset$

where $Init(q)$ represents the immediate actions of state q .

Hierarchy of Equivalences

The equivalences considered above can be precisely related (see [3] for a first study). Their relationships over the class of finite transition systems with only visible actions are summarized by the figure below, where the upward arrow indicates containment of the induced relations over states and \equiv indicates coincidence.

$$\begin{array}{c} \simeq_F \equiv \simeq_M \\ \uparrow \\ \simeq_T \equiv \simeq_m \end{array}$$

Overall, the figure states that may testing gives rise to a relation (\simeq_m) that coincides with traces equivalence, while must testing gives rise to a relation \simeq_M

that coincides with failures equivalence. For the considered class of systems, it also holds that must and testing equivalence \simeq_{test} do coincide. Thus, bisimilarity implies testing equivalence that in turn implies traces equivalence.

Weak Variants of the Equivalences

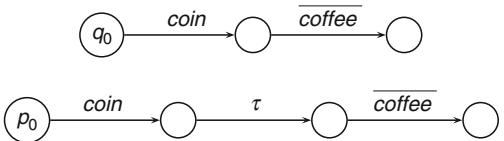
When considering abstract versions of systems making use of invisible actions, it turns out that all equivalences considered above are too discriminating. Indeed, traces, testing/failures, and observation equivalence would distinguish the two machines of Fig. 2 that, nevertheless, exhibit similar observable behaviors: get a coin and deliver a coffee. The second one can be obtained, e.g., from the term

$$coin.grinding.\overline{coffee}.nil$$

by hiding the *grinding* action that is irrelevant for the customer.

Because of this overdiscrimination, *weak variants* of the equivalences have been defined that permit ignoring (to different extents) internal actions when considering the behavior of systems. The key step of their definition is the introduction of a new transition relation that ignores silent actions. Thus, $q \xrightarrow{a} q'$ denotes that q reduces to q' by performing the visible action a possibly preceded and followed by any number (also 0) of invisible actions (τ). The transition $q \xrightarrow{s} q'$, instead, denotes that q reduces to q' by performing the sequence s of visible actions, each of which can be preceded and followed by τ -actions, while $\xrightarrow{\epsilon}$ indicates that only τ -actions, possibly none, are performed.

Weak traces equivalence The weak variant of traces equivalence is obtained by simply replacing the transitions $p \xrightarrow{s} p'$ above with the observable transitions $p \xrightarrow{s} p'$.



Behavioral Equivalences. Fig. 2 Weakly equivalent vending machines

Two states p and q are *weak traces equivalent* ($p \approx_T q$) if for any $s \in A^*$:

1. $p \xrightarrow{s} p'$ implies $q \xrightarrow{s} q'$ for some q'
2. $q \xrightarrow{s} q'$ implies $p \xrightarrow{s} p'$ for some p'

Weak testing equivalence To define the weak variants of may, must, and testing equivalences (denoted by \approx_m , \approx_M , \approx_{test} respectively), it suffices to change experiments so that processes and observers can freely perform silent actions. To this purpose, one only needs to change the inference rule of the observation step: $\langle q_i, o_i \rangle \xrightarrow{\mu} \langle q_{i+1}, o_{i+1} \rangle$ that can now be proved using:

$$\begin{array}{c} E \xrightarrow{\tau} E' \\ \hline \langle E, F \rangle \xrightarrow{\tau} \langle E', F \rangle \end{array} \quad \begin{array}{c} F \xrightarrow{\tau} F' \\ \hline \langle E, F \rangle \xrightarrow{\tau} \langle E, F' \rangle \end{array}$$

$$\frac{E \xrightarrow{a} E' \quad F \xrightarrow{a} F'}{\langle E, F \rangle \xrightarrow{a} \langle E', F' \rangle} \quad a \in A$$

To define, instead, *weak failures equivalence*, it suffices to replace $p \xrightarrow{s} p'$ with $p \overset{s}{\Rightarrow} p'$ in the definition of its strong variant. It holds that weak traces equivalence coincides with weak may equivalence, and that weak failures equivalence \approx_F coincides with weak must equivalence.

Weak bisimulation equivalence For defining weak observational equivalence, a new notion of (weak) bisimulation is defined that again assigns a special role to τ 's. To avoid having four items, the definition below requires that the candidate bisimulation relations be symmetric:

A symmetric relation $R \subseteq Q \times Q$ is a *weak bisimulation* if, for any pair of states p and q such that $\langle p, q \rangle \in R$, the following holds:

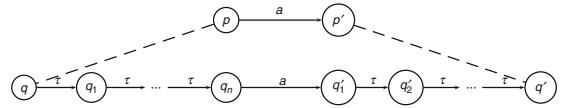
- For all $a \in A$ and $p' \in Q$, if $p \xrightarrow{a} p'$ then $q \xrightarrow{a} q'$ for some $q' \in Q$ s.t. $\langle p', q' \rangle \in R$
- For all $p' \in Q$, if $p \xrightarrow{\tau} p'$ then $q \xrightarrow{\epsilon} q'$ for some $q' \in Q$ s.t. $\langle p', q' \rangle \in R$

Two states p, q are *weakly bisimilar* ($p \approx q$) if there exists a weak bisimulation R such that $\langle p, q \rangle \in R$.

The figure below describes the intuition behind weak bisimilarity. In order to consider two states, say p and q , equivalent, it is necessary that for each visible action performed by one of them the other has to have the possibility of performing the same visible

action possibly preceded and followed by any number of invisible actions.

B



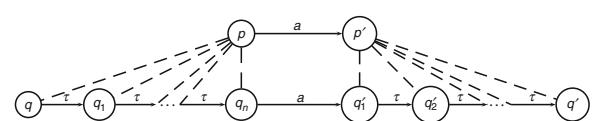
Branching bisimulation equivalence An alternative to weak bisimulation has also been proposed that considers those τ -actions important that appear in branching points of systems descriptions: only silent actions that do not eliminate possible interaction with external observers are ignored.

A symmetric relation $R \subseteq Q \times Q$ is a *branching bisimulation* if, for any pair of states p and q such that $\langle p, q \rangle \in R$, if $p \xrightarrow{\mu} p'$, with $\mu \in A_\tau$ and $p' \in Q$, at least one of the following conditions holds:

- $\mu = \tau$ and $\langle p', q \rangle \in R$
- $q \xrightarrow{\epsilon} q'' \xrightarrow{\mu} q'$ for some $q', q'' \in Q$ such that $\langle p, q'' \rangle \in R$ and $\langle p', q' \rangle \in R$

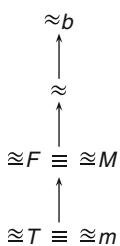
Two states p, q are *branching bisimilar* ($p \approx_b q$) if there exists a branching bisimulation R such that $\langle p, q \rangle \in R$.

The figure below describes the intuition behind branching bisimilarity; it corresponds to the definition above although it might appear, at first glance, more demanding. In order to consider two states, say p and q , equivalent, it is necessary, like for weak bisimilarity, that for each visible action performed by one of them the other has to have the possibility of performing the same visible action possibly preceded and followed by any number of invisible actions. Branching bisimilarity, however, imposes the additional requirement that all performed internal actions are not used to change equivalent class. Thus, all states reached via τ 's before performing the visible action are required to be equivalent to p , while all states reached via τ 's after performing the visible action are required to be equivalent to p' .



Hierarchy of Weak Equivalences

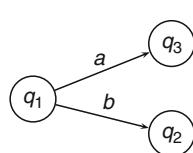
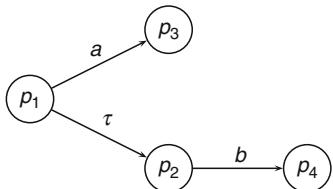
Like for the strong case, also weak equivalences can be clearly related. Their relationships over the class of finite transition systems with invisible actions, but without τ -loops (so-called *non-divergent* or *strongly convergent* LTSs) are summarized by the figure below, where the upward arrow indicates containment of the induced relations over states.



Thus, over *strongly convergent* LTSs with silent actions, branching bisimilarity implies weak bisimilarity, and this implies testing and failures equivalences; and these imply traces equivalence.

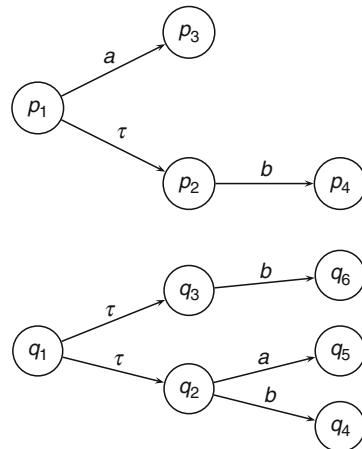
A number of counterexamples can be provided to show that the implications of the figure above are proper and thus that the converse does not hold.

The two LTSs reported below are weak traces equivalent and weakly may equivalent, but are distinguished by all the other equivalences.



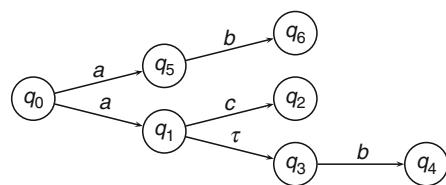
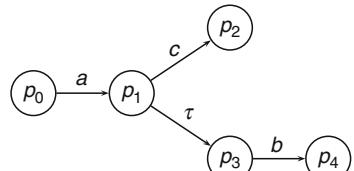
Indeed, they can perform exactly the same weak traces, but while the former can silently reach a state in which an a -action can be refused the second cannot.

The next two LTSs are equated by weak trace and weak must equivalences, but are distinguished by weak bisimulation and branching bisimulation.



Both of them, after zero or more silent actions, can be either in a state where both actions a and b are possible or in a state in which only a b transition is possible. However, via a τ -action, the topmost system can reach a state that has no equivalent one in the bottom one, thus they are not weakly bisimilar.

The next two LTSs are instead equated by weak bisimilarity, and thus by weak trace and weak must equivalences, but are not branching bisimilar.



It is easy to see that from the states p_0 and q_0 , the same visible action is possible and bisimilar states can be reached. The two states p_0 and q_0 are instead not branching bisimilar because p_0 , in order to match the a action of q_0 to q_5 and reach a state equivalent to q_5 , needs to reach p_3 through p_1 , but these two states, connected by a τ -action, are not branching bisimilar.

It is worth concluding that the two LTSs of Fig. 2 are equated by all the considered weak equivalences.

Future Directions

The study on behavioral equivalences of transition systems is still continuing. LTSs are increasingly used as the basis for specifying and proving properties of reactive systems. For example, they are used in *model checking* as the model against which logical properties are checked. It is then important to be able to use minimal systems that are, nevertheless, equivalent to the original larger ones so that preservation of the checked properties is guaranteed. Thus, further research is expected on devising efficient algorithms for equivalence checking and on understanding more precisely the properties of systems that are preserved by the different equivalences.

Related Entries

- [Actors](#)
- [Bisimulation](#)
- [CSP \(Communicating Sequential Processes\)](#)
- [Pi-Calculus](#)
- [Process Algebras](#)

Bibliographic Notes and Further Reading

The theories of equivalences can be found in a number of books targeted to describing the different process algebras. The theory of bisimulation is introduced in [8], while failure and trace semantics are considered in [7] and [9]. The testing approach is presented in [5].

Moreover, interesting papers relating the different approaches are [3], the first paper to establish precise relationships between the many equivalences proposed in the literature, and the two papers by R. van Glabbeek: [11], considering systems with only visible actions, and [12], considering also systems with invisible actions. In his two companion papers, R. van Glabbeek provides a uniform, model-independent account of many of the equivalences proposed in the literature and proposes several motivating testing scenarios, phrased in terms of “button pushing experiments” on reactive machines to capture them.

Bisimulation and its relationships with modal logics is deeply studied in [6], while a deep study of its origins and its use in other areas of computer science is provided in [10]. Branching bisimulation was first introduced in [1], while the testing based equivalences

were introduced in [4]. Failure semantic was first introduced in [2].

Bibliography

1. Baeten JCM, Weijland WP (1984) Process algebra. Cambridge University Press, Cambridge
2. Brookes SD, Hoare CAR, Roscoe AW (1984) A theory of communicating sequential processes. J ACM 31(3):560–599
3. De Nicola R (1987) Extensional equivalences for transition systems. Acta Informatica 24(2):211–237
4. De Nicola R, Hennessy M (1984) Testing equivalences for processes. Theor Comput Sci 34:83–133
5. Hennessy M (1988) Algebraic theory of processes. The MIT Press, Cambridge
6. Hennessy M, Milner R (1985) Algebraic laws for nondeterminism and concurrency. J ACM 32(1):137–161
7. Hoare CAR (1985) Communicating sequential processes. Prentice-Hall, Englewood Cliffs
8. Milner R (1989) Communication and concurrency. Prentice-Hall, Upper Saddle River
9. Roscoe AW (1998) The theory and practice of concurrency. Prentice-Hall, Hertfordshire
10. Sangiorgi D (2009) On the origins of bisimulation and coinduction. ACM Trans Program Lang Syst 31(4):15.1–15.41
11. van Glabbeek RJ (2001) The linear time-branching time spectrum I: the semantics of concrete, sequential processes. In: Bergstra JA, Ponse A, Smolka SA (eds) Handbook of process algebra, Elsevier, Amsterdam, pp 3–99
12. van Glabbeek RJ (1993) The linear time-branching time spectrum II. In: Best E (ed) CONCUR ’93, 4th international conference on concurrency theory, Hildesheim, Germany, Lecture notes in computer science, vol 715. Springer-Verlag, Heidelberg, pp 66–81

Behavioral Relations

- [Behavioral Equivalences](#)

Benchmarks

JACK DONGARRA, PIOTR LUSZCZEK
University of Tennessee, Knoxville, TN, USA

Definition

Computer benchmarks are computer programs that form standard tests of the performance of a computer

and the software through which it is used. They are written to a particular programming model and implemented by specific software, which is the final arbiter as to what the programming model is. A benchmark is therefore testing a software interface to a computer, and not a particular type of computer architecture.

Discussion

The basic goal of performance modeling is to measure, predict, and understand the performance of a computer program or set of programs on a computer system. In other words, it transcends the measurement of basic architectural and system parameters and is meant to enhance the understanding of the performance behavior of full complex applications. However, the programs and codes used in different areas of science differ in a large number of features. Therefore the performance of full application codes cannot be characterized in a general way independent of the application and code used. The understanding of the performance characteristics is tightly bound to the specific computer program code used. Therefore the careful selection of an interesting program for analysis is the crucial first step in any more detailed and elaborate investigation of full application code performance. The applications of performance modeling are numerous, including evaluation of algorithms, optimization of code implementations, parallel library development, and comparison of system architectures, parallel system design, and procurement of new systems.

A number of projects such as Perfect, NPB, Park-Bench, HPC Challenge, and others have laid the groundwork for a new era in benchmarking and evaluating the performance of computers. The complexity of these machines requires a new level of detail in measurement and comprehension of the results. The quotation of a single number for any given advanced architecture is a disservice to manufacturers and users alike, for several reasons. First, there is a great variation in performance from one computation to another on a given machine; typically the variation may be one or two orders of magnitude, depending on the type of machine. Secondly, the ranking of similar machines often changes as one goes from one application to

another, so the best machine for circuit simulation may not be the best machine for computational fluid dynamics. Finally, the performance depends greatly on a combination of compiler characteristics and human efforts were expended on obtaining the results.

The conclusions drawn from a benchmark study of computer performance depend not only on the basic timing results obtained, but also on the way these are interpreted and converted into performance figures. The choice of the performance metric, may itself influence the conclusions. For example, is it desirable to have a computer that generates the most mega op per second (or has the highest Speedup), or the computer that solves the problem in the least time? It is now well known that high values of the first metrics do not necessarily imply the second property. This confusion can be avoided by choosing a more suitable metric that reflects solution time directly, for example, either the Temporal, Simulation, or Benchmark performance, defined below. This issue of the sensible choice of performance metric is becoming increasingly important with the advent of massively parallel computers which have the potential of very high Giga-op rates, but have much more limited potential for reducing solution time.

Given the time of execution T and the floating-point operation-count several different performance measures can be defined. Each metric has its own uses, and gives different information about the computer and algorithm used in the benchmark. It is important therefore to distinguish the metrics with different names, symbols and units, and to understand clearly the difference between them. Much confusion and wasted work can arise from optimizing a benchmark with respect to an inappropriate metric. If the performance of different algorithms for the solution of the same problem needs to be compared, then the correct performance metric to use is the Temporal Performance which is defined as the inverse of the execution time

$$R_T = 1/T.$$

A special case of temporal performance occurs for simulation programs in which the benchmark problem is defined as the simulation of a certain period of physical time, rather than a certain number of timesteps. In this case, the term “simulation performance” is used,

and it is measured in units such as simulated days per day (written sim-d/d or 'd'/d) in weather forecasting, where the apostrophe is used to indicate "simulated" or simulated pico-seconds per second (written sim-ps/s or 'ps'/s) in electronic device simulation. It is important to use simulation performance rather than timestep/s for comparing different simulation algorithms which may require different sizes of timestep for the same accuracy (e.g., an implicit scheme that can use a large timestep, compared with an explicit scheme that requires a much smaller step). In order to compare the performance of a computer on one benchmark with its performance on another, account must be taken of the different amounts of work (measured in op) that the different problems require for their solution. The benchmark performance is defined as the ratio of the floating-point operation-count and the execution time

$$R_B = F_B/T.$$

The units of benchmark performance are Giga-op/s (benchmark name), where the name of the benchmark is included in parentheses to emphasize that the performance may depend strongly on the problem being solved, and to emphasize that the values are based on the nominal benchmark op-count. In other contexts such performance figures would probably be quoted as examples of the so-called sustained performance of a computer. For comparing the observed performance with the theoretical capabilities of the computer hardware, the actual number of floating-point operations performed F_H is computed, and from it the actual hardware performance

$$R_H = F_H/T.$$

Parallel speedup is a popular metric that has been used for many years in the study of parallel computer performance. Speedup is usually defined as the ratio of execution time of one-processor T_1 and execution time on p -processors T_p .

One factor that has hindered the use of full application codes for benchmarking parallel computers in the past is that such codes are difficult to parallelize and to port between target architectures. In addition, full application codes that have been successfully parallelized are

often proprietary, and/or subject to distribution restrictions. To minimize the negative impact of these factors, the use of compact applications was proposed in many benchmarking efforts. Compact applications are typical of those found in research environments (as opposed to production or engineering environments), and usually consist of up to a few thousand lines of source code. Compact applications are distinct from kernel applications since they are capable of producing scientifically useful results. In many cases, compact applications are made up of several kernels, interspersed with data movements and I/O operations between the kernels.

Any of the performance metrics, R , can be described with a two-parameter Amdahl saturation, for a fixed problem size as a function of number of processors p ,

$$R = R_\infty / (1 + p_{1/2}/p)$$

where R_∞ is the saturation performance approached as $p \rightarrow \infty$ and $p_{1/2}$ is the number of processors required to reach half the saturation performance. This universal Amdahl curve [1, 2] could be matched against the actual performance curves by changing values of the two parameters (R_∞ , $p_{1/2}$).

Related Entries

- [HPC Challenge Benchmark](#)
- [LINPACK Benchmark](#)
- [Livermore Loops](#)
- [TOP500](#)

Bibliography

1. Hockney RW (1992) A framework for benchmark analysis. *Supercomputer* 48(IX-2):9–22
2. Addison C, Allwright J, Binsted N, Bishop N, Carpenter B, Dalloz P, Gee D, Getov V, Hey A, Hockney R, Lemke M, Merlin J, Pinches M, Scott C, Wolton I (1993) The genesis distributed-memory benchmarks. Part I: methodology and general relativity benchmark with results for the SUPRENUM computer. *Concurrency: Practice and Experience* 5(1):1–22

Beowulf Clusters

- [Clusters](#)

Beowulf-Class Clusters

►Clusters

Bernstein's Conditions

PAUL FEAUTRIER

Ecole Normale Supérieure de Lyon, Lyon, France

Definition

Bersntein's conditions [4] are a simple test for deciding if statements or operations can be interchanged without modifying the program results. The test applies to operations which read and write memory at well defined addresses. If u is an operation, let $\mathcal{M}(u)$ be the set of (addresses of) the memory cells it modifies, and $\mathcal{R}(u)$ the set of cells it reads. Operations u and v can be reordered if:

$$\mathcal{M}(u) \cap \mathcal{M}(v) = \mathcal{M}(u) \cap \mathcal{R}(v) = \mathcal{R}(u) \cap \mathcal{M}(v) = \emptyset \quad (1)$$

If these conditions are met, one says that u and v commute or are independent.

Note that in most languages, each operation writes at most one memory cell: $W(u)$ is a singleton. However, there are exceptions: multiple and parallel assignments, vector assignments among others.

The importance of this result stems from the fact that most program optimizations consist – or at least, involve – moving operations around. For instance, to improve cache performance, one must move all uses of a datum as near as possible to its definition. In parallel programming, if u and v are assigned to different threads or processors, their order of execution may be unpredictable, due to arbitrary decisions of a scheduler or to the presence of competing processes. In this case, if Bernstein's conditions are not met, u and v must be kept in the same thread.

Checking Bernstein's conditions is easy for operations accessing scalars (but beware of aliases), is more difficult for array accesses, and is almost impossible for pointer dereferencing. See the ►Dependences entry for an in-depth discussion of this question.

Discussion

Notations and Conventions

In this essay, a program is represented as a sequence of operations, i.e., of instances of high level statements or machine instructions. Such a sequence is called a *trace*. Each operation has a unique name, u , and a text $T(u)$, usually specified as a (high-level language) statement. There are many schemes for naming operations: for polyhedral programs, one may use integer vectors, and operations are executed in lexicographic order. For flowcharts programs, one may use words of a regular language to name operations, and if the program has function calls, words of a context-free language [2]. In the last two cases, u is executed before v iff u is a prefix of v . In what follows, $u < v$ is a shorthand for “ u is executed before v .” For sequential programs, $<$ is a well-founded total order: there is no infinite chain $x_0, x_1, \dots, x_i, \dots$ such that $x_{i+1} < x_i$. This is equivalent to stipulating that a program execution has a beginning, but may not have an end.

All operations will be assumed deterministic: the state of memory after execution of u depends only on $T(u)$ and on the previous state of memory.

For *static control programs*, one can enumerate the unique trace – or at least describe it – once and for all. One can also consider static control program *families*, where the trace depends on a few parameters which are known at program start time. Lastly, one can consider static control parts of programs or SCoPs. Most of this essay will consider only static control programs.

When applying Bernstein's conditions, one usually considers a reference trace, which comes from the original program, and a candidate trace, which is the result of some optimization or parallelization. The problem is to decide whether the two traces are equivalent, in a sense to be discussed later. Since program equivalence is in general undecidable, one has to restrict the set of admissible transformations. Bernstein's conditions are specially useful for dealing with operation reordering.

Commutativity

To prove that Bernstein's conditions are sufficient for commutativity, one needs the following facts:

- When an operation u is executed, the only memory cells which may be modified are those whose addresses are in $\mathcal{M}(u)$

- The values stored in $\mathcal{M}(u)$ depend only on u and on the values read from $\mathcal{R}(u)$.

▶ Consider two operations u and v which satisfy (Eq. 1). Assume that u is executed first. When v is executed later, it finds in $\mathcal{R}(v)$ the same values as if it were executed first, since $\mathcal{M}(u)$ and $\mathcal{R}(v)$ are disjoint. Hence, the values stored in $\mathcal{M}(v)$ are the same, and they do not overwrite the values stored by u , since $\mathcal{M}(u)$ and $\mathcal{M}(v)$ are disjoint. The same reasoning applies if v is executed first.

The fact that u and v do not meet Bernstein's conditions is written $u \perp v$ to indicate that u and v cannot be executed in parallel.

Atomicity

When dealing with parallel programs, commutativity is not enough for correctness. Consider for instance two operations u and v with $\mathcal{T}(u) = [x = x + 1]$ and $\mathcal{T}(v) = [x = x + 2]$. These two operations commute, since their sequential execution in whatever order is equivalent to a unique operation w such that $\mathcal{T}(w) = [x = x + 3]$. However, each one is compiled into a sequence of more elementary machine instructions, which when executed in parallel, may result in x being increased by 1 or 2 or 3 (see Fig. 1, where $r1$ and $r2$ are processor registers).

Observe that these two operations *do not* satisfy Bernstein's conditions. In contrast, operations that satisfy Bernstein's conditions do not need to be protected by critical sections when run in parallel. The reason is that neither operation modifies the input of the other, and that they write in distinct memory cells. Hence, the stored values do not depend on the order in which the writes are interleaved.

$x = 0$	$x = 0$	$x = 0$	$x = 0$
$r1 = x$	$--$	$r1 = x$	$--$
$--$	$r2 = x$	$r1 += 1$	$--$
$r1 += 1$	$--$	$x = r1$	$r2 = x$
$--$	$r2 += 2$	$--$	$r1 += 1$
$x = r1$	$--$	$r2 = x$	$--$
$--$	$x = r2$	$r2 += 2$	$r2 += 2$
$--$	$--$	$x = r2$	$x = r2$
$--$	$--$	$x = r2$	$x = r1$
$x = 2$	$x = 3$	$x = 3$	$x = 1$
P #1	P #2	P #1	P #2

Bernstein's Conditions. Fig. 1 Several possible interleaves of $x = x + 1$ and $x = x + 2$

Legality

Here, the question is to decide whether a candidate trace is equivalent to a reference trace, where the two traces contains exactly the same operations. There are two possibilities for deciding equivalence. Firstly, if the traces are finite, one may examine the state of memory after their termination. There is equivalence if these two states are identical. Another possibility is to construct the *history* of each memory cell. This is a list of values ordered in time. A new value is appended to the history of x each time an operation u such that $x \in \mathcal{M}(u)$ is executed. Two traces are equivalent if all cells have the same history. This is clearly a stronger criterion than equality of the final memory; it has the advantage of being applicable both to terminating programs and to non-terminating systems. The histories are especially simple when a trace has the *single assignment property*: there is only one operation that writes into x . In that case, each history has only one element.

Terminating Programs

A terminating program is specified by a finite list of operations, $[u_1, \dots, u_n]$, in order of sequential execution. There is a dependence relation $u_i \rightarrow u_j$ iff $i < j$ and $u_i \perp u_j$.

All reorderings of the u : $[v_1, \dots, v_n]$ such that the execution order of dependent operations is not modified:

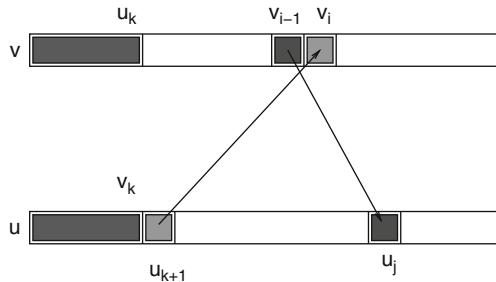
$$u_i \rightarrow u_j, u_i = v_{i'}, u_j = v_{j'} \Rightarrow i' < j'$$

are legal.

- ▶ The proof is by a double induction. Let k be the length of the common prefix of the two programs:

$$u_i = v_i, i = 1, k.$$

Note that k may be null. The element u_{k+1} occurs somewhere among the v , at position $i > k$. The element v_{i-1} occurs among the u at position $j > k+1$ (see Fig. 2). It follows that $u_{k+1} = v_i$ and $u_j = v_{i-1}$ are ordered differently in the two programs, and hence must satisfy Bernstein's condition. v_{i-1} and v_i can therefore be exchanged without modifying the result of the reordered program. Continuing in this way, v_i can be brought in position $k+1$, which means that the common prefix has been extended one position to the right. This process can be continued until the length of the prefix is n . The two programs are now identical, and the final result of the candidate trace has not been modified.



Bernstein's Conditions. Fig. 2 The commutation Lemma

The property which has just been proved is crucial for program optimization, since it gives a simple test for the legality of statement motion, but what is its import for parallel programming?

The point is that when parallelizing a program, its operations are distributed among several processors or among several threads. Most parallel architectures do not try to combine simultaneous writes to the same memory cell, which are arbitrarily ordered by the bus arbiter or a similar device. It follows that if one is only interested in the final result, each parallel execution is equivalent to some interleave of the several threads of the program. Taking care that operations which do not satisfy Bernstein's condition are executed in the order specified by the original sequential program guarantees deterministic execution and equivalence to the sequential program.

Single Assignment Programs

A trace is in single assignment form if, for each memory cell x , there is one and only one operation u such that $x \in \mathcal{M}(u)$. Any trace can be converted to (dynamic) single assignment form – at least in principle – by the following method.

Let A be an (associative) array indexed by the operation names. Assuming that all $\mathcal{M}(u)$ are singletons, operation u now writes into $A[u]$ instead of $\mathcal{M}(u)$. The source of cell x at u , noted $\sigma(x, u)$, is defined as:

- $x \in \mathcal{M}(\sigma(x, u))$
- $\sigma(x, u) < u$
- there is no v such that $\sigma(x, u) < v < u$ and $x \in \mathcal{M}(v)$

In words, $\sigma(x, u)$ is the last write to x that precedes u . Now, in the text of u , replace all occurrences of $y \in \mathcal{R}(u)$

by $A[\sigma(y, u)]$. That the new trace has the single assignment property is clear. It is equivalent to the reference trace in the following sense: for each cell x , construct a history by appending the value of $A[u]$ each time an operation u such that $x \in \mathcal{M}(u)$ is executed. Then the histories of a cell in the reference trace and in the single assignment trace are identical.

- ▶ Let us say that an operation u has a discrepancy for x if the value assigned to x by u in the reference trace is different from the value of $A[u]$ in the single assignment trace. Let u_0 be the earliest such operation. Since all operations are assumed deterministic, this means that there is a cell $y \in \mathcal{R}(u_0)$ whose value is different from $A[\sigma(y, u_0)]$. Hence $\sigma(y, u_0) < u_0$ also has a discrepancy, a contradiction.

Single assignment programs (SAP) where first proposed by Tesler and Enea [9] as a tool for parallel programming. In a SAP, the sets $\mathcal{M}(u) \cap \mathcal{M}(v)$ are always empty, and if there is a non-empty $\mathcal{R}(u) \cap \mathcal{M}(v)$ where $u < v$, it means that some variable is read before being assigned, a programming error. Some authors [3] then noticed that a single assignment program is a collection of algebraic equations, which simplifies the construction of correctness proofs.

Non-Terminating Systems

The reader may have noticed that the above legality proof depends on the finiteness of the program trace. What happens when one wants to build a non-terminating parallel system, as found for instance in signal processing applications or operating systems? For assessing the correctness of a transformation, one cannot observe the final result, which does not exist. Beside, one clearly needs some fairness hypothesis: it would not do to execute all even numbered operations, *ad infinitum*, and then to execute all odd numbered operations, even if Bernstein's conditions would allow it. The needed property is that for all operations u in the reference trace, there is a finite integer n such that u is the n -th operation in the candidate trace.

Consider first the case of two single assignment traces, one of which is the reference trace, the other having been reordered while respecting dependences. Let u be an operation. By the fairness hypothesis, u is present in both traces. Assume that the values written in $A[u]$ by the two traces are distinct. As above, one can find

an operation v such that $A[v]$ is read by u , $A[v]$ has different values in the two traces, and $v < u$ in the two traces. One can iterate this process indefinitely, which contradicts the well-foundedness of $<$.

Consider now two ordinary traces. After conversion to single assignment, one obtain the same values for the $A[u]$. If one extract an history for each cell x as above, one obtain two identical sequence of values, since operations that write to x are in dependence and hence are ordered in the same direction in the two traces.

Observe that this proof applies also to terminating traces. If the cells of two terminating traces have identical histories, it obviously follows that the final memory states are identical. On the other hand, the proof for terminating traces applies also, in a sequential context, to operations which commutes without satisfying Bernstein's conditions.

Dynamic Control Programs

The presence of tests whose outcome cannot be predicted at compile time greatly complicates program analysis. The simplest case is that of well structured programs, which uses only the **if then else** construct. For such programs, a simple syntactical analysis allows the compiler to identify all tests which have an influence on the execution of each operation. One has to take into account three new phenomena:

- A test is an operation in itself, which has a set of read cells, and perhaps a set of modified cells if the source language allows side effects
- An operation cannot be executed before the outcomes of all controlling tests are known
- No dependence exists for two operations which belong to opposite branches of a test

A simple solution, known as *if-conversion* [1], can be used to solve all three problems at once. Each test: **if(e) then ... else ...** is replaced by a new operation $\mathbf{b} = \mathbf{e}$; where b is a fresh boolean variable. Each operation in the range of the test is guarded by b or $\neg b$, depending on whether the operation is on the **then** or **else** branch of the test. In the case of nested tests, this transformation is applied recursively; the result is that each operation is guarded by a conjunction of the b 's or their complements. Bernstein's conditions are then applied to the resulting trace, the b variables being included in the read and modified sets as necessary. This insures that the test

is not executed too early, and since there is a dependence from a test to each enclosed operation, that no operation is executed before the tests results are known. One must take care not to compute dependences between operations u and v which have incompatible guards g_u and g_v such that $g_u \wedge g_v = \text{false}$.

The case of **while** loops is more complex. Firstly, the construction **while(true)** is the simplest way of writing a non terminating program, whose analysis has been discussed above. Anything that follows an infinite loop is dead code, and no analysis is needed for it. Consider now a terminating loop:

```
while (p) do S;
```

The several executions of the continuation predicate, p , must be considered as operations. Strictly speaking, one cannot execute an instance of S before the corresponding instance of p , since if the result of p is false, S is not executed. On the other hand, there must be a dependence from S to p , since otherwise the loop would not terminate. Hence, a **while** loop must be executed sequentially. The only way out is to run the loop *speculatively*, i.e., to execute instances of the loop body before knowing the outcome of the continuation predicate, but this method is beyond the scope of this essay.

Related Entries

- [Dependences](#)
- [Polyhedron Model](#)

Bibliographic Notes and Further Reading

See Allen and Kennedy's book [7] for many uses of the concept of dependence in program optimization.

For more information on the transformation to Single Assignment form, see [5] or [6]. For the use of Single Assignment Programs for hardware synthesis, see [8] or [10].

Bibliography

1. Allen JR, Kennedy K, Porterfield C, Warren J (1983) Conversion of control dependence to data dependence. In: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on principles of programming languages, POPL '83, ACM, New York, pp 177–189
2. Amiranoff P, Cohen A, Feautrier P (2006) Beyond iteration vectors: instancewise relational abstract domains. In: Static analysis symposium (SAS '06), Seoul, August 2006

3. Arsac J (1977) *La construction de programmes structurés*. Dunod, Paris
4. Bernstein AJ (1966) Analysis of programs for parallel processing. *IEEE Trans Electron Comput EC-15*:757–762
5. Feautrier P (1991) Dataflow analysis of scalar and array references. *Int J Parallel Program* 20(1):23–53
6. Feautrier P (2001) Array dataflow analysis. In: Pande S, Agrawal D (eds) *Compiler optimizations for scalable parallel systems*. Lecture notes in computer science, vol 1808, chapter 6. Springer, Berlin, pp 173–216
7. Kennedy K, Allen R (2001) Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufman, San Francisco
8. Leverage H, Mauras C, Quinton P (1991) The alpha language and its use for the design of systolic arrays. *J VLSI Signal Process* 3:173–182
9. Tesler LG, Enea HJ (1968) A language design for concurrent processes. In: AFIPS SJCC 32, Thomson Book Co., pp 403–408
10. Verdoolaege S, Nikolov H, Stefanov T (2006) Improved derivation of process networks. In: Digest of the 4th workshop on optimization for DSP and embedded systems, New York, March 2006, pp 1–10

is punctuated by the occasional development of new technologies in the field which generally created new types of data acquisition (such as microarrays to capture gene expression developed in the mid-1990s) or more rapid acquisition of data (such as the development of next-generation sequencing technologies in the mid-2000s). Generally, these developments have ushered in new avenues of bioinformatics research due to new applications enabled by novel data sources, or increases in the scale of data that need to be archived and analyzed, or new applications that come within reach due to improved scales and efficiencies. For example, the rapid adoption of microarray technologies for measuring gene expressions ushered in the era of systems biology; the relentless increases in cost efficiencies in sequencing enabled genome sequencing for many species, which then formed the foundation for the field of comparative genomics.

Thanks to the aforementioned advances and our continually improving knowledge of how biological systems are designed and operate, bioinformatics developed into a broad field with several well-defined sub-fields of specialization – computational genomics, comparative genomics, metagenomics, phylogenetics, systems biology, structural biology, etc. Several entries in this encyclopedia are designed along the lines of such subfields whenever a sufficient body of work exists in development of parallel methods in the area. This entry contains general remarks about the field of parallel computational biology; the readers are referred to the related entries for an in-depth discussion and appropriate references for specific topics. An alternative view to classifying bioinformatics research relies on the organisms of study – (1) microbial organisms, (2) plants, and (3) humans/animals. Even though the underlying bioinformatics techniques are generally applicable across organisms, the key target applications tend to vary based on this organismal classification. In studying microbial organisms, a key goal is the ability to engineer them for increased production of certain products or to meet certain environmental objectives. In agricultural biotechnology, key challenges being pursued include increasing yields, increasing nutritional content, developing robust crops and biofuel production. On the other hand, much of the research on humans is driven by medical concerns and understanding and treating complex diseases.

Bioinformatics

SRINIVAS ALURU
Iowa State University, Ames, IA, USA
Indian Institute of Technology Bombay, Mumbai, India

Synonyms

Computational biology

Definition

Bioinformatics and/or computational biology is broadly defined as the development and application of informatics techniques for solving problems arising in biological sciences.

Discussion

The terms “Bioinformatics” and “Computational Biology” are broadly used to represent research in computational models, methods, databases, software, and analysis tools aimed at solving applications in the biological sciences. Although the origins of the field can be traced as far back as 1970, the field exploded in prominence during the 1990s with the conception and execution of the human genome project, and such explosive growth continues to this date. This long time line

Perhaps the oldest studied problem in computational biology is that of molecular dynamics, originally studied in computational chemistry but increasingly being applied to the study of protein structures in biology. Outside of this classical area, research in parallel computational biology can be traced back to the development of parallel sequence alignment algorithms in the late 1980s. One of the early applications where parallel bioinformatics methods proved crucial is that of genome assembly. At the human genome scale, this requires inferring a sequence by assembling tens of millions of randomly sampled fragments of it, which would take weeks to months if done serially. Albeit the initial use of parallelism only in phases where such parallelism is obvious, the resulting improvements in time-to-solution proved adequate, and subsequently motivated the development of more sophisticated parallel bioinformatics methods for genome assembly. Spurred by the enormous data sizes, problem complexities, and multiple scales of biological systems, interest in parallel computational biology continues to grow. However, the development of parallel methods in bioinformatics represents a mere fraction of the problems for which sequential solutions have been developed. In addition, progress in parallel computational biology has not been uniform across all subfields of computational biology. For example, there has been little parallel work in the field of comparative genomics, not counting trivial uses of parallelism such as in all-pairs alignments. In other fields such as systems biology, work in parallel methods is in its nascent stages with certain problem areas (such as network inference) targeted more than others. The encyclopedia entries reflect this development and cover the topical areas that reflect strides in parallel computational biology.

Going forward, there are compelling developments that favor growing prominence of parallel computing in the field of bioinformatics and computational biology. One such development is the creation of high-throughput second- and third-generation sequencing technologies. After more than three decades of sequencing DNA one fragment at a time, next-generation sequencing technologies permit the simultaneous sequencing of a large number of DNA fragments. With throughputs increasing at the rate of a factor of 10 per year, sequencers that were generating a few million DNA reads per experiment in 2008 are delivering

upward of 2 billion reads per experiment by early 2011. The high throughput sequencing data generated by these systems is impacting many subfields of computational biology, and the data deluge is severely straining the limits of what can be achieved by sequential methods. Next-generation sequencing is shifting individual investigators into terascale and bigger organizations into petascale, necessitating the development of parallel methods. Many other types of high-throughput instrumentation are becoming commonplace in biology, raising further complexities in massive scale, heterogenous data integration and multiscale modeling of biological systems. Emerging high performance computing paradigms such as Clouds and manycore GPU platforms are also providing impetus to the development of parallel bioinformatics methods.

Related Entries

- [Genome Assembly](#)
- [Homology to Sequence Alignment, From](#)
- [Phylogenetics](#)
- [Protein Docking](#)
- [Suffix Trees](#)
- [Systems Biology, Network Inference in](#)

Bibliographic Notes and Further Reading

Readers who wish to conduct an in-depth study of bioinformatics and computational biology may find the comprehensive handbook on computational biology a useful reference [1]. Works on development of parallel methods in computational biology can be found in the book chapter [2], the survey article [7], the first edited volume on this subject [8], and several journal special issues [3–6]. The annual *IEEE International Workshop on High Performance Computational Biology* held since 2002 provides the primary forum for research dissemination in this field and the readers may consult the proceedings (www.hicomb.org) for scoping the progress in the field and for further reference.

Bibliography

1. Aluru S (ed) (2005) Handbook of computational molecular biology. Chapman & Hall/CRC Computer and Information Science Series, Boca Raton
2. Aluru S, Amato N, Bader DA, Bhandarkar S, Kale L, Marinescu D, Samatova N (2006) Parallel computational biology.

- In: Heroux MA, Raghavan P, Simon HD (eds) Parallel Processing for Scientific Computing (Software, Environments and Tools). Society for Industrial and Applied Mathematics (SIAM), Philadelphia, pp 357–378
3. Aluru S, Bader DA (2003) Special issue on high performance computational biology. *J Parallel Distr Com* 63(7–8):671–673
 4. Aluru S, Bader DA (2004) Special issue on high performance computational biology. *Concurrency-Pract Ex* 16(9):817–821
 5. Aluru S, Bader DA (2008) Special issue on high performance computational biology. *Parallel Comput* 34(11):613–615
 6. Amato N, Aluru S, Bader DA (2006) Special issue on high performance computational biology. *IEEE Trans Parallel Distrib Syst* 17(8):737–739
 7. Bader DA (2004) Computational biology and high-performance computing. *Commun ACM* 47(11):34–41
 8. Zomaya AY (ed) (2006) Parallel computing for bioinformatics and computational biology: models, enabling technologies, and case studies. Wiley, Hoboken

Bisimilarity

► [Bisimulation](#)

Bisimulation

ROBERT J. VAN GLABEEK
NICTA, Sydney, Australia

The University of New South Wales, Sydney, Australia
Stanford University, Standford, CA, USA

Synonyms

[Bisimulation equivalence](#); [Bisimilarity](#)

Definition

Bisimulation equivalence is a semantic equivalence relation on labeled transition systems, which are used to represent distributed systems. It identifies systems with the same branching structure.

Discussion

Labeled Transition Systems

A labeled transition system consists of a collection of states and a collection of transitions between them. The transitions are labeled by actions from a given set A that happen when the transition is taken, and the states may

be labeled by predicates from a given set P that hold in that state.

Definition 1 Let A and P be sets (of *actions* and *predicates*, respectively).

A *labeled transition system* (LTS) over A and P is a triple $(S, \rightarrow, \models)$ with:

- S a class (of *states*).
- \rightarrow a collection of binary relations $\xrightarrow{a} \subseteq S \times S$ – one for every $a \in A$ – (the *transitions*), such that for all $s \in S$ the class $\{t \in S \mid s \xrightarrow{a} t\}$ is a set.
- $\models \subseteq S \times P$. $s \models p$ says that predicate $p \in P$ holds in state $s \in S$.

LTSs with A a singleton (i.e., with \rightarrow a single binary relation on S) are known as *Kripke structures*, the models of modal logic. General LTSs (with A arbitrary) are the Kripke models for polymodal logic. The name “labeled transition system” is employed in concurrency theory. There, the elements of S represent the systems one is interested in, and $s \xrightarrow{a} t$ means that system s can evolve into system t while performing the action a . This approach identifies states and systems: The states of a system s are the systems reachable from s by following the transitions. In this realm P is often taken to be empty, or it contains a single predicate \checkmark indicating successful termination.

Definition 2 A *process graph* over A and P is a tuple $g = (S, I, \rightarrow, \models)$ with $(S, \rightarrow, \models)$ an LTS over A and P in which S is a set, and $I \in S$.

Process graphs are used in concurrency theory to disambiguate between states and systems. A process graph $(S, I, \rightarrow, \models)$ represents a single system, with S the set of its states and I its initial state. In the context of an LTS $(S, \rightarrow, \models)$ two concurrent systems are modeled by two members of S ; in the context of process graphs, they are two different graphs. The *nondeterministic finite automata* used in *automata theory* are process graphs with a finite set of states over a finite alphabet A and a set P consisting of a single predicate denoting *acceptance*.

Bisimulation Equivalence

Bisimulation equivalence is defined on the states of a given LTS, or between different process graphs.

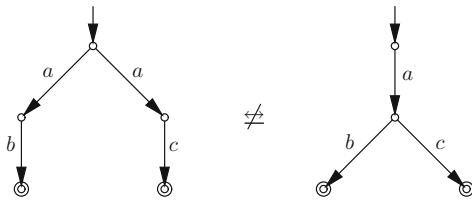
Definition 3 Let $(S, \rightarrow, \models)$ be an LTS over A and P . A *bisimulation* is a binary relation $R \subseteq S \times S$, satisfying:

- Λ if sRt then $s \models p \Leftrightarrow t \models p$ for all $p \in P$.
- Λ if sRt and $s \xrightarrow{a} s'$ with $a \in A$, then there exists a t' with $t \xrightarrow{a} t'$ and $s'Rt'$.
- Λ if sRt and $t \xrightarrow{a} t'$ with $a \in A$, then there exists an s' with $s \xrightarrow{a} s'$ and $s'Rt'$.

Two states $s, t \in S$ are *bisimilar*, denoted $s \leftrightarrow t$, if there exists a bisimulation R with sRt .

Bisimilarity turns out to be an equivalence relation on S , and is also called *bisimulation equivalence*.

Definition 4 Let $g = (S, I, \rightarrow, \models)$ and $h = (S', I', \rightarrow', \models')$ be process graphs over A and P . A *bisimulation* between g and h is a binary relation $R \subseteq S \times S'$, satisfying IRI' and the same three clauses as above. g and h are *bisimilar*, denoted $g \leftrightarrow h$, if there exists a bisimulation between them.



Example The two process graphs above (over $A = \{a, b, c\}$ and $P = \{\checkmark\}$), in which the initial states are indicated by short incoming arrows and the final states (the ones labeled with \checkmark) by double circles, are not bisimulation equivalent, even though in automata theory they accept the same language. The choice between b and c is made at a different moment (namely, before vs. after the a -action); that is, the two systems have a different *branching structure*. Bisimulation semantics distinguishes systems that differ in this manner.

Modal Logic

(Poly)modal logic is an extension of propositional logic with formulas $\langle a \rangle \varphi$, saying that it is possible to follow an a -transition after which the formula φ holds. Modal formulas are interpreted on the states of labeled transition systems. Two systems are bisimilar iff they satisfy the same infinitary modal formulas.

Definition 5 The language \mathcal{L} of *polymodal logic* over A and P is given by:

- $\top \in \mathcal{L}$
- $p \in \mathcal{L}$ for all $p \in P$
- if $\varphi, \psi \in \mathcal{L}$ for then $\varphi \wedge \psi \in \mathcal{L}$
- if $\varphi \in \mathcal{L}$ then $\neg\varphi \in \mathcal{L}$
- if $\varphi \in \mathcal{L}$ and $a \in A$ then $\langle a \rangle \varphi \in \mathcal{L}$

Basic (as opposed to *poly-*) modal logic is the special case where $|A| = 1$; there $\langle a \rangle \varphi$ is simply denoted $\diamond\varphi$. The *Hennessy–Milner logic* is polymodal logic with $P = \emptyset$. The language \mathcal{L}^∞ of *infinitary polymodal logic* over A and P is obtained from \mathcal{L} by additionally allowing $\bigwedge_{i \in I} \varphi_i$ to be in \mathcal{L}^∞ for arbitrary index sets I and $\varphi_i \in \mathcal{L}^\infty$ for $i \in I$. The connectives \top and \wedge are then the special cases $I = \emptyset$ and $|I| = 2$.

Definition 6 Let $(S, \rightarrow, \models)$ be an LTS over A and P . The relation $\models \subseteq S \times P$ can be extended to the *satisfaction relation* $\models \subseteq S \times \mathcal{L}^\infty$, by defining

- $s \models \bigwedge_{i \in I} \varphi_i$ if $s \models \varphi_i$ for all $i \in I$ – in particular, $s \models \top$ for any state $s \in S$
- $s \models \neg\varphi$ if $s \not\models \varphi$
- $s \models \langle a \rangle \varphi$ if there is a state t with $s \xrightarrow{a} t$ and $t \models \varphi$

Write $\mathcal{L}(s)$ for $\{\varphi \in \mathcal{L} \mid s \models \varphi\}$.

Theorem 1 [5] Let $(S, \rightarrow, \models)$ be an LTS and $s, t \in S$. Then $s \leftrightarrow t \Leftrightarrow \mathcal{L}^\infty(s) = \mathcal{L}^\infty(t)$.

In case the systems s and t are image finite, it suffices to consider finitary polymodal formulas only [3]. In fact, for this purpose it is enough to require that one of s and t is image finite.

Definition 7 Let $(S, \rightarrow, \models)$ be an LTS. A state $t \in S$ is *reachable* from $s \in S$ if there are $s_i \in S$ and $a_i \in A$ for $i = 0, \dots, n$ with $s = s_0, s_{i-1} \xrightarrow{a_i} s_i$ for $i = 1, \dots, n$, and $s_n = t$. A state $s \in S$ is *image finite* if for every state $t \in S$ reachable from s and for every $a \in A$, the set $\{u \in S \mid t \xrightarrow{a} u\}$ is finite.

Theorem 2 [4] Let $(S, \rightarrow, \models)$ be an LTS and $s, t \in S$ with s image finite. Then $s \leftrightarrow t \Leftrightarrow \mathcal{L}(s) = \mathcal{L}(t)$.

Non-well-Founded Sets

Another characterization of bisimulation semantics can be given by means of Aczel's universe \mathcal{V} of

non-well-founded sets [1]. This universe is an extension of the Von Neumann universe of well-founded sets, where the axiom of foundation (every chain $x_0 \ni x_1 \ni \dots$ terminates) is replaced by an *anti-foundation axiom*.

Definition 8 Let $(S, \rightarrow, \models)$ be an LTS, and let \mathcal{B} denote the unique function $\mathcal{M} : S \rightarrow \mathcal{V}$ satisfying, for all $s \in S$,

$$\mathcal{M}(s) = \{\langle a, \mathcal{M}(t) \rangle \mid s \xrightarrow{a} t\}.$$

It follows from Aczel's anti-foundation axiom that such a function exists. In fact, the axiom amounts to saying that systems of equations like the one above have unique solutions. $\mathcal{B}(s)$ could be taken to be the *branching structure* of s . The following theorem then says that two systems are bisimilar iff they have the same branching structure.

Theorem 3 [2] Let $(S, \rightarrow, \models)$ be an LTS and $s, t \in S$. Then $s \sqsubseteq t \Leftrightarrow \mathcal{B}(s) = \mathcal{B}(t)$.

Abstraction

In concurrency theory it is often useful to distinguish between *internal actions*, which do not admit interactions with the outside world, and *external ones*. As normally there is no need to distinguish the internal actions from each other, they all have the same name, namely, τ . If A is the set of external actions a certain class of systems may perform, then $A_\tau := A \dot{\cup} \{\tau\}$. Systems in that class are then represented by labeled transition systems over A_τ and a set of predicates P . The variant of bisimulation equivalence that treats τ just like any action of A is called *strong bisimulation equivalence*. Often, however, one wants to abstract from internal actions to various degrees. A system doing two τ actions in succession is then considered equivalent to a system doing just one. However, a system that can do either a or b is considered different from a system that can do either a or first τ and then b , because if the former system is placed in an environment where b cannot happen, it can still do a instead, whereas the latter system may reach a state (by executing the τ action) in which a is no longer possible.

Several versions of bisimulation equivalence that formalize these desiderata occur in the literature. *Branching bisimulation equivalence* [2], like strong bisimulation, faithfully preserves the branching structure

of related systems. The notions of *weak* and *delay* bisimulation equivalence, which were both introduced by Milner under the name *observational equivalence*, make more identifications, motivated by observable machine-behaviour according to certain testing scenarios.

Write $s \xrightarrow{\tau} t$ for $\exists n \geq 0 : \exists s_0, \dots, s_n : s = s_0 \xrightarrow{\tau} s_1 \xrightarrow{\tau} \dots \xrightarrow{\tau} s_n = t$, that is, a (possibly empty) path of τ -steps from s to t . Furthermore, for $a \in A_\tau$, write $s \xrightarrow{(a)} t$ for $s \xrightarrow{a} t \vee (a = \tau \wedge s = t)$. Thus $\xrightarrow{(a)}$ is the same as $\xrightarrow{\tau}$ for $a \in A$, and $\xrightarrow{\tau}$ denotes zero or one τ -step.

Definition 9 Let $(S, \rightarrow, \models)$ be an LTS over A_τ and P . Two states $s, t \in S$ are *branching bisimulation equivalent*, denoted $s \sqsubseteq_b t$, if they are related by a binary relation $R \subseteq S \times S$ (a *branching bisimulation*), satisfying:

- ^ if sRt and $s \models p$ with $p \in P$, then there is a t_1 with $t \xrightarrow{\tau} t_1 \models p$ and sRt_1 .
- ^ if sRt and $t \models p$ with $p \in P$, then there is a s_1 with $s \xrightarrow{\tau} s_1 \models p$ and s_1Rt .
- ^ if sRt and $s \xrightarrow{a} s'$ with $a \in A_\tau$, then there are t_1, t_2, t' with $t \xrightarrow{\tau} t_1 \xrightarrow{(a)} t_2 = t'$, sRt_1 , and $s'Rt'$.
- ^ if sRt and $t \xrightarrow{a} t'$ with $a \in A_\tau$, then there are s_1, s_2, s' with $s \xrightarrow{\tau} s_1 \xrightarrow{a} s_2 = s'$, s_1Rt , and $s'Rt'$.

Delay bisimulation equivalence, \sqsubseteq_d , is obtained by dropping the requirements sRt_1 and s_1Rt . *Weak bisimulation equivalence* [5], \sqsubseteq_w , is obtained by furthermore relaxing the requirements $t_2 = t'$ and $s_2 = s'$ to $t_2 \xrightarrow{\tau} t'$ and $s_2 \xrightarrow{\tau} s'$.

These definitions stem from concurrency theory. On Kripke structures, when studying modal or temporal logics, normally a stronger version of the first two conditions is imposed:

- ^ if sRt and $p \in P$, then $s \models p \Leftrightarrow t \models p$.

For systems without τ 's all these notions coincide with strong bisimulation equivalence.

Concurrency

When applied to *parallel systems*, capable of performing different actions at the same time, the versions of bisimulation discussed here employ *interleaving semantics*: no distinction is made between true parallelism and its nondeterministic sequential simulation. Versions of bisimulation that do make such a distinction have been developed as well, most notably the *ST-bisimulation* [2],

which takes temporal overlap of actions into account, and the *history preserving bisimulation* [2], which even keeps track of causal relations between actions. For this purpose, system representations such as *Petri nets* or *event structures* are often used instead of labeled transition systems.

Bibliography

1. Aczel P (1988) Non-well-founded Sets, CSLI Lecture Notes 14. Stanford University, Stanford, CA
2. van Glabbeek RJ (1990) Comparative concurrency semantics and refinement of actions. PhD thesis, Free University, Amsterdam. Second edition available as CWI tract 109, CWI, Amsterdam 1996
3. Hennessy M, Milner R (1985) Algebraic laws for nondeterminism and concurrency. J ACM 32(1):137–161
4. Hollenberg MJ (1995) Hennessy-Milner classes and process algebra. In: Pons A, de Rijke M, Venema Y (eds) Modal logic and process algebra: a bisimulation perspective, CSLI Lecture Notes 53, CSLI Publications, Stanford, CA, pp 187–216
5. Milner R (1990) Operational and algebraic semantics of concurrent processes. In: van Leeuwen J (ed) Handbook of theoretical computer science, Chapter 19. Elsevier Science Publishers B.V., North-Holland, pp 1201–1242

Further Readings

- Baeten JCM, Weijland WP (1990) Process algebra. Cambridge University Press
- Milner R (1989) Communication and concurrency. Prentice Hall, New Jersey
- Sangiorgi D (2009) on the origins of bisimulation and coinduction. ACM Trans Program Lang Syst 31(4). doi: 10.1145/1516507.1516510

Bisimulation Equivalence

► Bisimulation

SELIM G. AKL
Queen's University, Kingston, ON, Canada

Synonyms

Bitonic sorting network; Bitonic sorting

Definition

Bitonic Sort is a sorting algorithm that uses comparison-swap operations to arrange into nondecreasing order an input sequence of elements on which a linear order is defined (for example, numbers, words, and so on).

Discussion

Introduction

Henceforth, all inputs are assumed to be numbers, without loss of generality. For ease of presentation, it is also assumed that the length of the sequence to be sorted is an integer power of 2. The *Bitonic Sort* algorithm has the following properties:

1. *Oblivious*: The indices of all pairs of elements involved in comparison-swaps throughout the execution of the algorithm are predetermined, and do not depend in any way on the values of the input elements. It is important to note here that the elements to be sorted are assumed to be kept into an array and swapped in place. The indices that are referred to here are those in the array.
2. *Recursive*: The algorithm can be expressed as a procedure that calls itself to operate on smaller versions of the input sequence.
3. *Parallel*: It is possible to implement the algorithm using a set of special processors called “comparators” that operate simultaneously, each implementing a comparison-swap. A comparator receives a distinct pair of elements as input and produces that pair in sorted order as output in one time unit.

Bitonic sequence. A sequence $(a_1, a_2, \dots, a_{2m})$ is said to be *bitonic* if and only if:

(a) Either there is an integer j , $1 \leq j \leq 2m$, such that

$$a_1 \leq a_2 \leq \dots \leq a_j \geq a_{j+1} \geq a_{j+2} \geq \dots \geq a_{2m}$$

(b) Or the sequence does not initially satisfy the condition in (a), but can be shifted cyclically until the condition is satisfied.

For example, the sequence (8, 9, 10, 12, 7, 5, 4, 3, 1) is bitonic, as it satisfies condition (a). Similarly, the sequence (3, 2, 1, 5, 8, 9, 4), which does not satisfy condition (a), is also bitonic, as it can be shifted cyclically to obtain (1, 5, 8, 9, 4, 3, 2).

Let $(a_1, a_2, \dots, a_{2m})$ be a bitonic sequence, and let $d_i = \min(a_i, a_{m+i})$ and $e_i = \max(a_i, a_{m+i})$, for $i = 1, 2, \dots, m$. The following properties hold:

- (a) The sequences (d_1, d_2, \dots, d_m) and (e_1, e_2, \dots, e_m) are both bitonic.
- (b) $\max(d_1, d_2, \dots, d_m) \leq \min(e_1, e_2, \dots, e_m)$.

In order to prove the validity of these two properties, it suffices to consider sequences of the form:

$$a_1 \leq a_2 \leq \dots \leq a_{j-1} \leq a_j \geq a_{j+1} \geq \dots \geq a_{2m},$$

for some $1 \leq j \leq 2m$, since a cyclic shift of $\{a_1, a_2, \dots, a_{2m}\}$ affects $\{d_1, d_2, \dots, d_m\}$ and $\{e_1, e_2, \dots, e_m\}$ similarly, while affecting neither of the two properties to be established. In addition, there is no loss in generality to assume that $m < j \leq 2m$, since $\{a_{2m}, a_{2m-1}, \dots, a_1\}$ is also bitonic and neither property is affected by such reversal.

There are two cases:

1. If $a_m \leq a_{2m}$, then $a_i \leq a_{m+i}$. As a result, $d_i = a_i$, and $e_i = a_{m+i}$, for $1 \leq i \leq m$, and both properties hold.
2. If $a_m > a_{2m}$, then since $a_{j-m} \leq a_j$, an index k exists, where $j \leq k < 2m$, such that $a_{k-m} \leq a_k$ and $a_{k-m+1} > a_{k+1}$. Consequently:

$$d_i = a_i \text{ and } e_i = a_{m+i} \text{ for } 1 \leq i \leq k-m,$$

and

$$d_i = a_{m+i} \text{ and } e_i = a_i \text{ for } k-m < i \leq m.$$

Hence:

$$d_i \leq d_{i+1} \text{ for } 1 \leq i < k-m,$$

and

$$d_i \geq d_{i+1} \text{ for } k-m \leq i < m,$$

implying that $\{d_1, d_2, \dots, d_m\}$ is bitonic. Similarly, $e_i \leq e_{i+1}$, for $k-m \leq i < m$, $e_m \leq e_1$, $e_i \leq e_{i+1}$, for $1 \leq i < j-m$, and $e_i \geq e_{i+1}$, for $j-m \leq i < k-m$, implying that $\{e_1, e_2, \dots, e_m\}$ is also bitonic. Also,

$$\begin{aligned} \max(d_1, d_2, \dots, d_m) &= \max(d_{k-m}, d_{k-m+1}) \\ &= \max(a_{k-m}, a_{k+1}), \end{aligned}$$

and

$$\begin{aligned} \min(e_1, e_2, \dots, e_m) &= \min(e_{k-m}, e_{k-m+1}) \\ &= \min(a_k, a_{k-m+1}). \end{aligned}$$

Finally, since $a_k \geq a_{k+1}$, $a_k \geq a_{k-m}$, $a_{k-m+1} \geq a_{k-m}$, and $a_{k-m+1} \geq a_{k+1}$, it follows that:

$$\max(a_{k-m}, a_{k+1}) \leq \min(a_k, a_{k-m+1}).$$

Sorting a Bitonic Sequence

Given a bitonic sequence $(a_1, a_2, \dots, a_{2m})$, it can be sorted into a sequence $(c_1, c_2, \dots, c_{2m})$, arranged in nondecreasing order, by the following algorithm $MERGE_{2m}$:

Step 1. The two sequences (d_1, d_2, \dots, d_m) and (e_1, e_2, \dots, e_m) are produced.

Step 2. These two bitonic sequences are sorted independently and recursively, each by a call to $MERGE_m$.

It should be noted that in Step 2 the two sequences can be sorted independently (and simultaneously if enough comparators are available), since no element of (d_1, d_2, \dots, d_m) is larger than any element of (e_1, e_2, \dots, e_m) . The m smallest elements of the final sorted sequence are produced by sorting (d_1, d_2, \dots, d_m) , and the m largest by sorting (e_1, e_2, \dots, e_m) . The recursion terminates when $m = 2$, since $MERGE_2$ is implemented directly by one comparison-swap (or one comparator).

Sorting an Arbitrary Sequence

Algorithm $MERGE_{2m}$ assumes that the input sequence to be sorted is bitonic. However, it is easy to modify an arbitrary input sequence into a sequence of bitonic sequences as follows. Let the input sequence be (a_1, a_2, \dots, a_n) , and recall that, for simplicity, it is assumed that n is a power of 2. Now the following $n/2$ comparisons-swaps are performed: For all odd i , a_i is compared with a_{i+1} and a swap is applied if necessary. These comparison-swaps are numbered from 1 to $n/2$. Odd-numbered comparison-swaps place the smaller of (a_i, a_{i+1}) first and the larger of the pair second. Even-numbered comparison-swaps place the larger of (a_i, a_{i+1}) first and the smaller of the pair second.

At the end of this first stage, $n/4$ bitonic sequences are obtained. Each of these sequences is of length 4 and can be sorted using $MERGE_4$. These instances of $MERGE_4$ are numbered from 1 to $n/4$. Odd-numbered instances sort their inputs in nondecreasing order while

even-numbered instances sort their inputs in nonincreasing order. This yields $n/8$ bitonic sequences each of length 8.

The process continues until a single bitonic sequence of length n is produced and is sorted by giving it as input to $MERGE_n$. If a comparator is used to implement each comparison-swap, and all independent comparison-swaps are allowed to be executed in parallel, then the sequence is sorted in $((1 + \log n) \log n)/2$ time units (all logarithms are to the base 2). This is now illustrated.

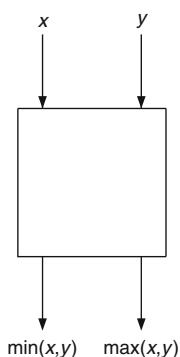
Implementation as a Combinational Circuit

[Figure 1](#) shows a schematic diagram of a comparator. The comparator receives two numbers x and y as input, and produces the smaller of the two on its left output line, and the larger of the two on its right output line.

A combinational circuit for sorting is a device, built entirely of comparators that takes an arbitrary sequence at one end and produces that sequence in sorted order at the other end. The comparators are arranged in rows. Each comparator receives its two inputs from the input sequence, or from comparators in the previous row, and delivers its two outputs to the comparators in the following row, or to the output sequence. A combinational circuit has no feedback: Data traverse the circuit in the same way as water flows from high to low terrain.

[Figure 2](#) shows a schematic diagram of a combinational circuit implementation of algorithm $MERGE_{2m}$, which sorts the bitonic sequence $(a_1, a_2, \dots, a_{2m})$ into nondecreasing order from smallest, on the leftmost line, to largest, on the rightmost.

Clearly, a bitonic sequence of length 2 is sorted by a single comparator. Combinational circuits for sorting



Bitonic Sort. Fig. 1 A comparator

bitonic sequences of length 4 and 8 are shown in [Figs. 3](#) and [4](#), respectively.

Finally, a combinational circuit for sorting an arbitrary sequence of numbers, namely, the sequence (5, 3, 2, 6, 1, 4, 7, 5) is shown in [Fig. 5](#). Comparators that reverse their outputs (i.e., those that produce the larger of their two inputs on the left output line, and the smaller on the right output line) are indicated with the letter R.

Analysis

The *depth* of a sorting circuit is the number of rows it contains – that is, the maximum number of comparators on a path from input to output. Depth, in other words, represents the time it takes the circuit to complete the sort. The *size* of a sorting circuit is defined as the number of comparators it uses.

Taking $2m = 2^i$, the depth of the circuit in [Fig. 2](#), which implements algorithm $MERGE_{2m}$ for sorting a bitonic sequence of length $2m$, is given by the recurrence:

$$\begin{aligned} d(2) &= 1 \\ d(2^i) &= 1 + d(2^{i-1}), \end{aligned}$$

whose solution is $d(2^i) = i$. The size of the circuit is given by the recurrence:

$$\begin{aligned} s(2) &= 1 \\ s(2^i) &= 2^{i-1} + 2s(2^{i-1}), \end{aligned}$$

whose solution is $s(2^i) = i2^{i-1}$.

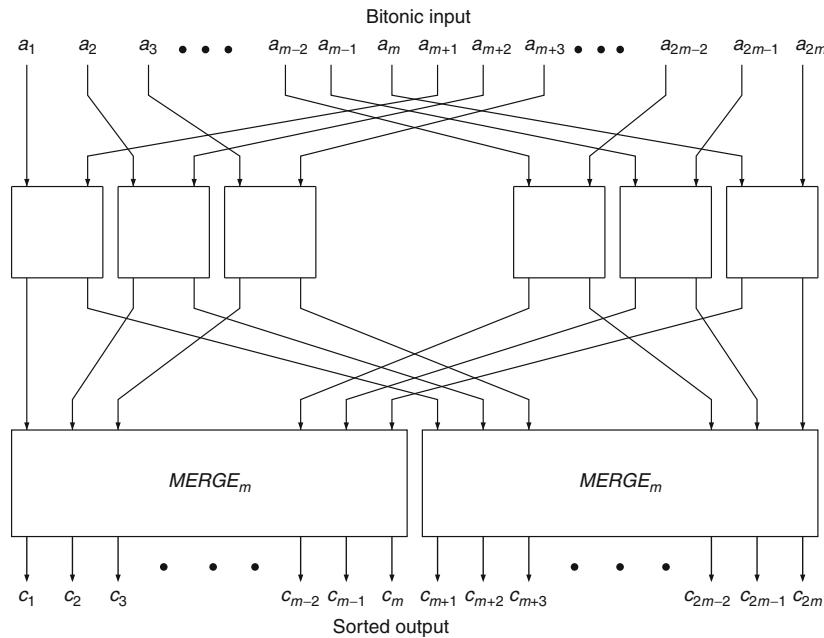
A circuit for sorting an arbitrary sequence of length n , such as the circuit in [Fig. 5](#) where $n = 8$, consists of $\log n$ phases: In the i th phase, $n/2^i$ circuits are required, each implementing $MERGE_{2^i}$, and having a size of $s(2^i)$ and a depth of $d(2^i)$, for $i = 1, 2, \dots, \log n$. The depth and size of this circuit are

$$\sum_{i=1}^{\log n} d(2^i) = \sum_{i=1}^{\log n} i = \frac{(1 + \log n) \log n}{2},$$

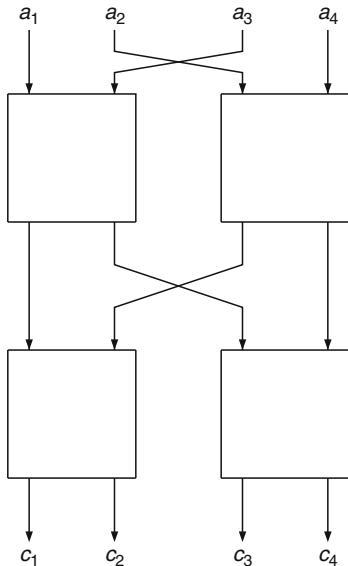
and

$$\begin{aligned} \sum_{i=1}^{\log n} 2^{(\log n)-i} s(2^i) &= \sum_{i=1}^{\log n} 2^{(\log n)-i} i 2^{i-1} \\ &= \frac{n(1 + \log n) \log n}{4}, \end{aligned}$$

respectively.



Bitonic Sort. Fig. 2 A circuit for sorting a bitonic sequence of length $2m$



Bitonic Sort. Fig. 3 A circuit for sorting a bitonic sequence of length 4

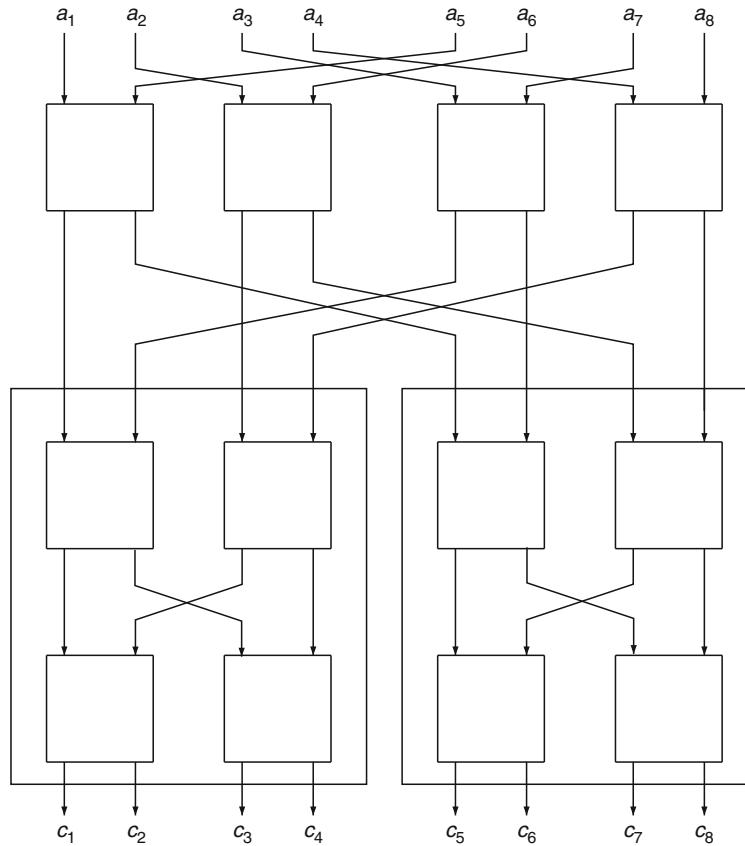
Lower Bounds

In order to evaluate the quality of the *Bitonic Sort* circuit, its depth and size are compared to two types of lower bounds.

A lower bound on the number of comparisons. A lower bound on the number of comparisons required in the worst case by a comparison-based sorting algorithm to sort a sequence of n numbers is derived as follows. All comparison-based sorting algorithms are modeled by a binary tree, each of whose nodes represents a comparison between two input elements. The leaves of the tree represent the $n!$ possible outcomes of the sort. A path from the root of the tree to a leaf is a complete execution of an algorithm on a given input. The length of the longest such path is $\log n!$, which is a quantity on the order of $n \log n$.

Several sequential comparison-based algorithms, such as *Heapsort* and *Mergesort*, for example, achieve this bound, to within a constant multiplicative factor, and are therefore said to be *optimal*. The *Bitonic Sort* circuit always runs in time on the order of $\log^2 n$ and is therefore significantly faster than any sequential algorithm. However, the number of comparators it uses, and therefore the number of comparisons it performs, is on the order of $n \log^2 n$, and consequently, it is not optimal in that sense.

Lower bounds on the depth and size. These lower bounds are specific to circuits. A lower bound on the depth of a sorting circuit for an input sequence of n elements is obtained as follows. Each comparator has



Bitonic Sort. Fig. 4 A circuit for sorting a bitonic sequence of length 8

two outputs, implying that an input element can reach at most 2^r locations after r rows. Since each element should be able to reach all n output positions, a lower bound on the depth of the circuit is on the order of $\log n$.

A lower bound on the size of a sorting circuit is obtained by observing that the circuit must be able to produce any of the $n!$ possible output sequences for each input sequence. Since each comparator can be in one of two states (swapping or not swapping its two inputs), the total number of configurations in which a circuit with c comparators can be is 2^c , and this needs to be at least equal to $n!$. Thus, a lower bound on the size of a sorting circuit is on the order of $n \log n$.

The *Bitonic Sort* circuit exceeds each of the lower bounds on depth and size by a factor of $\log n$, and is therefore not optimal in that sense as well.

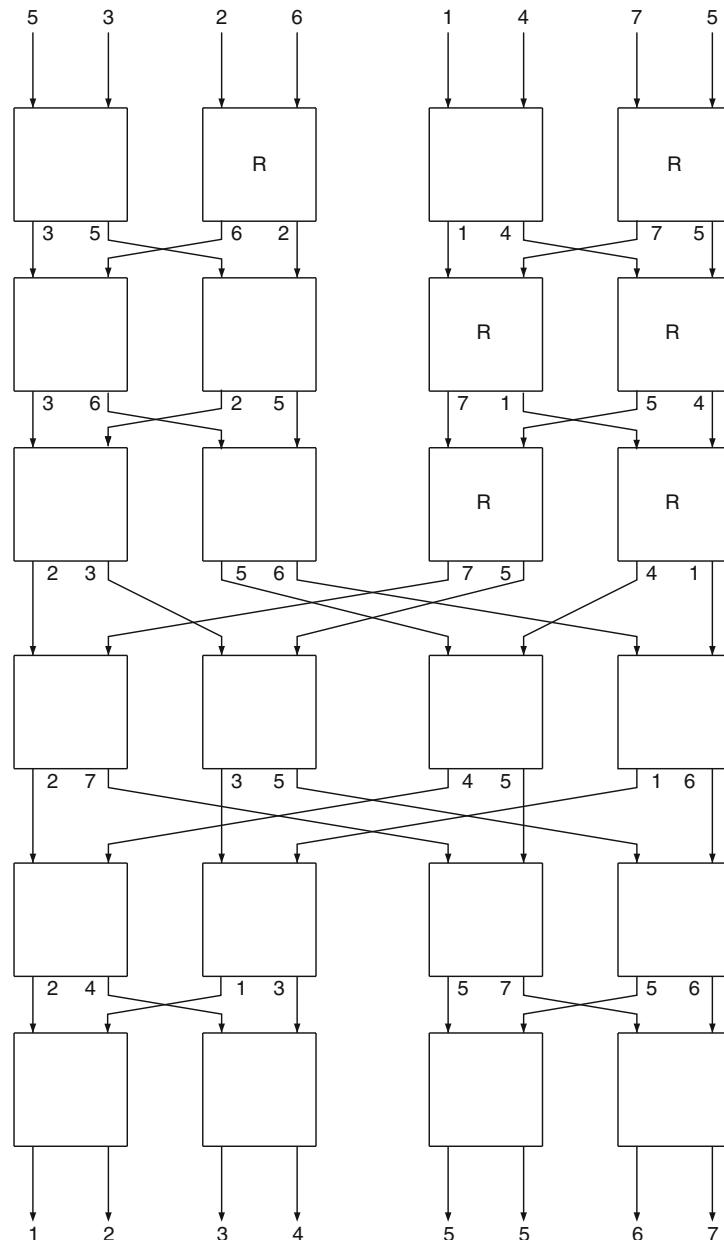
There exists a sorting circuit, known as the AKS circuit, which has a depth on the order of $\log n$ and a size on the order of $n \log n$. This circuit meets the lower

bound on the number of comparisons required to sort, in addition to the lower bounds on depth and size specific to circuits. It is, therefore, theoretically optimal on all counts.

In practice, however, the AKS circuit may not be very useful. In addition to its high conceptual complexity, its depth and size expressions are preceded by constants on the order of 10^3 . Even if the depth of the AKS circuit were as small as $200 \log n$, in order for the latter to be smaller than the depth of the *Bitonic Sort* circuit, namely, $((1 + \log n) \log n)/2$, the input sequence will need to have the astronomical length of $n > 2^{399}$, which is greater than the number of atoms that the observable universe is estimated to contain.

Future Directions

An interesting open problem in this area of research is to design a sorting circuit with the elegance, regularity, and simplicity of the *Bitonic Sort* circuit, while at the same



Bitonic Sort. Fig. 5 A circuit for sorting an arbitrary sequence of length 8

time matching the theoretical lower bounds on depth and size as closely as possible.

► Permutation Circuits

► Sorting

Related Entries

- ▶ AKS Sorting Network
 - ▶ Bitonic Sorting, Adaptive
 - ▶ Odd-Even Sorting

Bibliographic Notes and Further Reading

In 1968, Ken Batcher presented a paper at the AFIPS conference, in which he described two circuits for

sorting, namely, the *Odd-Even Sort* and the *Bitonic Sort* circuits [6]. This paper pioneered the study of parallel sorting algorithms and effectively launched the field of parallel algorithm design and analysis. Proofs of correctness of the *Bitonic Sort* algorithm are provided in [3, 6, 8]. Implementations of *Bitonic Sort* on other architectures beside combinational circuits have been proposed, including implementations on a perfect shuffle computer [20], a mesh of processors [16, 21], and on a shared-memory parallel machine [7].

In its simplest form, *Bitonic Sort* assumes that n , the length of the input sequence, is a power of 2. When n is not a power of 2, the sequence can be padded with z zeros such that $n + z$ is the smallest power of 2 larger than n . Alternatively, several variants of *Bitonic Sort* were proposed in the literature that are capable of sorting input sequences of arbitrary length [14, 15, 22].

Combinational circuits for sorting (also known as *sorting networks*) are discussed in [3–5, 8, 11–13, 17]. A lower bound for oblivious merging is derived in [9] and generalized in [23], which demonstrates that the bitonic merger is optimal to within a small constant factor. The AKS sorting circuit (whose name derives from the initials of its three inventors) was first described in [1] and then in [2]. Unlike the *Bitonic Sort* circuit that is based on the idea of repeatedly merging increasingly longer subsequences, the AKS circuit sorts by repeatedly splitting the original input sequence into increasingly shorter and disjoint subsequences. While theoretically optimal, the circuit suffers from large multiplicative constants in the expressions for its depth and size, making it of little use in practice, as mentioned earlier. A formulation in [18] manages to reduce the constants to a few thousands, still a prohibitive number. Descriptions of the AKS circuit appear in [5, 10, 17, 19]. Sequential sorting algorithms, including *Heapsort* and *Mergesort*, are covered in [8, 12].

One property of combinational circuits, not shared by many other parallel models of computation, is their ability to allow several input sequences to be processed simultaneously, in a *pipeline fashion*. This is certainly true of sorting circuits: Once the elements of the first input sequence have traversed the first row and moved on to the second, a new sequence can enter the first row,

and so on. If the circuit has depth D , then M sequences can be sorted in $D + M - 1$ time units [5].

Bibliography

1. Ajtai M, Komlós J, Szemerédi E (1983) An $O(n \log n)$ sorting network. In: Proceedings of the ACM symposium on theory of computing, Boston, Massachusetts, pp 1–9
2. Ajtai M, Komlós J, Szemerédi E (1983) Sorting in $c \log n$ parallel steps. Combinatorica 3:1–19
3. Akl SG (1985) Parallel sorting algorithms. Academic Press, Orlando, Florida
4. Akl SG (1989) The design and analysis of parallel algorithms. Prentice-Hall, Englewood Cliffs, New Jersey
5. Akl SG (1997) Parallel computation: models and methods. Prentice Hall, Upper Saddle River, New Jersey
6. Batcher KE (1968) Sorting networks and their applications. In: Proceedings of the AFIPS spring joint computer conference. Atlantic City, New Jersey, pp 307–314. Reprinted in: Wu CL, Feng TS (eds) Interconnection networks for parallel and distributed processing. IEEE Computer Society, 1984, pp 576–583
7. Bilardi G, Nicolau A (1989) Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. SIAM J Comput 18(2):216–228
8. Cormen TH, Leiserson CE, Rivest RL, Stein C (2001) Introduction to algorithms, 2nd edn. MIT Press/McGraw-Hill, Cambridge, Massachusetts/New York
9. Floyd RW (1972) Permuting information in idealized two-level storage. In: Miller RE, Thatcher JW (eds) Complexity of computer computations. Plenum Press, New York, pp 105–110
10. Gibbons A, Rytter W (1988) Efficient parallel algorithms. Cambridge University Press, Cambridge, England
11. JáJá J (1992) An introduction to parallel algorithms. Addison-Wesley, Reading, Massachusetts
12. Knuth DE (1998) The art of computer programming, vol 3. Addison-Wesley, Reading, Massachusetts
13. Leighton FT (1992) Introduction to parallel algorithms and architectures. Morgan Kaufmann, San Mateo, California
14. Liszka KJ, Batcher KE (1993) A generalized bitonic sorting network. In: Proceedings of the international conference on parallel processing, vol 1, pp 105–108
15. Nakatani T, Huang ST, Arden BW, Tripathi SK (1989) K-way bitonic sort. IEEE Trans Comput 38(2):283–288
16. Nassimi D, Sahni S (1980) Bitonic sort on a mesh-connected parallel computer. IEEE Trans Comput C-28(1):2–7
17. Parberry I (1987) Parallel complexity theory. Pitman, London
18. Paterson MS (1990) Improved sorting networks with $O(\log N)$ depth. Algorithmica 5:75–92
19. Smith JR (1993) The design and analysis of parallel algorithms. Oxford University Press, Oxford, England
20. Stone HS (1971) Parallel processing with the perfect shuffle. IEEE Trans Comput C-20(2):153–161

21. Thompson CD, Kung HT (1977) Sorting on a mesh-connected parallel computer. Commun ACM 20(4):263–271
22. Wang BF, Chen GH, Hsu CC (1991) Bitonic sort with an arbitrary number of keys. In: Proceedings of the international conference on parallel processing, vol 3, Illinois, pp 58–65
23. Yao AC, Yao FF (1975) Lower bounds for merging networks. J ACM 23(3):566–571

- It can be implemented in a highly parallel manner on modern architectures, such as a streaming architecture (GPUs), even without any scatter operations, that is, without random access writes.

One of the main differences between “regular” bitonic sorting and adaptive bitonic sorting is that regular bitonic sorting is data-independent, while adaptive bitonic sorting is data-dependent (hence the name).

As a consequence, adaptive bitonic sorting cannot be implemented as a sorting network, but only on architectures that offer some kind of flow control. Nonetheless, it is convenient to derive the method of adaptive bitonic sorting from bitonic sorting.

Sorting networks have a long history in computer science research (see the comprehensive survey [2]). One reason is that sorting networks are a convenient way to describe parallel sorting algorithms on CREW-PRAMs or even EREW-PRAMs (which is also called PRAC for “parallel random access computer”).

In the following, let n denote the number of keys to be sorted, and p the number of processors. For the sake of clarity, n will always be assumed to be a power of 2. (In their original paper [5], Bilardi and Nicolau have described how to modify the algorithms such that they can handle arbitrary numbers of keys, but these technical details will be omitted in this article.)

The first to present a sorting network with optimal asymptotic complexity were Ajtai, Komlós, and Szemeredi [1]. Also, Cole [6] presented an optimal parallel merge sort approach for the CREW-PRAM as well as for the EREW-PRAM. However, it has been shown that neither is fast in practice for reasonable numbers of keys [8, 15].

In contrast, adaptive bitonic sorting requires less than $2n \log n$ comparisons in total, independent of the number of processors. On p processors, it can be implemented in $O\left(\frac{n \log n}{p}\right)$ time, for $p \leq \frac{n}{\log n}$.

Even with a small number of processors it is efficient in practice: in its original implementation, the sequential version of the algorithm was at most by a factor 2.5 slower than quicksort (for sequence lengths up to 2^{19}) [5].

Fundamental Properties

One of the fundamental concepts in this context is the notion of a *bitonic sequence*.

Bitonic Sorting Network

► Bitonic Sort

Bitonic Sorting, Adaptive

GABRIEL ZACHMANN

Clausthal University, Clausthal-Zellerfeld, Germany

Definition

Adaptive bitonic sorting is a sorting algorithm suitable for implementation on EREW parallel architectures. Similar to bitonic sorting, it is based on merging, which is recursively applied to obtain a sorted sequence. In contrast to bitonic sorting, it is data-dependent. Adaptive bitonic merging can be performed in $O\left(\frac{n}{p}\right)$ parallel time, p being the number of processors, and executes only $O(n)$ operations in total. Consequently, adaptive bitonic sorting can be performed in $O\left(\frac{n \log n}{p}\right)$ time, which is optimal. So, one of its advantages is that it executes a factor of $O(\log n)$ less operations than bitonic sorting. Another advantage is that it can be implemented efficiently on modern GPUs.

Discussion

Introduction

This chapter describes a parallel sorting algorithm, *adaptive bitonic sorting* [5], that offers the following benefits:

- It needs only the optimal total number of comparison/exchange operations, $O(n \log n)$.
- The hidden constant in the asymptotic number of operations is less than in other optimal parallel sorting methods.

Definition 1 (Bitonic sequence) Let $\mathbf{a} = (a_0, \dots, a_{n-1})$ be a sequence of numbers. Then, \mathbf{a} is *bitonic*, iff it monotonically increases and then monotonically decreases, or if it can be cyclically shifted (i.e., rotated) to become monotonically increasing and then monotonically decreasing.

Figure 1 shows some examples of bitonic sequences.

In the following, it will be easier to understand any reasoning about bitonic sequences, if one considers them as being arranged in a circle or on a cylinder: then, there are only two inflection points around the circle. This is justified by **Definition 1**. **Figure 2** depicts an example in this manner.

As a consequence, all index arithmetic is understood *modulo n*, that is, index $i + k \equiv i + k \bmod n$, unless otherwise noted, so indices range from 0 through $n - 1$.

As mentioned above, adaptive bitonic sorting can be regarded as a variant of bitonic sorting, which is in order to capture the notion of “rotational invariance” (in some

sense) of bitonic sequences; it is convenient to define the following *rotation operator*.

Definition 2 (Rotation) Let $\mathbf{a} = (a_0, \dots, a_{n-1})$ and $j \in \mathbb{N}$. We define a rotation as an operator R_j on the sequence \mathbf{a} :

$$R_j \mathbf{a} = (a_j, a_{j+1}, \dots, a_{j+n-1})$$

This operation is performed by the network shown in **Fig. 4**. Such networks are comprised of elementary comparators (see **Fig. 3**).

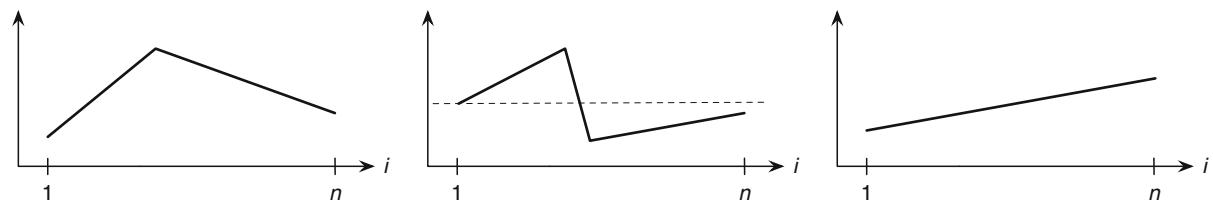
Two other operators are convenient to describe sorting.

Definition 3 (Half-cleaner) Let $\mathbf{a} = (a_0, \dots, a_{n-1})$.

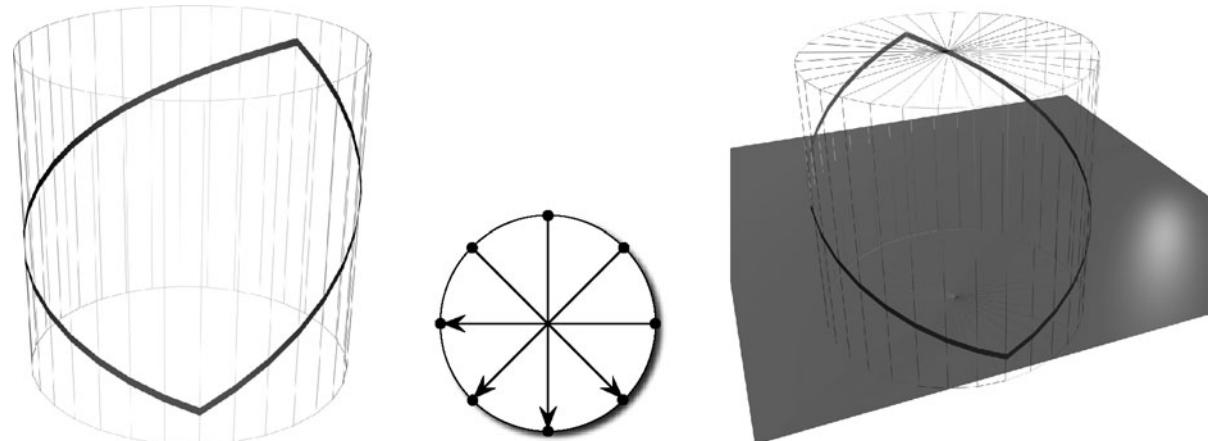
$$L\mathbf{a} = (\min(a_0, a_{\frac{n}{2}}), \dots, \min(a_{\frac{n}{2}-1}, a_{n-1})),$$

$$U\mathbf{a} = (\max(a_0, a_{\frac{n}{2}}), \dots, \max(a_{\frac{n}{2}-1}, a_{n-1})).$$

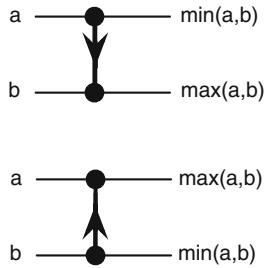
In [7], a network that performs these operations together is called a *half-cleaner* (see **Fig. 5**).



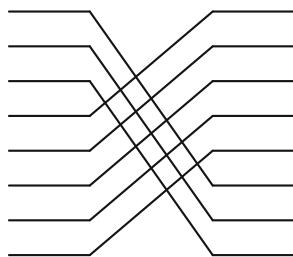
Bitonic Sorting, Adaptive. Fig. 1 Three examples of sequences that are bitonic. Obviously, the mirrored sequences (either way) are bitonic, too



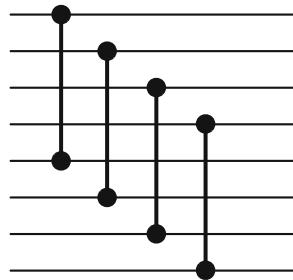
Bitonic Sorting, Adaptive. Fig. 2 Left: according to their definition, bitonic sequences can be regarded as lying on a cylinder or as being arranged in a circle. As such, they consist of one monotonically increasing and one decreasing part. Middle: in this point of view, the network that performs the L and U operators (see **Fig. 5**) can be visualized as a wheel of “spokes.” Right: visualization of the effect of the L and U operators; the blue plane represents the median



Bitonic Sorting, Adaptive. Fig. 3 Comparator/exchange elements



Bitonic Sorting, Adaptive. Fig. 4 A network that performs the rotation operator



Bitonic Sorting, Adaptive. Fig. 5 A network that performs the L and U operators

It is easy to see that, for any j and \mathbf{a} ,

$$L\mathbf{a} = R_{-j \bmod \frac{n}{2}} LR_j \mathbf{a}, \quad (1)$$

and

$$U\mathbf{a} = R_{-j \bmod \frac{n}{2}} UR_j \mathbf{a}. \quad (2)$$

This is the reason why the cylinder metaphor is valid.

The proof needs to consider only two cases: $j = \frac{n}{2}$ and $1 \leq j < \frac{n}{2}$. In the former case, Eq. 1 becomes $L\mathbf{a} =$

$LR_{\frac{n}{2}} \mathbf{a}$, which can be verified trivially. In the latter case, Eq. 1 becomes

$$\begin{aligned} LR_j \mathbf{a} &= (\min(a_j, a_{j+\frac{n}{2}}), \dots, \min(a_{\frac{n}{2}-1}, a_{n-1}), \dots, \\ &\quad \min(a_{j-1}, a_{j-1+\frac{n}{2}})) \\ &= R_j L\mathbf{a}. \end{aligned}$$

Thus, with the cylinder metaphor, the L and U operators basically do the following: cut the cylinder with circumference n at any point, roll it around a cylinder with circumference $\frac{n}{2}$, and perform position-wise the max and min operator, respectively. Some examples are shown in Fig. 6.

The following theorem states some important properties of the L and U operators.

Theorem 1 Given a bitonic sequence \mathbf{a} ,

$$\max\{L\mathbf{a}\} \leq \min\{U\mathbf{a}\}.$$

Moreover, $L\mathbf{a}$ and $U\mathbf{a}$ are bitonic too.

In other words, each element of $L\mathbf{a}$ is less than or equal to each element of $U\mathbf{a}$.

This theorem is the basis for the construction of the bitonic sorter [4]. The first step is to devise a *bitonic merger* (BM). We denote a BM that takes as input bitonic sequences of length n with BM_n . A BM is recursively defined as follows:

$$BM_n(\mathbf{a}) = (BM_{\frac{n}{2}}(L\mathbf{a}), BM_{\frac{n}{2}}(U\mathbf{a})).$$

The base case is, of course, a two-key sequence, which is handled by a single comparator. A BM can be easily represented in a network as shown in Fig. 7.

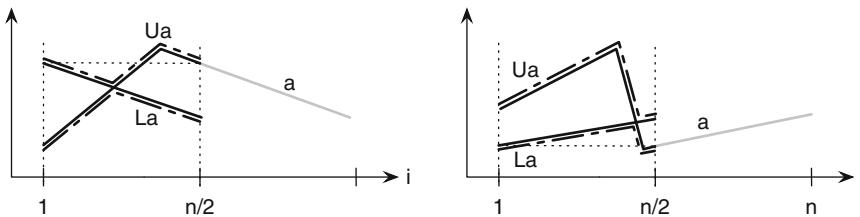
Given a bitonic sequence \mathbf{a} of length n , one can show that

$$BM_n(\mathbf{a}) = Sorted(\mathbf{a}). \quad (3)$$

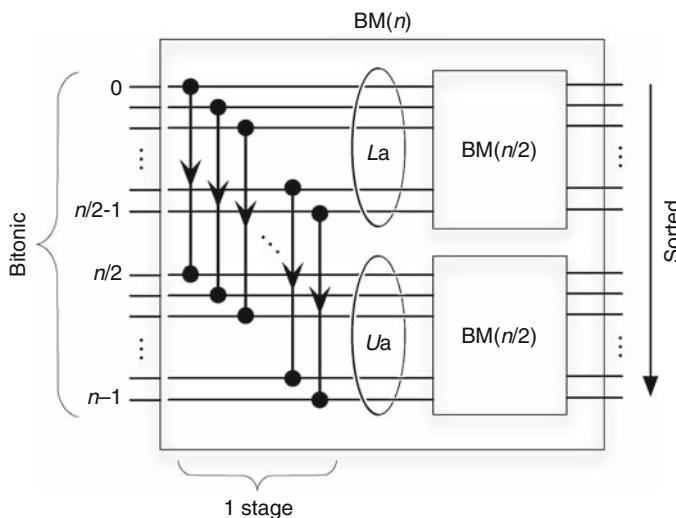
It should be obvious that the sorting direction can be changed simply by swapping the direction of the elementary comparators.

Coming back to the metaphor of the cylinder, the first stage of the bitonic merger in Fig. 7 can be visualized as $\frac{n}{2}$ comparators, each one connecting an element of the cylinder with the opposite one, somewhat like spokes in a wheel. Note that here, while the cylinder can rotate freely, the “spokes” must remain fixed.

From a bitonic merger, it is straightforward to derive a bitonic sorter, BS_n , that takes an unsorted sequence, and produces a sorted sequence either up or down. Like the BM, it is defined recursively, consisting of two



Bitonic Sorting, Adaptive. Fig. 6 Examples of the result of the L and U operators. Conceptually, these operators fold the bitonic sequence (black), such that the part from indices $\frac{n}{2} + 1$ through n (light gray) is shifted into the range 1 through $\frac{n}{2}$ (black); then, L and U yield the upper (medium gray) and lower (dark gray) hull, respectively



Bitonic Sorting, Adaptive. Fig. 7 Schematic, recursive diagram of a network that performs bitonic merging

smaller bitonic sorters and a bitonic merger (see Fig. 8). Again, the base case is the two-key sequence.

Analysis of the Number of Operations of Bitonic Sorting

Since a bitonic sorter basically consists of a number of bitonic mergers, it suffices to look at the total number of comparisons of the latter.

The total number of comparators, $C(n)$, in the bitonic merger BM_n is given by:

$$C(n) = 2C\left(\frac{n}{2}\right) + \frac{n}{2}, \quad \text{with } C(2) = 1,$$

which amounts to

$$C(n) = \frac{1}{2}n \log n.$$

As a consequence, the bitonic sorter consists of $O(n \log^2 n)$ comparators.

Clearly, there is some redundancy in such a network, since n comparisons are sufficient to merge two sorted sequences. The reason is that the comparisons performed by the bitonic merger are *data-independent*.

Derivation of Adaptive Bitonic Merging

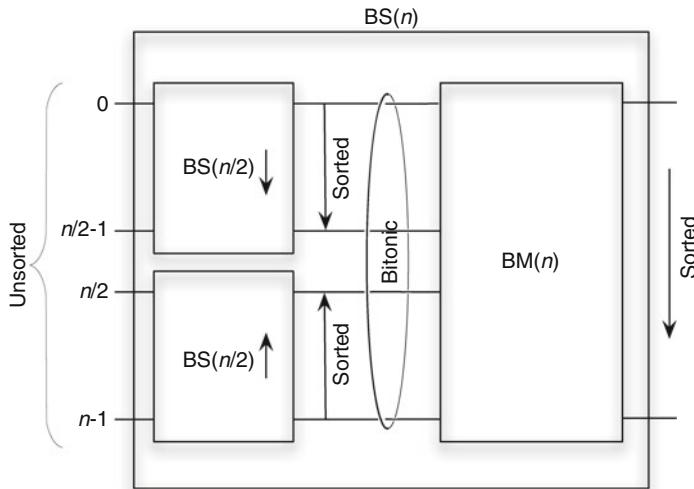
The algorithm for adaptive bitonic sorting is based on the following theorem.

Theorem 2 Let \mathbf{a} be a bitonic sequence. Then, there is an index q such that

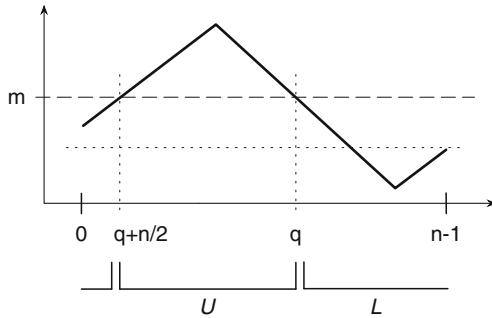
$$L\mathbf{a} = (a_q, \dots, a_{q+\frac{n}{2}-1}) \quad (4)$$

$$U\mathbf{a} = (a_{q+\frac{n}{2}}, \dots, a_{q-1}) \quad (5)$$

(Remember that index arithmetic is always modulo n .)



Bitonic Sorting, Adaptive. Fig. 8 Schematic, recursive diagram of a bitonic sorting network



Bitonic Sorting, Adaptive. Fig. 9 Visualization for the proof of Theorem 2

The following outline of the proof assumes, for the sake of simplicity, that all elements in \mathbf{a} are distinct. Let m be the median of all a_i , that is, $\frac{n}{2}$ elements of \mathbf{a} are less than or equal to m , and $\frac{n}{2}$ elements are larger. Because of Theorem 1,

$$\max\{L\mathbf{a}\} \leq m < \min\{U\mathbf{a}\}.$$

Employing the cylinder metaphor again, the median m can be visualized as a horizontal plane $z = m$ that cuts the cylinder. Since \mathbf{a} is bitonic, this plane cuts the sequence in exactly two places, that is, it partitions the sequence into two contiguous halves (actually, any horizontal plane, i.e., any percentile partitions a bitonic sequence in two contiguous halves), and since it is

the median, each half must have length $\frac{n}{2}$. The indices where the cut happens are q and $q + \frac{n}{2}$. Figure 9 shows an example (in one dimension).

The following theorem is the final keystone for the adaptive bitonic sorting algorithm.

Theorem 3 Any bitonic sequence \mathbf{a} can be partitioned into four subsequences $(\mathbf{a}^1, \mathbf{a}^2, \mathbf{a}^3, \mathbf{a}^4)$ such that either

$$(L\mathbf{a}, U\mathbf{a}) = (\mathbf{a}^1, \mathbf{a}^4, \mathbf{a}^3, \mathbf{a}^2) \quad (6)$$

or

$$(L\mathbf{a}, U\mathbf{a}) = (\mathbf{a}^3, \mathbf{a}^2, \mathbf{a}^1, \mathbf{a}^4). \quad (7)$$

Furthermore,

$$|\mathbf{a}^1| + |\mathbf{a}^2| = |\mathbf{a}^3| + |\mathbf{a}^4| = \frac{n}{2}, \quad (8)$$

$$|\mathbf{a}^1| = |\mathbf{a}^3|, \quad (9)$$

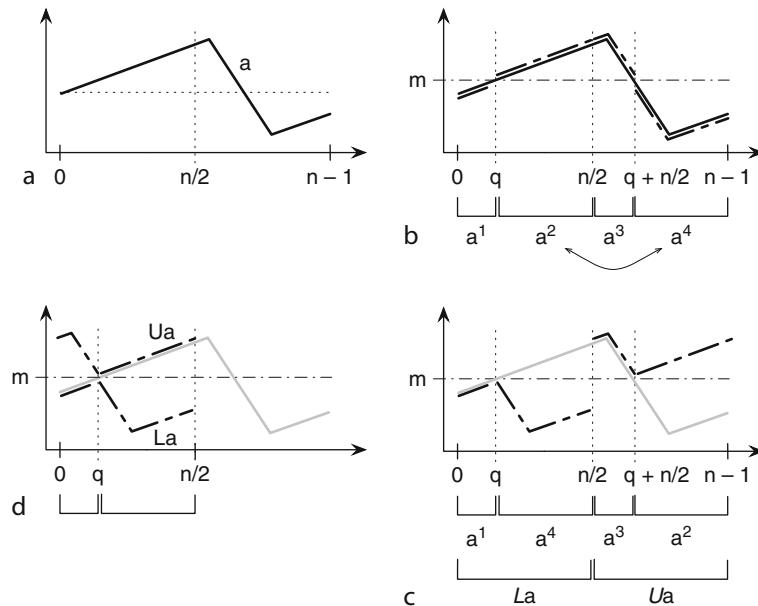
and

$$|\mathbf{a}^2| = |\mathbf{a}^4|, \quad (10)$$

where $|\mathbf{a}|$ denotes the length of sequence \mathbf{a} .

Figure 10 illustrates this theorem by an example.

This theorem can be proven fairly easily too: the length of the subsequences is just q and $\frac{n}{2} - q$, where q is the same as in Theorem 2. Assuming that $\max\{\mathbf{a}^1\} < m < \min\{\mathbf{a}^3\}$, nothing will change between those two subsequences (see Fig. 10). However, in that case $\min\{\mathbf{a}^2\} > m > \max\{\mathbf{a}^4\}$; therefore, by swapping



Bitonic Sorting, Adaptive. Fig. 10 Example illustrating Theorem 3

a^2 and a^4 (which have equal length), the bounds $\max\{a^1, a^4\} < m < \min\{a^3, a^3\}$ are obtained. The other case can be handled analogously.

Remember that there are $\frac{n}{2}$ comparator-and-exchange elements, each of which compares a_i and $a_{i+\frac{n}{2}}$. They will perform exactly this exchange of subsequences, without ever looking at the data.

Now, the idea of adaptive bitonic sorting is to find the subsequences, that is, to find the index q that marks the border between the subsequences. Once q is found, one can (conceptually) swap the subsequences, instead of performing $\frac{n}{2}$ comparisons unconditionally.

Finding q can be done simply by binary search driven by comparisons of the form $(a_i, a_{i+\frac{n}{2}})$.

Overall, instead of performing $\frac{n}{2}$ comparisons in the first stage of the bitonic merger (see Fig. 7), the adaptive bitonic merger performs $\log(\frac{n}{2})$ comparisons in its first stage (although this stage is no longer representable by a network).

Let $C(n)$ be the total number of comparisons performed by adaptive bitonic merging, in the worst case. Then

$$C(n) = 2C\left(\frac{n}{2}\right) + \log(n) = \sum_{i=0}^{k-1} 2^i \log\left(\frac{n}{2^i}\right),$$

with $C(2) = 1, C(1) = 0$ and $n = 2^k$. This amounts to

$$C(n) = 2n - \log n - 2.$$

The only question that remains is how to achieve the data rearrangement, that is, the swapping of the subsequences a^1 and a^3 or a^2 and a^4 , respectively, without sacrificing the worst-case performance of $O(n)$. This can be done by storing the keys in a perfectly balanced tree (assuming $n = 2^k$), the so-called bitonic tree. (The tree can, of course, store only $2^k - 1$ keys, so the n -th key is simply stored separately.) This tree is very similar to a search tree, which stores a monotonically increasing sequence: when traversed in-order, the bitonic tree produces a sequence that lists the keys such that there are exactly two inflection points (when regarded as a circular list).

Instead of actually copying elements of the sequence in order to achieve the exchange of subsequences, the adaptive bitonic merging algorithm swaps $O(\log n)$ pointers in the bitonic tree. The recursion then works on the two subtrees. With this technique, the overall number of operations of adaptive bitonic merging is $O(n)$. Details can be found in [5].

Clearly, the adaptive bitonic sorting algorithm needs $O(n \log n)$ operations in total, because it consists of $\log(n)$ many complete merge stages (see Fig. 8).

It should also be fairly obvious that the adaptive bitonic sorter performs an (adaptive) subset of the comparisons that are executed by the (nonadaptive) bitonic sorter.

The Parallel Algorithm

So far, the discussion assumed a sequential implementation. Obviously, the algorithm for adaptive bitonic merging can be implemented on a parallel architecture, just like the bitonic merger, by executing recursive calls on the same level in parallel.

Unfortunately, a naïve implementation would require $O(\log^2 n)$ steps in the worst case, since there are $\log(n)$ levels. The bitonic merger achieves $O(\log n)$ parallel time, because all pairwise comparisons within one stage can be performed in parallel. But this is not straightforward to achieve for the $\log(n)$ comparisons of the binary-search method in adaptive bitonic merging, which are inherently sequential.

However, a careful analysis of the data dependencies between comparisons of successive stages reveals that the execution of different stages can be partially overlapped [5]. As La , Ua are being constructed in one stage by moving down the tree in parallel layer by layer (occasionally swapping pointers); this process can be started for the next stage, which begins one layer beneath the one where the previous stage began, before the first stage has finished, provided the first stage has progressed “far enough” in the tree. Here, “far enough” means exactly two layers ahead.

This leads to a parallel version of the adaptive bitonic merge algorithm that executes in time $O\left(\frac{n}{p}\right)$ for $p \in O\left(\frac{n}{\log n}\right)$, that is, it can be executed in $(\log n)$ parallel time.

Furthermore, the data that needs to be communicated between processors (either via memory, or via communication channels) is in $O(p)$.

It is straightforward to apply the classical sorting-by-merging approach here (see Fig. 8), which yields the *adaptive bitonic sorting* algorithm. This can be implemented on an EREW machine with p processors in $O\left(\frac{n \log n}{p}\right)$ time, for $p \in O\left(\frac{n}{\log n}\right)$.

A GPU Implementation

Because adaptive bitonic sorting has excellent scalability (the number of processors, p , can go up to $n/\log(n)$) and the amount of inter-process communication is fairly low (only $O(p)$), it is perfectly suitable for implementation on stream processing architectures. In addition, although it was designed for a random access architecture, adaptive bitonic sorting can be adapted to a stream processor, which (in general) does not have the ability of random-access writes. Finally, it can be implemented on a GPU such that there are only $O(\log^2(n))$ passes (by utilizing $O(n/\log(n))$ (conceptual) processors), which is very important, since the

Algorithm 1: Adaptive construction of La and Ua (one stage of adaptive bitonic merging)

```

input : Bitonic tree, with root node r and extra
       node e, representing bitonic sequence a
output: La in the left subtree of r plus root r, and Ua
       in the right subtree of r plus extra node e

// phase 0: determine case
if value(r) < value(e) then
  case = 1
else
  case = 2
  swap value(r) and value(e)
  (p, q) = (left(r), right(r))
for i = 1, . . . , log n - 1 do
  // phase i
  test = (value(p) > value(q))
  if test == true then
    swap values of p and q
    if case == 1 then
      swap the pointers left(p) and
      left(q)
    else
      swap the pointers right(p) and
      right(q)
  if (case == 1 and test == false) or (case ==
  2 and test == true) then
    (p, q) = (left(p), left(q))
  else
    (p, q) = (right(p), right(q))

```

Algorithm 2: Merging a bitonic sequence to obtain a sorted sequence

input : Bitonic tree, with root node r and extra node e , representing bitonic sequence a

output: Sorted tree (produces $\text{sort}(a)$ when traversed in-order)

construct La and Ua in the bitonic tree by Algorithm 1

call merging recursively with $\text{left}(r)$ as root and r as extra node

call merging recursively with $\text{right}(r)$ as root and e as extra node

number of passes is one of the main limiting factors on GPUs.

This section provides more details on the implementation on a GPU, called “GPU-ABiSort” [11, 12]. For the sake of simplicity, the following always assumes increasing sorting direction, and it is thus not explicitly specified. As noted above, the sorting direction must be reversed in the right branch of the recursion in the bitonic sorter, which basically amounts to reversing the comparison direction of the values of the keys, that is, compare for $<$ instead of $>$ in Algorithm 3.

As noted above, the bitonic tree stores the sequence (a_0, \dots, a_{n-2}) in in-order, and the key a_{n-1} is stored in the *extra node*. As mentioned above, an algorithm that constructs (La, Ua) from a can traverse this bitonic tree and swap pointers as necessary. The index q , which is mentioned in the proof for Theorem 3, is only determined implicitly. The two different cases that are mentioned in Theorem 3 and Eqs. 6 and 7 can be distinguished simply by comparing elements $a_{\frac{n}{2}-1}$ and a_{n-1} .

This leads to Algorithm 1. Note that the root of the bitonic tree stores element $a_{\frac{n}{2}-1}$ and the extra node stores a_{n-1} . Applying this recursively yields Algorithm 2. Note that the bitonic tree needs to be constructed only once at the beginning during setup time.

Because branches are very costly on GPUs, one should avoid as many conditionals in the inner loops as possible. Here, one can exploit the fact that $R_{n/2}a = (a_{\frac{n}{2}}, \dots, a_{n-1}, a_0, \dots, a_{\frac{n}{2}-1})$ is bitonic, provided a is bitonic too. This operation basically amounts to swapping the two pointers $\text{left}(\text{root})$ and $\text{right}(\text{root})$. The

Algorithm 3: Simplified adaptive construction of La and Ua

input : Bitonic tree, with root node r and extra node e , representing bitonic sequence a

output: La in the left subtree of r plus root r , and Ua in the right subtree of r plus extra node e

// phase 0

if $\text{value}(r) > \text{value}(e)$ **then**
 swap $\text{value}(r)$ and $\text{value}(e)$
 swap pointers $\text{left}(r)$ and $\text{right}(r)$
 $(p, q) = (\text{left}(r), \text{right}(r))$

for $i = 1, \dots, \log n - 1$ **do**

// phase i

if $\text{value}(p) > \text{value}(q)$ **then**
 swap $\text{value}(p)$ and $\text{value}(q)$
 swap pointers $\text{left}(p)$ and $\text{left}(q)$
 $(p, q) = (\text{right}(p), \text{right}(q))$

else
 $(p, q) = (\text{left}(p), \text{left}(q))$

simplified construction of La and Ua is presented in Algorithm 3. (Obviously, the simplified algorithm now really needs trees with pointers, whereas Bilardi’s original bitonic tree could be implemented pointer-less (since it is a complete tree). However, in a real-world implementation, the keys to be sorted must carry pointers to some “payload” data anyway, so the additional memory overhead incurred by the child pointers is at most a factor 1.5.)

Outline of the Implementation

As explained above, on each recursion level $j = 1, \dots, \log(n)$ of the adaptive bitonic sorting algorithm, $2^{\log n - j + 1}$ bitonic trees, each consisting of 2^{j-1} nodes, have to be merged into $2^{\log n - j}$ bitonic trees of 2^j nodes. The merge is performed in j stages. In each stage $k = 0, \dots, j-1$, the construction of La and Ua is executed on 2^k subtrees. Therefore, $2^{\log n - j} \cdot 2^k$ instances of the La / Ua construction algorithm can be executed in parallel during that stage. On a stream architecture, this potential parallelism can be exposed by allocating a stream consisting of $2^{\log n - j + k}$ elements and executing a so-called kernel on each element.

The *La / Ua* construction algorithm consists of $j - k$ phases, where each phase reads and modifies a pair of nodes, (p, q) , of a bitonic tree. Assume that a kernel implementation performs the operation of a single phase of this algorithm. (How such a kernel implementation is realized without random-access writes will be described below.) The temporary data that have to be preserved from one phase of the algorithm to the next one are just two node pointers (p and q) per kernel instance. Thus, each of the $2^{\log n-j+k}$ elements of the allocated stream consist of exactly these two node pointers. When the kernel is invoked on that stream, each kernel instance reads a pair of node pointers, (p, q) , from the stream, performs one phase of the *La/Ua* construction algorithm, and finally writes the updated pair of node pointers (p, q) back to the stream.

Eliminating Random-Access Writes

Since GPUs do not support random-access writes (at least, for almost all practical purposes, random-access writes would kill any performance gained by the parallelism) the kernel has to be implemented so that it modifies node pairs (p, q) of the bitonic tree without random-access writes. This means that it can output node pairs from the kernel only via linear stream write. But this way it cannot write a modified node pair to its original location from where it was read. In addition, it cannot simply take an input stream (containing a bitonic tree) and produce another output stream (containing the modified bitonic tree), because then it would have to process the nodes in the same order as they are stored in memory, but the adaptive bitonic merge processes them in a random, data-dependent order.

Fortunately, the bitonic tree is a linked data structure where all nodes are directly or indirectly linked to the root (except for the extra node). This allows us to change the location of nodes in memory during the merge algorithm as long as the child pointers of their respective parent nodes are updated (and the root and extra node of the bitonic tree are kept at well-defined memory locations). This means that for each node that is modified its parent node has to be modified also, in order to update its child pointers.

Notice that Algorithm 3 basically traverses the bitonic tree down along a path, changing some of the

nodes as necessary. The strategy is simple: simply output every node visited along this path to a stream. Since the data layout is fixed and predetermined, the kernel can store the index of the children with the node as it is being written to the output stream. One child address remains the same anyway, while the other is determined when the kernel is still executing for the current node. Figure 11 demonstrates the operation of the stream program using the described stream output technique.

Complexity

A simple implementation on the GPU would need $O(\log^2 n)$ phases (or “passes” in GPU parlance) in total for adaptive bitonic sorting, which amounts to $O(\log^3 n)$ operations in total.

This is already very fast in practice. However, the optimal complexity of $O(\log n)$ passes can be achieved exactly as described in the original work [5], that is, phase i of a stage k can be executed immediately after phase $i + 1$ of stage $k - 1$ has finished. Therefore, the execution of a new stage can start at every other step of the algorithm.

The only difference from the simple implementation is that kernels now must write to parts of the output stream, because other parts are still in use.

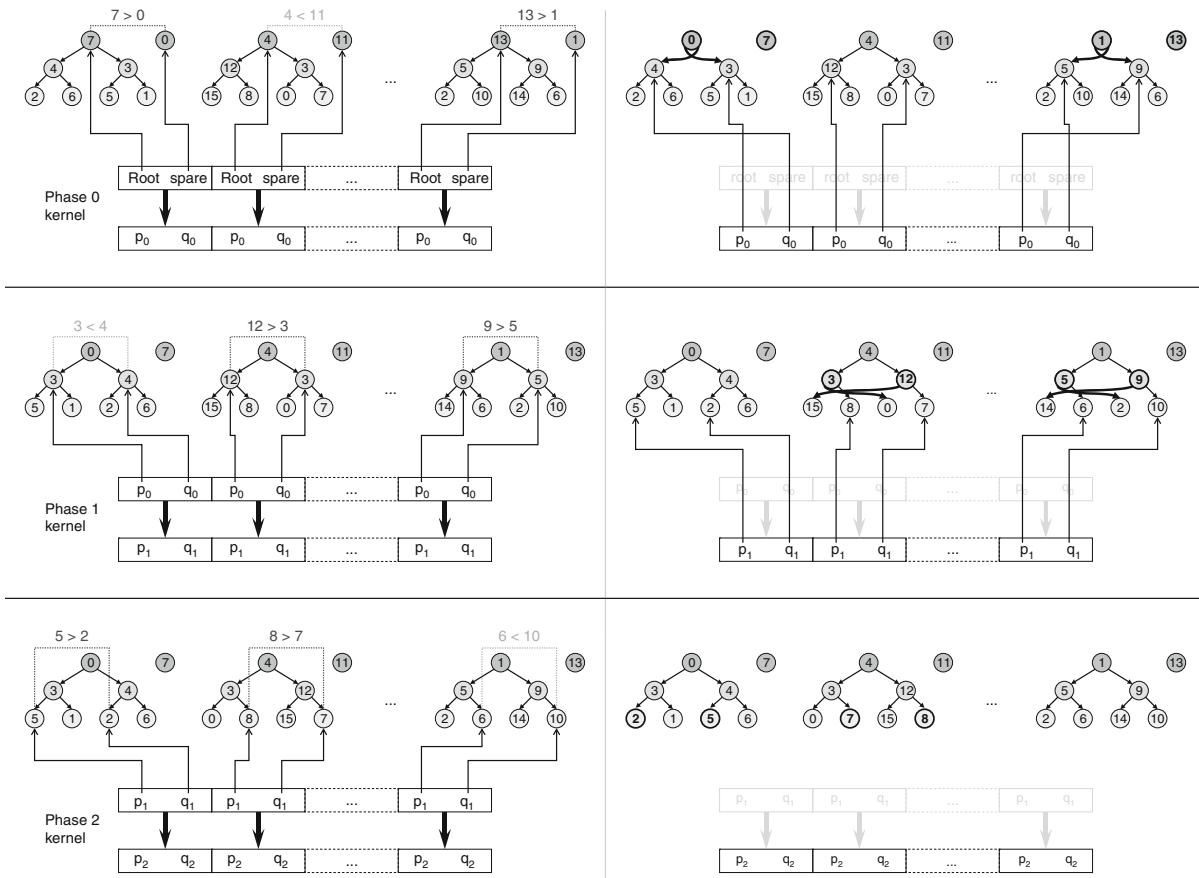
GPU-Specific Details

For the input and output streams, it is best to apply the *ping-pong* technique commonly used in GPU programming: allocate two such streams and alternately use one of them as input and the other one as output stream.

Preconditioning the Input

For merge-based sorting on a PRAM architecture (and assuming $p < n$), it is a common technique to sort *locally*, in a first step, p blocks of n/p values, that is, each processor sorts n/p values using a standard sequential algorithm.

The same technique can be applied here by implementing such a *local sort* as a kernel program. However, since there is no random write access to non-temporary memory from a kernel, the number of values that can be sorted locally by a kernel is restricted by the number of temporary registers.



Bitonic Sorting, Adaptive. Fig. 11 To execute several instances of the adaptive La/Ua construction algorithm in parallel, where each instance operates on a bitonic tree of 2^3 nodes, three phases are required. This figure illustrates the operation of these three phases. On the left, the node pointers contained in the input stream are shown as well as the comparisons performed by the kernel program. On the right, the node pointers written to the output stream are shown as well as the modifications of the child pointers and node values performed by the kernel program according to Algorithm 3

On recent GPUs, the maximum output data size of a kernel is 16×4 bytes. Since usually the input consists of key/pointer pairs, the method starts with a local sort of 8-key/pointer pairs per kernel. For such small numbers of keys, an algorithm with asymptotic complexity of $O(n)$ performs faster than asymptotically optimal algorithms.

After the local sort, a further stream operation converts the resulting sorted subsequences of length 8 pairwise to bitonic trees, each containing 16 nodes. Thereafter, the GPU-ABiSort approach can be applied as described above, starting with $j = 4$.

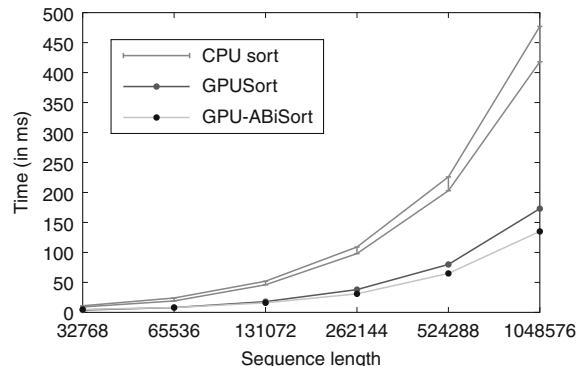
The Last Stage of Each Merge

Adaptive bitonic merging, being a recursive procedure, eventually merges small subsequences, for instance of length 16. For such small subsequences it is better to use a (nonadaptive) bitonic merge implementation that can be executed in a single pass of the whole stream.

Timings

The following experiments were done on arrays consisting of key/pointer pairs, where the key is a uniformly distributed random 32-bit floating point value and the pointer a 4-byte address. Since one can assume (without

n	CPU sort	GPUSort	GPU-ABiSort
32,768	9–11 ms	4 ms	5 ms
65,536	19–24 ms	8 ms	8 ms
131,072	46–52 ms	18 ms	16 ms
262,144	98–109 ms	38 ms	31 ms
524,288	203–226 ms	80 ms	65 ms
1,048,576	418–477 ms	173 ms	135 ms



Bitonic Sorting, Adaptive. Fig. 12 Timings on a GeForce 7800 system. (There are two curves for the CPU sort, so as to visualize that its running time is somewhat data-dependent)

loss of generality) that all pointers in the given array are unique, these can be used as secondary sort keys for the adaptive bitonic merge.

The experiments described in the following compare the implementation of GPU-ABiSort of [11, 12] with sorting on the CPU using the C++ STL sort function (an optimized quicksort implementation) as well as with the (nonadaptive) bitonic sorting network implementation on the GPU by Govindaraju et al., called GPUSort [10].

Contrary to the CPU STL sort, the timings of GPU-ABiSort do not depend very much on the data to be sorted, because the total number of comparisons performed by the adaptive bitonic sorting is not data-dependent.

Figure 12 shows the results of timings performed on a PCI Express bus PC system with an AMD Athlon-64 4200+ CPU and an NVIDIA GeForce 7800 GTX GPU with 256 MB memory. Obviously, the speedup of GPU-ABiSort compared to CPU sorting is 3.1–3.5 for $n \geq 2^{17}$. Furthermore, up to the maximum tested sequence length $n = 2^{20}$ (= 1,048,576), GPU-ABiSort is up to 1.3 times faster than GPUSort, and this speedup is increasing with the sequence length n , as expected.

The timings of the GPU approaches assume that the input data is already stored in GPU memory. When embedding the GPU-based sorting into an otherwise purely CPU-based application, the input data has to be transferred from CPU to GPU memory, and afterwards the output data has to be transferred back to CPU memory. However, the overhead of this transfer is usually negligible compared to the achieved sorting speedup:

according to measurements by [11], the transfer of one million key/pointer pairs from CPU to GPU and back takes in total roughly 20 ms on a PCI Express bus PC.

Conclusion

Adaptive bitonic sorting is not only appealing from a theoretical point of view, but also from a practical one. Unlike other parallel sorting algorithms that exhibit optimal asymptotic complexity too, adaptive bitonic sorting offers low hidden constants in its asymptotic complexity and can be implemented on parallel architectures by a reasonably experienced programmer. The practical implementation of it on a GPU outperforms the implementation of simple bitonic sorting on the same GPU by a factor 1.3, and it is a factor 3 faster than a standard CPU sorting implementation (STL).

Related Entries

- ▶ [AKS Network](#)
- ▶ [Bitonic Sort](#)
- ▶ [Non-Blocking Algorithms](#)
- ▶ [Metrics](#)

Bibliographic Notes and Further Reading

As mentioned in the introduction, this line of research began with the seminal work of Batcher [4] in the late 1960s, who described parallel sorting as a network.

Research of parallel sorting algorithms was reinvigorated in the 1980s, where a number of theoretical questions have been settled [1, 3, 5, 6, 14, 18].

Another wave of research on parallel sorting ensued from the advent of affordable, massively parallel architectures, namely, GPUs, which are, more precisely, streaming architectures. This spurred the development of a number of practical implementations [9, 11–13, 16, 17, 19].

Bibliography

1. Ajtai M, Komlós J, Szemerédi J (1983) An $O(n \log n)$ sorting network. In: Proceedings of the fifteenth annual ACM symposium on theory of computing (STOC '83), New York, NY, pp 1–9
2. Akl SG (1990) Parallel sorting algorithms. Academic, Orlando, FL
3. Azar Y, Vishkin U (1987) Tight comparison bounds on the complexity of parallel sorting. SIAM J Comput 16(3):458–464
4. Batcher KE (1968) Sorting networks and their applications. In: Proceedings of the 1968 Spring joint computer conference (SJCC), Atlanta City, NJ, vol 32, pp 307–314
5. Bilardi G, Nicolau A (1989) Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. SIAM J Comput 18(2):216–228
6. Cole R (1988) Parallel merge sort. SIAM J Comput 17(4):770–785. see Correction in SIAM J. Comput. 22, 1349
7. Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to algorithms, 3rd edn. MIT Press, Cambridge, MA
8. Gibbons A, Rytter W (1988) Efficient parallel algorithms. Cambridge University Press, Cambridge, England
9. Govindaraju NK, Gray J, Kumar R, Manocha D (2006) GPU-TeraSort: high performance graphics coprocessor sorting for large database management. Technical Report MSR-TR-2005-183, Microsoft Research (MSR), December 2005. In: Proceedings of ACM SIGMOD conference, Chicago, IL
10. Govindaraju NK, Raghuvanshi N, Henson M, Manocha D (2005) A cache efficient sorting algorithm for database and data mining computations using graphics processors. Technical report, University of North Carolina, Chapel Hill
11. Graß A, Zachmann G (2006) GPU-ABiSort: optimal parallel sorting on stream architectures. In: Proceedings of the 20th IEEE international parallel and distributed processing symposium (IPDPS), Rhodes Island, Greece, p 45
12. Graß A, Zachmann G (2006) Gpu-abisort: Optimal parallel sorting on stream architectures. Technical Report IfI-06-11, TU Clausthal, Computer Science Department, Clausthal-Zellerfeld, Germany
13. Kipfer P, Westermann R (2005) Improved GPU sorting. In: Pharr M (ed) GPU Gems 2: programming techniques for high-performance graphics and general-purpose computation. Addison-Wesley, Reading, MA, pp 733–746
14. Leighton T (1984) Tight bounds on the complexity of parallel sorting. In: STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing, ACM, New York, NY, USA, pp 71–80
15. Natvig L (1990) Logarithmic time cost optimal parallel sorting is not yet fast in practice! In: Proceedings supercomputing '90, New York, NY, pp 486–494
16. Purcell TJ, Donner C, Cammarano M, Jensen HW, Hanrahan P (2003) Photon mapping on programmable graphics hardware. In: Proceedings of the 2003 annual ACM SIGGRAPH/eurographics conference on graphics hardware (EGGH '03), ACM, New York, pp 41–50
17. Satish N, Harris M, Garland M (2009) Designing efficient sorting algorithms for manycore gpus. In: Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing (IPDPS), IEEE Computer Society, Washington, DC, USA, pp 1–102
18. Schnorr CP, Shamir A (1986) An optimal sorting algorithm for mesh connected computers. In: Proceedings of the eighteenth annual ACM symposium on theory of computing (STOC), ACM, New York, NY, USA, pp 255–263
19. Sintorn E, Assarsson U (2008) Fast parallel gpu-sorting using a hybrid algorithm. J Parallel Distrib Comput 68(10): 1381–1388

BLAS (Basic Linear Algebra Subprograms)

ROBERT VAN DE GEIJN, KAZUSHIGE GOTO
The University of Texas at Austin, Austin, TX, USA

Definition

The Basic Linear Algebra Subprograms (BLAS) are an interface to commonly used fundamental linear algebra operations.

Discussion

Introduction

The BLAS interface supports portable high-performance implementation of applications that are matrix and vector computation intensive. The library or application developer focuses on casting computation in terms of the operations supported by the BLAS, leaving the architecture-specific optimization of that software layer to an expert.

A Motivating Example

The use of the BLAS interface will be illustrated by considering the Cholesky factorization of an $n \times n$ matrix A . When A is Symmetric Positive Definite (a property that guarantees that the algorithm completes), its Cholesky factorization is given by the lower triangular matrix L such that $A = LL^T$.

An algorithm for this operation can be derived as follows: Partition

$$A \rightarrow \left(\begin{array}{c|c} \alpha_{11} & * \\ \hline a_{21} & A_{22} \end{array} \right) \quad \text{and} \quad L \rightarrow \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array} \right),$$

where α_{11} and λ_{11} are scalars, a_{21} and l_{21} are vectors, A_{22} is symmetric, L_{22} is lower triangular, and the $*$ indicates the symmetric part of A that is not used. Then

$$\begin{aligned} \left(\begin{array}{c|c} \alpha_{11} & * \\ \hline a_{21} & A_{22} \end{array} \right) &= \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array} \right) \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array} \right)^T \\ &= \left(\begin{array}{c|c} \lambda_{11}^2 & * \\ \hline \lambda_{11}l_{21} & l_{21}l_{21}^T + L_{22}L_{22}^T \end{array} \right) \end{aligned}$$

This yields the following algorithm for overwriting A with L :

- $\alpha_{11} \leftarrow \sqrt{\alpha_{11}}$.
- $a_{21} \leftarrow a_{21}/\alpha_{11}$.
- $A_{22} \leftarrow -a_{21}a_{21}^T + A_{22}$, updating only the lower triangular part of A_{22} . (This is called a symmetric rank-1 update.)
- Continue by overwriting A_{22} with L_{22} where $A_{22} = L_{22}L_{22}^T$.

A simple code in Fortran is given by

```
do j=1, n
  A( j,j ) = sqrt( A( j,j ) )
  do i=j+1,n
    A( i,j ) = A( i,j ) / A( j,j )
  enddo
  do k=j+1,n
    do i=k,n
      A( i,k ) = A( i,k )
      - A( i,j ) * A( k,j )
    enddo
  enddo
enddo
```

Vector–Vector Operations (Level-1 BLAS)

The first BLAS interface was proposed in the 1970s when vector supercomputers were widely used for computational science. Such computers could achieve near-peak performance as long as the bulk of computation was cast in terms of vector operations and memory was accessed mostly contiguously. This interface is now referred to as the Level-1 BLAS.

Let x and y be vectors of appropriate length and α be scalar. Commonly encountered vector operations are multiplication of a vector by a scalar ($x \leftarrow \alpha x$), inner (dot) product ($\alpha \leftarrow x^T y$), and scaled vector addition ($y \leftarrow \alpha x + y$). This last operation is known as an `axpy`: alpha times x plus y .

The Cholesky factorization, coded in terms of such operations, is given by

```
do j=1, n
  A( j,j ) = sqrt( A( j,j ) )
  call dscal( n-j, 1.0d00/A( j,j ), 
  A( j+1, j ), 1 )
  do k=j+1,n
    call daxpy( n-k+1, -A( k,j ), 
    A(k,j), 1, A( k, k ), 1 );
  enddo
enddo
```

Here

- The first letter in `dscal` and `daxpy` indicates that the computation is with `double` precision numbers.
- The call to `dscal` performs the computation $a_{21} \leftarrow a_{21}/\alpha_{11}$.
- The loop

```
do i=k,n
  A( i,k ) = A( i,k ) - A( i,j )
  * A( k,j )
enddo
```

is replaced by the call

```
call daxpy( n-k+1, -A( k,j ), 
A(k,j), 1, A( k, k ), 1 )
```

If the operations supported by `dscal` and `daxpy` achieve high performance on a target architecture *then* so will the implementation of the Cholesky factorization, since it casts most computation in terms of those operations.

A representative calling sequence for a Level-1 BLAS routine is given by

```
_axpy( n, alpha, x, incx, y, incy )
```

which implements the operation $y = \alpha x + y$. Here

- The “_” indicates the data type. The choices for this first letter are

s	<u>s</u> ingle precision
d	<u>d</u> ouble precision
c	<u>s</u> ingle precision <u>c</u> omplex
z	<u>d</u> ouble precision <u>c</u> omplex

- The operation is identified as axpy: alpha times x plus y.
- n indicates the number of elements in the vectors x and y.
- alpha is the scalar α .
- x and y indicate the memory locations where the first elements of x and y are stored, respectively.
- incx and incy equal the increment by which one has to stride through memory to locate the elements of vectors x and y, respectively.

The following are the most frequently used Level-1 BLAS:

Routine/ Function	Operation
<u>_swap</u>	$x \leftrightarrow y$
<u>_scal</u>	$x \leftarrow \alpha x$
<u>_copy</u>	$y \leftarrow x$
<u>_axpy</u>	$y \leftarrow \alpha x + y$
<u>_dot</u>	$x^T y$
<u>_nrm2</u>	$\ x\ _2$
<u>_asum</u>	$\ \text{re}(x)\ _1 + \ \text{im}(x)\ _1$
<u>i_max</u>	$\min(k) : \text{re}(x_k) + \text{im}(x_k) = \max(\text{re}(x_i) + \text{im}(x_i))$

Matrix–Vector Operations (Level-2 BLAS)

The next level of BLAS supports operations with matrices and vectors. The simplest example of such an operation is the matrix–vector product: $y \leftarrow Ax$ where x and y are vectors and A is a matrix. Another example is the computation $A_{22} = -a_{21}a_{21}^T + A_{22}$ (symmetric rank-1 update) in the Cholesky factorization. This operation can be recoded as

```
do j=1, n
```

```

A( j,j ) = sqrt( A( j,j ) )
call dscal( n-j, 1.0d00 /
A( j,j ), A( j+1, j ), 1 )
call dsyr( 'Lower triangular',
n-j, -1.0d00,
A( j+1,j ), 1, A( j+1,j+1 ), lda )
enddo
```

Here, dsyr is the routine that implements a double precision symmetric rank-1 update. Readability of the code is improved by casting computation in terms of routines that implement the operations that appear in the algorithm: dscal for $a_{21} = a_{21}/\alpha_{11}$ and dsyr for $A_{22} = -a_{21}a_{21}^T + A_{22}$.

The naming convention for Level-2 BLAS routines is given by

```
_XXYY,
```

where

- “_” can take on the values s, d, c, z.
- XX indicates the shape of the matrix:

XX	matrix shape
ge	<u>g</u> eneral (<u>r</u> ectangular)
sy	<u>s</u> ymmetric
he	<u>H</u> ermitian
tr	<u>t</u> riangular

In addition, operations with banded matrices are supported, which we do not discuss here.

- YY indicates the operation to be performed:

YY	matrix shape
mv	<u>m</u> atrix <u>v</u> ector multiplication
sv	<u>s</u> olve <u>v</u> ector
r	<u>r</u> ank-1 update
r2	<u>r</u> ank-2 update

A representative call to a Level-2 BLAS operation is given by

```
dsyr( uplo, n, alpha, x, incx, A,
      lda )
```

which implements the operation $A = \alpha xx^T + A$, updating the lower or upper triangular part of A by choosing uplo as ‘Lower triangular’ or ‘Upper triangular’, respectively. The parameter lda (the

leading dimension of matrix A) indicates the increment by which memory has to be traversed in order to address successive elements in a row of matrix A .

The following table gives the most commonly used Level-2 BLAS operations:

Routine/ Function	Operation
<code>_gemv</code>	general <u>matrix-vector</u> multiplication
<code>_symv</code>	symmetric <u>matrix-vector</u> multiplication
<code>_trmv</code>	triangular <u>matrix-vector</u> multiplication
<code>_trsv</code>	triangular <u>solve vector</u>
<code>_ger</code>	general <u>rank-1 update</u>
<code>_syr</code>	symmetric <u>rank-1 update</u>
<code>_syr2</code>	symmetric <u>rank-2 update</u>

There are also interfaces for operation with banded matrices stored in packed format as well as for operations with Hermitian matrices.

Matrix-Matrix Operations (Level-3 BLAS)

The problem with vector operations and matrix–vector operations is that they perform $O(n)$ computations with $O(n)$ data and $O(n^2)$ computations with $O(n^2)$ data, respectively. This makes it hard, if not impossible, to leverage cache memory now that processing speeds greatly outperform memory speeds, unless the problem size is relatively small (fits in cache memory).

The solution is to cast computation in terms of matrix-matrix operations like matrix–matrix multiplication. Consider again Cholesky factorization. Partition

$$A \rightarrow \left(\begin{array}{c|c} A_{11} & * \\ \hline A_{21} & A_{22} \end{array} \right) \quad \text{and} \quad L \rightarrow \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array} \right),$$

where A_{11} and L_{11} are $n_b \times n_b$ submatrices. Then

$$\begin{aligned} \left(\begin{array}{c|c} A_{11} & * \\ \hline A_{21} & A_{22} \end{array} \right) &= \\ \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array} \right) \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array} \right)^T &= \\ \left(\begin{array}{c|c} L_{11}L_{11}^T & * \\ \hline L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T \end{array} \right) & \end{aligned}$$

This yields the algorithm

- $A_{11} = L_{11}$ where $A_{11} = L_{11}L_{11}^T$ (Cholesky factorization of a smaller matrix).
- $A_{21} = L_{21}$ where $L_{21}L_{11}^T = A_{21}$ (triangular solve with multiple right-hand sides).
- $A_{22} = -L_{21}L_{21}^T + A_{22}$, updating only the lower triangular part of A_{22} (symmetric rank-k update).
- Continue by overwriting A_{22} with L_{22} where $A_{22} = L_{22}L_{22}^T$.

A representative code in Fortran is given by

```
do j=1, n, nb
    jb = min( nb, n-j+1 )
    call chol( jb, A( j, j ), lda )

    call dtrsm( 'Right', 'Lower
                triangular',
                'Transpose',
                'Nonunit diag',
                J-JB+1, JB,
                1.0d00, A( j, j ),
                lda, A( j+jb, j ),
                lda )

    call dsyrk( 'Lower triangular',
                'No transpose',
                J-JB+1, JB,
                -1.0d00,
                A( j+jb, j ), lda,
                1.0d00,
                A( j+jb, j+jb ),
                lda )
enddo
```

Here subroutine `chol` performs a Cholesky factorization; `dtrsm` and `dsyrk` are level-3 BLAS routines:

- The call to `dtrsm` implements $A_{21} \leftarrow L_{21}$ where $L_{21}L_{11}^T = A_{21}$.
- The call to `dsyrk` implements $A_{22} \leftarrow -L_{21}L_{21}^T + A_{22}$.

The bulk of the computation is now cast in terms of matrix–matrix operations which can achieve high performance.

The naming convention for Level-3 BLAS routines are similar to those for the Level-2 BLAS. A representative call to a Level-3 BLAS operation is given by

```
dsyrk( uplo, trans, n, k, alpha, A,
       lda, beta, C, ldc )
```

which implements the operation $C \leftarrow \alpha AA^T + \beta C$ or $C \leftarrow \alpha A^T A + \beta C$ depending on whether `trans` is chosen as ‘No transpose’ or ‘Transpose,’ respectively. It updates the lower or upper triangular part of C depending on whether `uplo` equal ‘Lower triangular’ or ‘Upper triangular,’ respectively. The parameters `lda` and `ldc` are the leading dimensions of arrays A and C , respectively.

The following table gives the most commonly used Level-3 BLAS operations:

Routine/ Function	Operation
<code>_gemm</code>	<u>general matrix-matrix multiplication</u>
<code>_symm</code>	<u>symmetric matrix-matrix multiplication</u>
<code>_trmm</code>	<u>triangular matrix-matrix multiplication</u>
<code>_trsm</code>	<u>triangular solve with multiple right-hand sides</u>
<code>_syrk</code>	<u>symmetric rank-k update</u>
<code>_syr2k</code>	<u>symmetric rank-2k update</u>

Impact on Performance

Figure 1 illustrates the performance benefits that come from using the different levels of BLAS on a typical architecture.

BLAS-Like Interfaces

CBLAS

A C interface for the BLAS, CBLAS, has also been defined to simplify the use of the BLAS from C and C++. The CBLAS support matrices stored in row and column major format.

Libflame

The libflame library that has resulted from the FLAME project encompasses the functionality of the BLAS as well as higher level linear algebra operations. It uses an object-based interface so that a call to a BLAS routine like `_syrk` becomes

```
FLA_Syrk( uplo, trans, alpha, A,
           beta, C )
```

thus hiding many of the dimension and indexing details.

Sparse BLAS

Several efforts were made to define interfaces for BLAS-like operations with sparse matrices. These do not seem to have caught on, possibly because the storage of sparse matrices is much more complex.

Parallel BLAS

Parallelism with BLAS operations can be achieved in a number of ways.

Multithreaded BLAS

On shared-memory architectures multithreaded BLAS are often available. Such implementations achieve parallelism within each BLAS call without need for changing code that is written in terms of the interface. Figure 2 shows the performance of the Cholesky factorization codes when multithreaded BLAS are used on a multi-core architecture.

PBLAS

As part of the ScaLAPACK project, an interface for distributed memory parallel BLAS was proposed, the PBLAS. The goal was to make this interface closely resemble the traditional BLAS. A call to `dsyrk` becomes

```
pdsyrk(uplo, trans, n, k, alpha, A,
        iA, jA, descA, beta, C, iC, jC,
        descC)
```

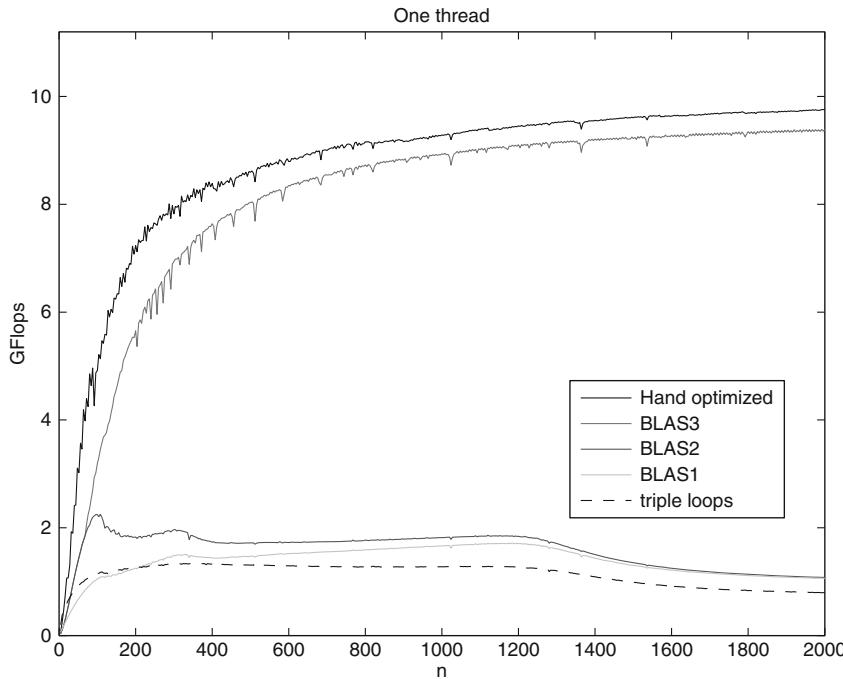
where the new parameters `iA`, `jA`, `descA`, etc., encapsulate information about the submatrix with which to multiply and the distribution to a logical two-dimensional mesh of processing nodes.

PLAPACK

The PLAPACK project provides an alternative to ScaLAPACK. It also provides BLAS for distributed memory architectures, but (like libflame) goes one step further toward encapsulation. The call for parallel symmetric rank-k update becomes

```
PLA_Syrk( uplo, trans, alpha, A,
           beta, C )
```

where all information about the matrices, their distribution, and the storage of local submatrices are encapsulated in the parameters A and C .



BLAS (Basic Linear Algebra Subprograms). Fig. 1 Performance of the different implementations of Cholesky factorization that use different levels of BLAS. The target processor has a peak of 11.2 Gflops (billions of floating point operations per second). BLAS1, BLAS2, and BLAS3 indicate that the bulk of computation was cast in terms of Level-1, -2, or -3 BLAS, respectively

Available Implementations

Many of the software and hardware vendors market high-performance implementations of the BLAS. Examples include IBM's ESSL, Intel's MKL, AMD's ACML, NEC's MathKeisan, and HP's MLIB libraries. Widely used open source implementations include ATLAS and the GotoBLAS. Comparisons of performance of some of these implementations are given in Figs. 3 and 4.

The details about the platform on which the performance data was gathered nor the versions of the libraries that were used are given because architectures and libraries continuously change and therefore which is faster or slower can easily change with the next release of a processor or library.

Related Entries

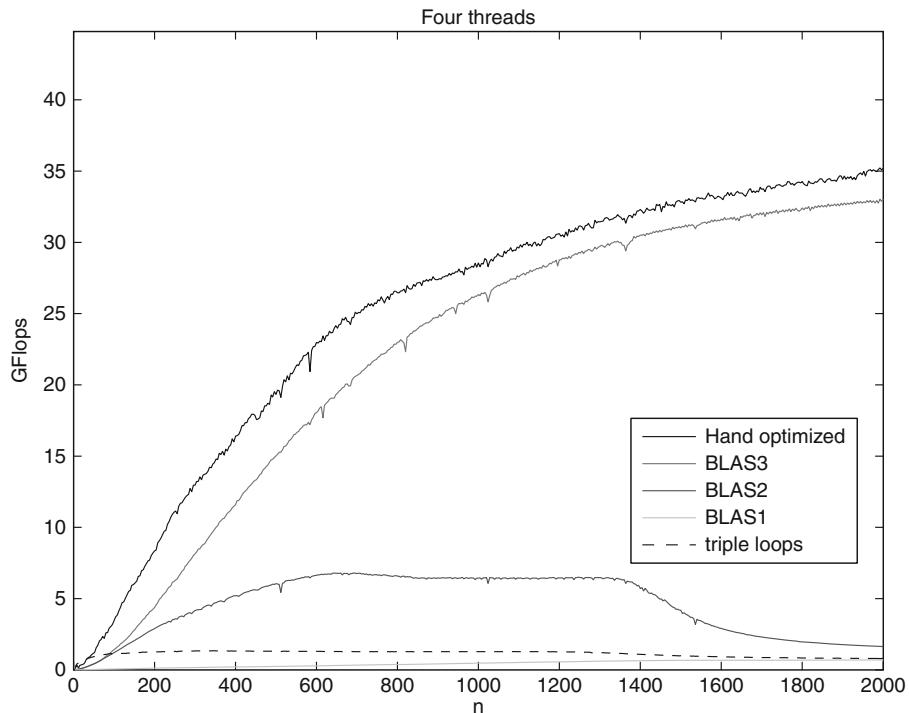
- [ATLAS \(Automatically Tuned Linear Algebra Software\)](#)
- [libflame](#)
- [LAPACK](#)

- [PLAPACK](#)
- [ScaLAPACK](#)

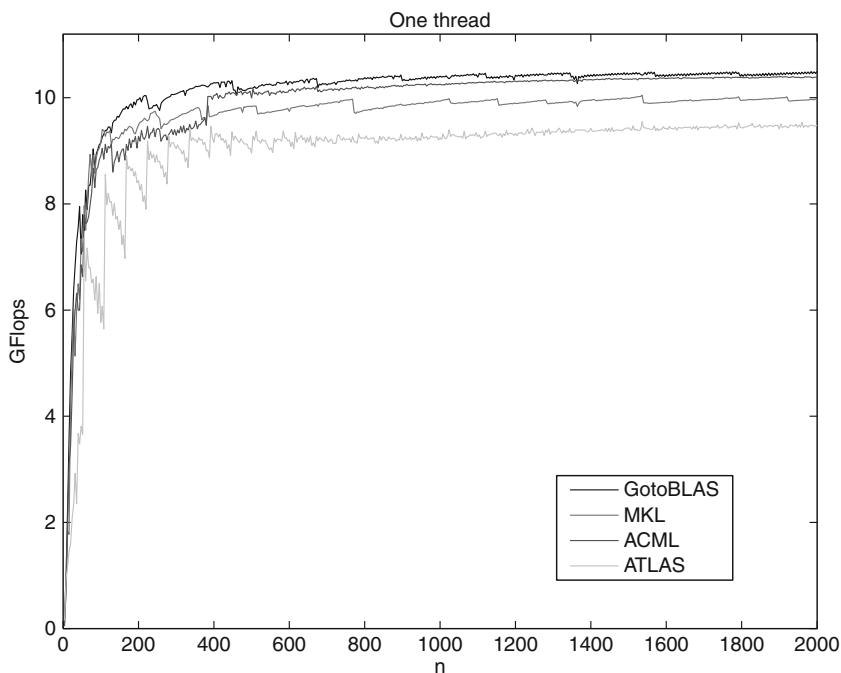
Bibliographic Notes and Further Reading

What came to be called the Level-1 BLAS were first published in 1979, followed by the Level-2 BLAS in 1988 and Level-3 BLAS in 1990 [4, 5, 10].

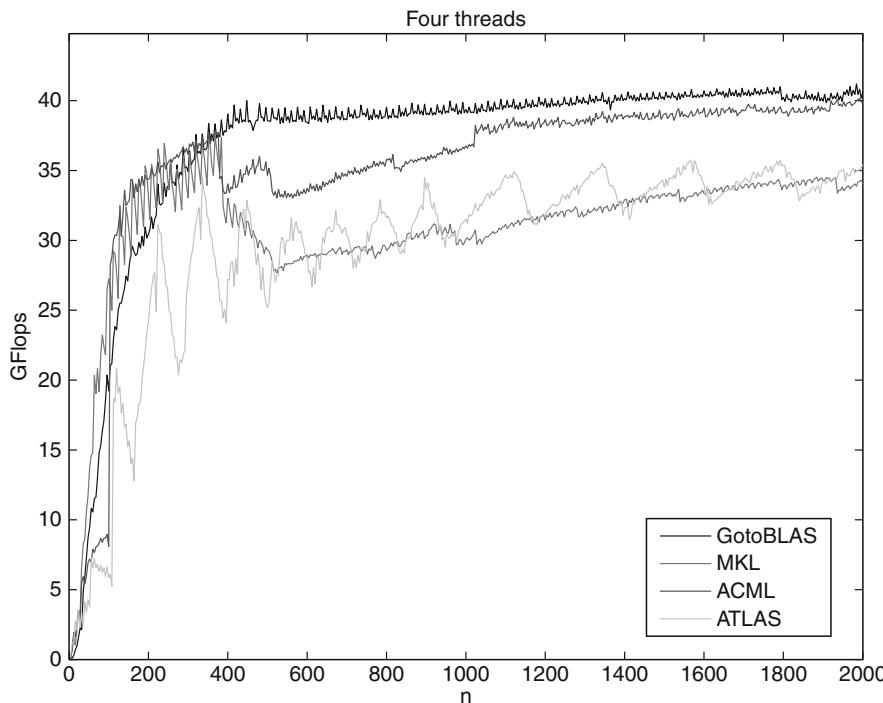
Matrix-matrix multiplication (`_gemm`) is considered the most important operation, since high-performance implementations of the other Level-3 BLAS can be coded in terms of it [9]. Many implementations of `_gemm` are now based on the techniques developed by Kazushige Goto [8]. These techniques extend to the high-performance implementation of other Level-3 BLAS [7] and multithreaded architectures [11]. Practical algorithms for the distributed memory parallel implementation of matrix-matrix multiplication, used by ScaLAPACK and PLAPACK, were first discussed in [1, 12] and for other Level-3 BLAS in [3].



BLAS (Basic Linear Algebra Subprograms). Fig. 2 Performance of the different implementations of Cholesky factorization that use different levels of BLAS, using four threads on architectures with four cores and a peak of 44.8 GFlops



BLAS (Basic Linear Algebra Subprograms). Fig. 3 Performance of different BLAS libraries for matrix–matrix multiplication (dgemm)



BLAS (Basic Linear Algebra Subprograms). Fig. 4 Parallel performance of different BLAS libraries for matrix–matrix multiplication (d_{gemm})

As part of the BLAS Technical Forum, an effort was made in the late 1990s to extend the BLAS interfaces to include additional functionality [2]. Outcomes included the CBLAS interface, which is now widely supported, and an interface for Sparse BLAS [6].

Bibliography

- Agarwal RC, Gustavson F, Zubair M (1994) A high-performance matrix multiplication algorithm on a distributed memory parallel computer using overlapped communication. *IBM J Res Dev* 38(6)
- Blackford LS, Demmel J, Dongarra J, Duff I, Hammarling S, Henry G, Heroux M, Kaufman L, Lumsdaine A, Petitet A, Pozo R, Remington K, Whaley RC (June 2002) An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans Math Softw* 28(2):135–151
- Chtchelkanova A, Gunnels J, Morrow G, Overfelt J, van de Geijn RA (Sept 1997) Parallel implementation of BLAS: general techniques for level-3 BLAS. *Concurrency: Pract Exp* 9(9):837–857
- Dongarra JJ, Du Croz J, Hammarling S, Duff I (March 1990) A set of level-3 basic linear algebra subprograms. *ACM Trans Math Softw* 16(1):1–17
- Dongarra JJ, Du Croz J, Hammarling S, Hanson RJ (March 1998) An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans Math Softw* 14(1):1–17
- Duff IS, Heroux MA, Pozo R (June 2002) An overview of the sparse basic linear algebra subprograms: the new standard from the BLAS technical forum. *ACM Trans Math Softw* 28(2):239–267
- Goto K, van de Geijn R (2008) High-performance implementation of the level-3 BLAS. *ACM Trans Math Softw* 35(1):1–14
- Goto K, van de Geijn RA (2008) Anatomy of high-performance matrix multiplication. *ACM Trans Math Softw* 34(3):1–25
- Kågström B, Ling P, Van Loan C (1998) GEMM-based level-3 BLAS: high performance model implementations and performance evaluation benchmark. *ACM Trans Math Softw* 24(3):268–302
- Lawson CL, Hanson RJ, Kincaid DR, Krogh FT (Sept 1979) Basic linear algebra subprograms for Fortran usage. *ACM Trans Math Softw* 5(3):308–323
- Marker B, Van Zee FG, Goto K, Quintana-Ortí G, van de Geijn RA (2007) Toward scalable matrix multiply on multithreaded architectures. In: Kermarrec A-M, Bougé L, Priol T (eds) *Euro-Par, LNCS 4641*, pp 748–757
- van de Geijn R, Watts J (April 1997) SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Pract Exp* 9(4):255–274

Blocking

► Tiling

Blue CHiP

LAWRENCE SNYDER

University of Washington, Seattle, WA, USA

Synonyms

Blue CHiP project; CHiP architecture; CHiP computer; Configurable, Highly parallel computer; Programmable interconnect computer; Reconfigurable computer

Definition

The CHiP Computer is the Configurable, Highly Parallel architecture, a multiprocessor composed of processor-memory elements in a lattice of programmable switches [1]. Designed in 1980 to exploit Very Large Scale Integration (VLSI), the switches are set under program control to connect and reconnect processors. Though the CHiP architecture was the first use of programmable interconnect, it has since been applied most widely in Field Programmable Gate Arrays (FPGAs). The Blue CHiP Project, the effort to investigate and develop the CHiP architecture, took Carver Mead's "tall, thin man" vision as its methodology. Accordingly, it studied "How best to apply VLSI technology" from five research perspectives: VLSI, architecture, software, algorithms, and theory.

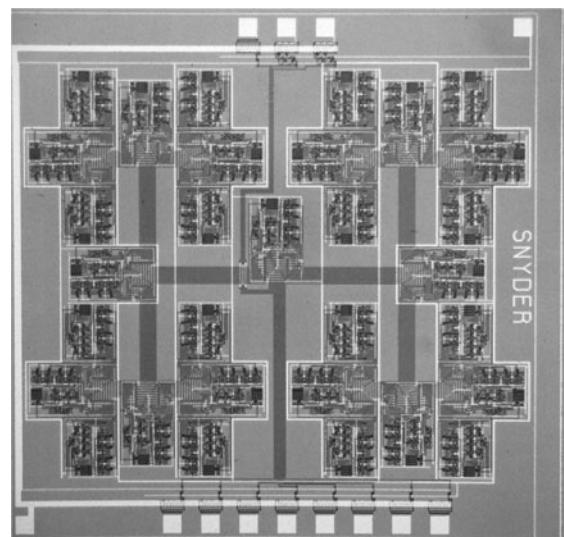
Discussion

Background and Motivation

In the late 1970s as VLSI densities were improving to allow significant functionality on a chip, Caltech's Carver Mead offered a series of short courses at several research universities [2]. He taught the basics of chip design to computer science grad students and faculty. In his course, which required students to do a project fabricated using ARPA's multi-project chip facility, Mead argued that the best way to leverage VLSI technology was for a designer to be a "tall, thin man," a person knowledgeable in the entire chip development stack: electronics, circuits, layout, architecture, algorithms, software, and applications. "Thin" implied that the person had a working knowledge of each level, but might not be an expert. The Blue CHiP project adopted Mead's idea: Use the "tall, thin man" approach to create new ways to apply VLSI technology.

The Blue CHiP Project and the CHiP architecture were strongly influenced by ideas tested in a design included in the MPC-79 multi-project chip, a batch of chip designs created by Mead's students [3]. That design, called the Tree Organized Processor Structure (referenced as AG-4 in the archive [3]), was a height 5 binary tree with field-programmable processors for evaluating Boolean expressions [4]. Though the design used field-effect programmability, the binary tree was embedded directly into the layout; see Fig. 1. Direct embedding seemed unnecessarily restrictive. The lattice structure of the CHiP architecture, developed in Spring, 1980, was to provide a more flexible communication capability.

Because parallel computation was not well understood, because the CHiP computer was an entirely new architecture, because nothing was known about programmable interconnect, and because none of the "soft" parts of the project (algorithms, OS, languages, applications) had ever been considered in the context of configurability, there was plenty of research to do. And it involved so many levels that Mead's "tall, thin man" approach was effectively a requirement. Having secured Office of Naval Research funding for the Blue CHiP Project in Fall, 1980, work began at the start of 1981.



Blue CHiP. Fig. 1 The tree organized processor structure, an antecedent to the CHiP architecture [1]; the root is in the center and its children are to its left and right

The CHiP Architecture Overview and Operation

The focal point of the Blue CHiP project was the Configurable, Highly Parallel (CHiP) computer [5]. The CHiP architecture used a set of processor elements (PEs) – 32-bit processors with random access memory for program and data – embedded in a regular grid, or *lattice*, of wires and programmable switches. Interprocessor communication was implemented by setting the switches under program control to connect PEs with direct, circuit-switched channels. The rationale was simple: Processor design was well understood, so a specific hardware implementation made sense, but the optimal way to connect together parallel processors was not known. Building a dynamically configurable switching fabric that could reconnect processes as the computation progressed permitted optimal communication.

Phases

The CHiP architecture exploited the observation that large computations (worthy of parallel processing) are typically expressed as a sequence of algorithms, or *phases*, that perform the major steps of the computation. Each phase – matrix multiplication, FFT, pivot selection, etc. – is a building block for the overall computation. The phases execute one after another, often being repeated for iterative methods, simulations, approximations, etc. Phases tend to have a specific characteristic communication structure. For example, FFT uses the butterfly graph to connect processors; the Kung-Leiserson systolic array matrix multiplication uses a “hex array” interconnect [MC80]; parallel prefix computations use a complete binary tree, etc. Rather than trying to design a one-size-fits-all interconnect, the CHiP architecture provides the mechanism to program it.

From an OS perspective a parallel computation on the CHiP machine proceeds as follows:

```

Load switch memories in the lattice
with configuration settings for all
phases
Load PE memories with binary object
code for all phases
i = 0
while phases not complete {
    Select the lattice configuration

```

```

        connecting processors as
        required for Phase[i]
        Run Phase[i] to completion
        i=i+1
    }
}
```

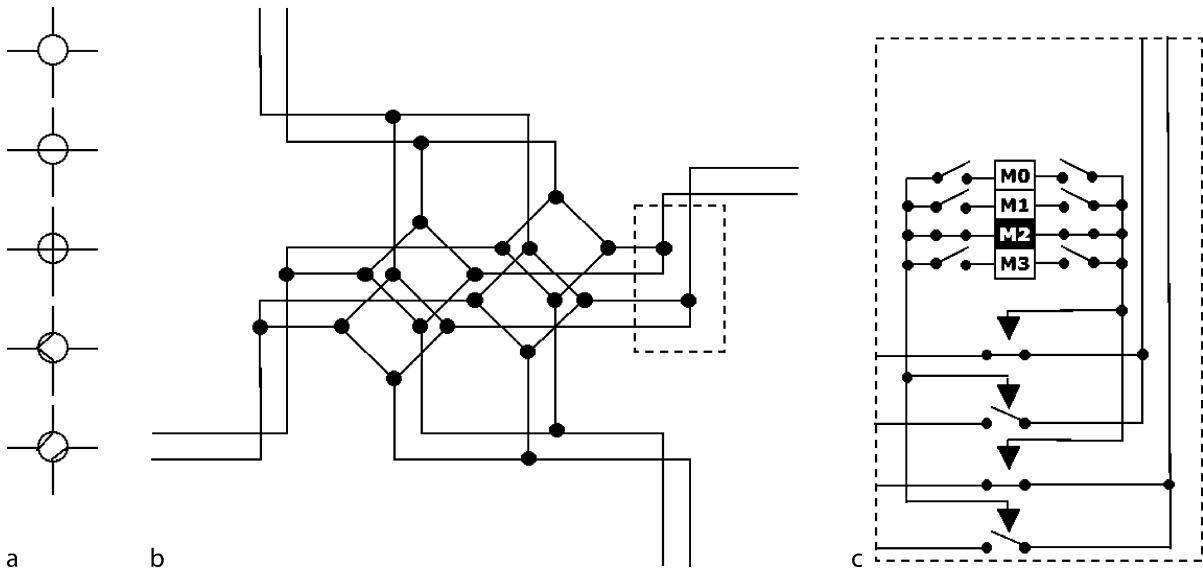
Notice that lattice configurations are typically reused during a program execution.

Configuration

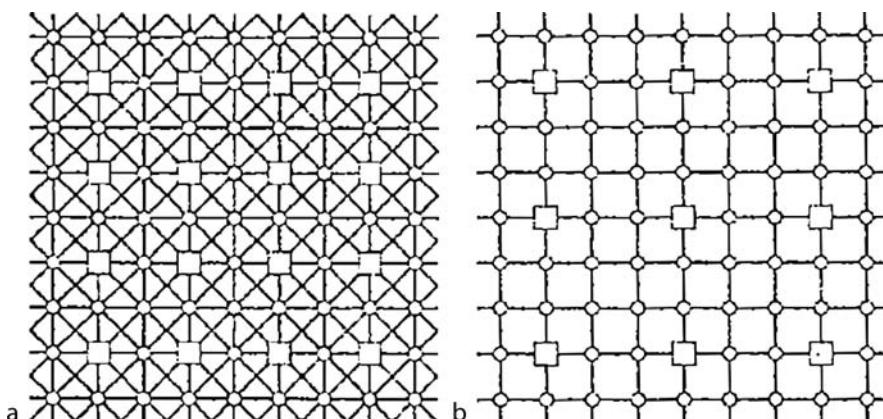
Configuring a communication structure works as follows: The interconnection graph structure, or configuration, is stored in the lattice, and when selected, it remains in effect for the period of that phase of execution, allowing the PEs to communicate and implement the algorithm. PE-to-PE communication is realized by a direct circuit-switched connection [1]. The overall graph structure is realized by setting each switch appropriately.

A switch has circuitry to connect its $2d$ incident edges (bit-parallel data wires), allowing up to d communication paths to cross the switch independently. See [Figure 2](#). A switch has memory to store the settings needed for a set of configurations. Each switch stores its setting for configuration i in its memory location i . The lattice implements configuration i by selecting the i th memory location for all switches.

[Figure 2](#) shows schematic diagrams of a degree 4 switch capable of 2 independent connections, that is, paths that can crossover. Interconnections are programmed graphically, using an iconography shown in [Fig. 2a](#) (and [Fig. 3](#)). [Figure 2b](#) shows the connectivity allowing the wires to connect arbitrarily. Notice that fan-out is possible. The independent connections require separate buses, shown as paired, overlapping diamonds. Higher degree switches (more incident edges) require more buses; wider datapaths (2-wide is shown) require more diamonds. [Figure 2c](#) shows the configuration setting in memory location M2 asserting the connection of the east edge to the right bus. Notice that memory is provided to store the connections for each bus, and that there are three states: disconnected, connect to left, and connect to right. Not shown in the figure is the logic to select a common memory location on all switches, that is, the logic selecting M2, as well as circuitry to insure the quality of the signals.



Blue CHiP. Fig. 2 A schematic diagram for a degree four switch: (a) iconography used in programming interconnection graphs; from top, unconnected, east-west connection, east-west and north-south connections, fan-out connection, two corner-turning connections; (b) wire arrangement for four incident edges (2-bit-parallel wires); the buses implementing the connection are shown as overlapping diamonds; (c) detail for making a connection of the east edge to the right bus per configuration stored in memory location M2



Blue CHiP. Fig. 3 Sample switch lattices in which processors are represented by squares and switches by circles; (a) an 8-degree, single wide lattice; (b) a 4-degree, double wide lattice [6]

As a final detail, notice that connections can cause the order of the wires of a datapath to be flipped. Processor ports contain logic to flip the order of the bits to the opposite of the order it received. When a new configuration comes into effect, each connection sends/receives an lsb 1, revealing whether the bits are true or reversed, allowing the ports to select/deselect the flipping hardware.

Processor Elements

Each PE is a computer, a processor with memory, in which all input and output is accessible via ports. (Certain “edge PEs” also connect to external I/O devices.) A lattice with switches having $2d$ incident edges hosts PEs with $2d$ incident edges, and therefore $2d$ ports. Programs communicate simply by sending and receiving through the ports; since the lattice implements

circuit-switched communication, the values can be sent directly with little “packetizing” beyond a task designation in the receiving processor.

Each PE has its own memory, so in principle each PE can execute its own computation in each phase; that is, the CHiP computer is a MIMD, or multiple-instruction, multiple-data, computer. As phase-based parallel algorithms became better understood, it became clear that PEs typically execute only a few different computations to implement a phase. For example, in parallel-prefix algorithms, which have tree-connected processors, the root, the interior nodes, and the leaves execute slightly different computations. Because these are almost always variations of each other, a single program, multiple data (SPMD) model was adopted.

The Controller

The orchestration of phase execution is performed by an additional processor, a “front end” machine, called the *controller*. The controller loads the configuration settings into the switches, and loads the PEs with their code for each phase. It then steps through the phase execution logic (see “Phases”), configuring the lattice for a phase and executing it to completion; processors use the controller’s network to report completion.

The controller code embodies the highest-level logic of a computation, which tends to be very straightforward. The controller code for the Simple Benchmark [7]

for $m := o$ **to** m_{max} **do**

begin

```
    phase(hydro);
    phase(viscos);
    phase(new_Δt);
    phase(thermo);
    phase(row_solve);
    phase(column_solve);
    phase(energy_bal);
```

end.

illustrates this point and is typical.

Blue CHiP Project Research

The Blue CHiP Project research, following Mead, conducted research in five different topic areas: VLSI, architecture, software, algorithms, and theory. These five topics provide an appropriate structure for the remainder of this entry, when extended with an additional section to summarize the ideas. Though much of

the work was targeted at developing a CHiP computer, related topics were also studied.

VLSI: Programmable Interconnect

Because the CHiP concept was such a substantial departure from conventional computers and the parallel computers that preceded it, initial research focused less on building chips, that is, the VLSI, and more on the overall properties of the lattice that were the key to exploiting VLSI. A simple field-effect programmable switch had been worked out in the summer of 1980, giving the team confidence that fabricating a production switch would be doable when the overall lattice design was complete.

The CHiP computer’s lattice is a regular grid of (data) wires with switches at the cross points and processors inset at regular intervals. The lattice abstraction uses circles for switches and squares for processors. An important study was to understand how a given size lattice hosted various graphs; see Fig. 3 for typical abstractions. The one layer metal processes of the early 1980s caused the issue of lattice bandwidth to be a significant concern.

Lattice Properties Among the key parameters [6] that describe a lattice are the *degree* of the nodes, that is, the number of incident edges and *corridor width*, the number of switches separating two PEs. The architectural question is: What is a sufficiently rich lattice to handle most interconnection graphs? This is a graph embedding question, and though graph embedding was an active research area at the time, asymptotic results were not helpful. Rather, it was important for us to understand graph embeddings for a limited number of nodes, say 256 or fewer, of the sort of graphs arising in phase-based parallel computation.

What sort of communication graphs do parallel algorithms use? Table 1 lists some typical graphs and algorithms that use them; there are other, less familiar graphs. Since lattices permit nonplanar embeddings, that is, wires can cross over, a sufficiently wide corridor lattice of sufficiently high degree switches can embed any graph in our domain. But economy is crucial, especially with VLSI technologies with few layers.

Embeddings It was found that graphs could be embedded in very economical lattices. Examples of two of the less obvious embeddings are shown in Fig. 4. The embedding of the binary tree is an adaptation of the “H-embedding” of a binary tree into the plane. The

shuffle exchange graph for 64 nodes of a single wide lattice is quite involved – shuffle edges are solid, exchange edges are dashed. Shuffle exchange on 256 nodes is not known to be embeddable in a *single* wide lattice, and was assumed to require corridor width of 2. Notice that once a graph has been embedded in a lattice, it can be placed in a library for use by those who prefer to program using the logical communication structure rather than the implementing physical graph.

Techniques Most graphs are easy to layout because most parallel computations use sparse graphs with little complexity. In programming layouts, however, one is often presented with the same problem repeatedly. For example, laying out a torus is easy, but minimizing the

wire length – communication time is proportional to length – takes some technique. See Fig. 5. An interleaving of the row and column PEs of a torus by “folding” at the middle vertically, and then “folding” at the middle horizontally, produces a torus with interleaved processors. Each of a PE’s four neighbors is at most three switches away. Opportunities to use this “trick” of interleaving PEs to shorten long wires arise in other cases. A related idea (Fig. 5b) is to “lace” the communication channels in a corridor to maximize the number of datapaths used.

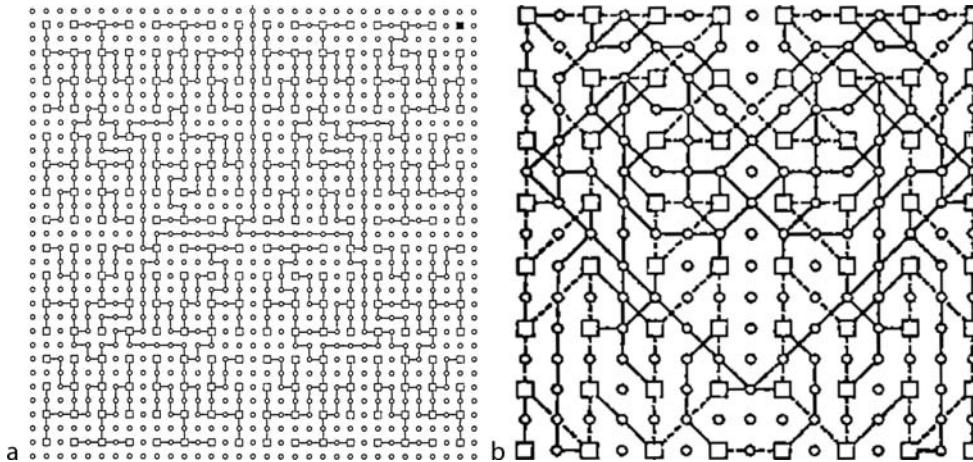
The conclusion from the study of lattice embeddings indicated, generally, that the degree should be 8, and that corridor width is sufficiently valuable that it should be “as wide as possible.” Given the limitations of VLSI technology at the time, only 2 was realistic.

Blue CHiP. Table 1 Communication graph families and parallel algorithms that communicate using those graphs

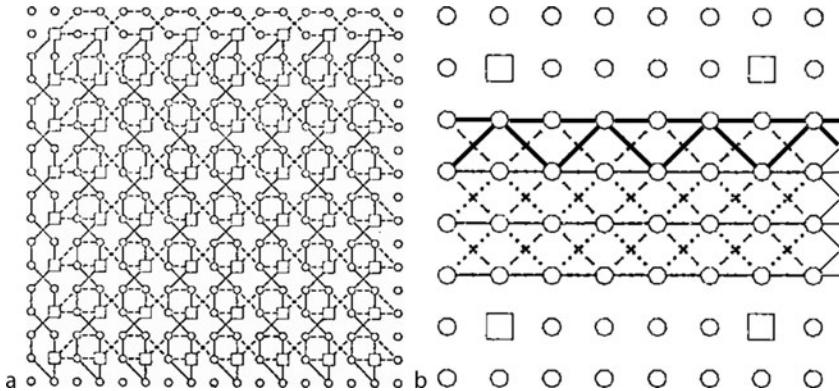
Communication Graph	Parallel Applications
4-degree, 8-degree mesh	Jacobi relaxation for Laplace equations
4-degree, 6-degree mesh	Matrix multiplication, dynamic programming
Binary tree	Aggregation (sum, max), parallel prefix
Toroidal 4-degree, 8-degree mesh	Simulations with periodic boundaries
Butterfly graph	Fast Fourier transform
Shuffle-exchange graph	Sorting

Architecture: The Pringle

When the project began VLSI technology did not yet have the densities necessary to build a prototype CHiP computer beyond single-bit processors; fitting one serious processor on a die was difficult enough, much less a lattice and multiple processors. Nevertheless, the Roadmap of future technology milestones was promising, so the project opted to make a CHiP simulator to use while waiting for technology to improve. The simulated CHiP was called the Pringle [8] in reference to the snack food by that name that also approximates a chip.



Blue CHiP. Fig. 4 Example embeddings; (a) a 255-node binary tree embedded in a 256 node, 4-degree, single wide lattice; (b) a 64-node shuffle exchange graph embedded in a 64 node, 8-degree, single wide lattice



Blue CHiP. Fig. 5 Embedding techniques; (a) a 4-degree torus embedded using “alternating PE positions” to reduce the length of communication paths to 3 switches; solid lines are vertical connections, dashed lines are horizontal connections; (b) “lacing” a width 4 corridor to implement 10 independent datapaths, two of which have been highlighted in bold

Blue CHiP. Table 2 The Pringle datasheet; Pringle was an emulator for the CHiP architecture with an 8-bit datapath

Processor Elements		Switch	
Number of PEs	64	Switch structure	Polled Bus
PE microprocessor	Intel 8031	Switch clock rate	8 MHz
PE datapath width	8-bits	Bus Bandwidth	64 Mb/s
PE floating point chip	Intel 8231	Switch Datapath	8 bits
PE RAM size	2 Kb		
PE EPROM size	4 Kb	Controller	
PE clock rate	12 MHz	Controller microprocessor	Intel 8086

The Pringle was designed to behave like a CHiP computer except that instead of using a lattice for inter-processor communication, it simulated the lattice with a polled bus structure. Of course, this design serializes the CHiP’s parallel communication, but the Pringle could potentially advance 64 separate communications in one polling sweep. The switch controller stored multiple configurations in its memory (up to 11), implementing one per phase. See the datasheet in [Table 2](#).

Two copies of Pringle were eventually built, one for Purdue University and one for the University of Washington. Further data on the Pringle is available in the reports [8].

Software: Poker

The primary software effort was building the Poker Parallel Programming Environment [9]. Poker was targeted at programming the CHiP Computer, which informed

the structure of the programming system substantially. But there were other influences as well.

For historical context the brainstorming and design of Poker began at Purdue in January 1982 in a seminar dedicated to that purpose. At that point, Xerox PARC’s Alto was well established as the model for building a modern workstation computer. It had introduced bit-mapped graphic displays, and though they were still extremely rare, the Blue CHiP Project was committed to using them. The Alto project also supported interactive programming environments such as SmallTalk80 and Interlisp. It was agreed that Poker needed to support interactive programming, too, but the principles and techniques for building windows, menus, etc., were not yet widely understood. Perhaps the most radical aspect of the Alto, which also influenced the project, was dedicating an entire computer to support the work of one user. It was decided to use a VAX 11/780 for that purpose. This decision was greeted with widespread

astonishment among the Purdue CS faculty, because all of the rest of the departmental computing was performed by another time-shared VAX 11/780.

The Poker Parallel Programming Environment opened with a “configuration” screen, called CHiP Params, that asked programmers for the parameters of the CHiP computer they would be programming. Items such as the number of processors, the degree of the switches, etc. were requested. Most of this information would not change in a production setting using a physical computer, but for a research project, it was important.

Programming the CHiP machine involved developing phase programs for specific algorithms such as matrix multiplication, and then assembling these phase pieces into the overall computation [10]. Each phase program requires these components:

- Switch Settings (SS) specification – defines the interconnection graph
- Code Names (CN) specification – assigns a process to each PE, together with parameters
- Port Names (PN) specification – defines the names used by each PE to refer to its neighbors
- I/O Names (IO) specification – defines the external data files used and created by the phase
- Text files – process code written in an imperative language (XX or C)

Not all phases read or write external files, so IO is not always needed; all of the other specifications are required.

Poker emphasized graphic programming and minimized symbolic programming. Programmers used one of several displays – full-screen windows – to program the different types of information describing a computation. Referring to Fig. 6, the windows were: Switch Settings for defining the interconnection graph (Fig. 6a), Code Names for assigning a process (and parametric data) to each processing element (Fig. 6b), Port Names for naming a PE’s neighbors (Fig. 6c), I/O Names for reading and writing to standard in and out, and Trace View (Fig. 6d) for seeing the execution in the same form as used in the “source” code.

In addition to the graphic display windows, a text editor was used to write the process code, whose names are assigned in the Code Names window. Process code was written in a very simple language called XX. (The

letters XX were used until someone could think of an appropriate name for the language, but soon it was simply called Dos Equis.) Later, a C-preprocessor called Poker C was also implemented. One additional facility was a Command Request window for assembling phases.

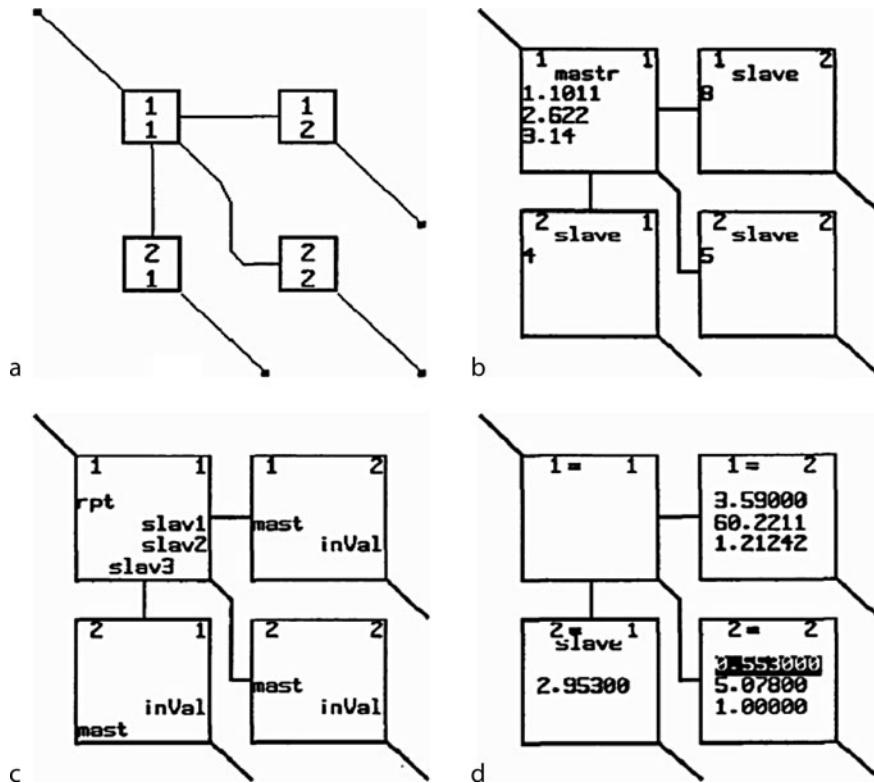
After building a set of phases, programmers needed to assemble them, using the Command Request (CR) interface, into a sequence of phase invocations to solve the problem. This top-level logic is critical because when a phase change occurs – for example, moving from a pivot selection step to an elimination step – the switches in the lattice typically change.

The overall structure of the Poker Environment is shown in Fig. 7 [11]. The main target of CHiP programs was a generic simulator, since production level computations were initially not needed. The Cosmic Cube from Caltech and the Sequent were contemporary parallel machines that were also targeted [12].

Algorithms: Simple

In terms of developing CHiP-appropriate algorithms, the Blue CHiP Project benefited from the wide interest at the time in systolic array computation. Systolic arrays are data parallel algorithms that map easily to a CHiP computer phase [13]. Other parallel algorithms of the time – parallel prefix, nearest neighbor iterative relaxation algorithms, etc., – were also easy to program. So, with basic building blocks available, project personnel split their time between composing these known algorithms (phases) into complete computations, and developing new parallel algorithms. The SIMPLE computation illustrates the former activity.

SIMPLE The SIMPLE computation is, as its name implies, a simple example of a Lagrangian hydrodynamics computation developed by researchers at Livermore National Labs to illustrate the sorts of computations of interest to them that would benefit from performance improvements. The program existed only in Fortran, so the research question became developing a clean parallel solution that eventually became the top-level logic of the earlier Section “The Controller.” Phases had to be defined, interconnection graphs had to be created, data flow between phases had to be managed, and so forth. This common task – converting from sequential to parallel for the CHiP machine – is fully described for SIMPLE in a very nice paper by Gannon and Panetta [7].



Blue CHiP. Fig. 6 Poker screen shots of a phase program for a 4 processor CHiP computer for a master/slave computation; (a) Switch Settings (1,1 is the master), editing mode for SS is shown in Fig. 4a; (b) Code Names; (c) Port Names; and (d) Trace View midway through the (interpreted) computation

New Algorithms Among the new algorithms were image-processing computations, since they were thought to be a domain benefiting from CHiP-style parallelism [14]. One particularly elegant connected components counting algorithm illustrates the style.

Using a 4-degree version of a morphological transformation due to Levialdi, a bit array is modified using two transformations applied at all positions,



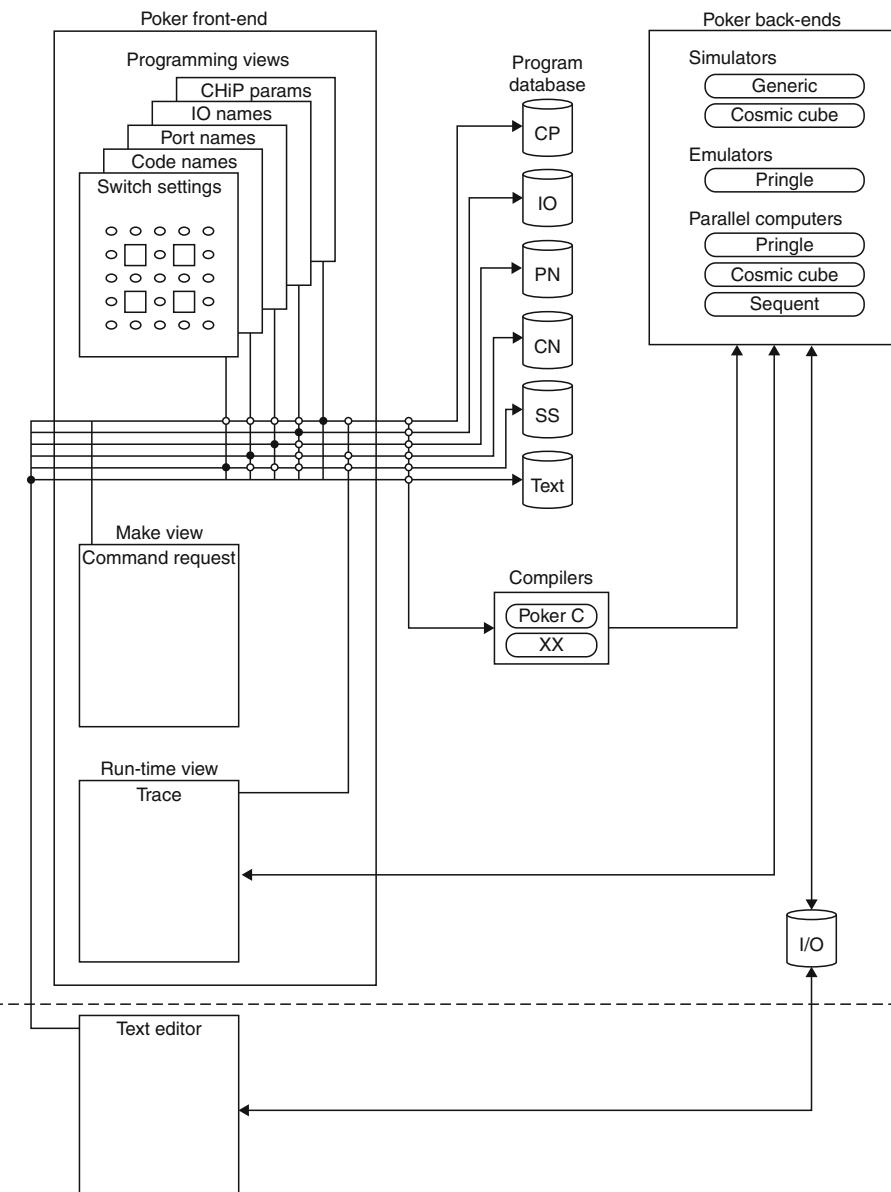
producing a series of morphed arrays. In words, the first rule specifies that a 1-bit adjacent to 0-bits to its north, northwest, and west becomes a 0-bit in the next generation; the second rule specifies that a 0-bit adjacent to 1-bits to its north and west becomes a 1-bit in the next generation; all other bits are unchanged; an isolated bit

that is about to disappear due to the first rule should be counted as a component. See Fig. 8.

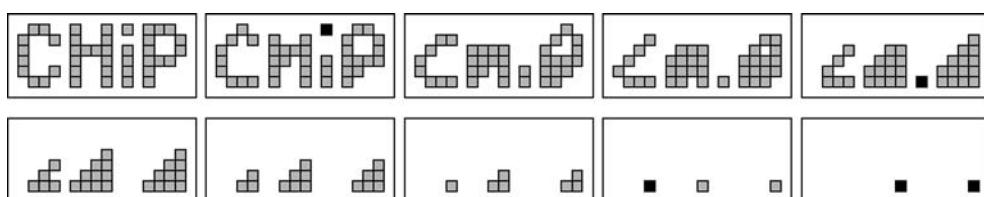
The CHiP phase for the counting connected components computation assigns a rectangular subarray of pixels to each processor, and uses a 4-degree mesh interconnection. PEs exchange bits around the edge of their subarray with adjacent neighbors, and then apply the rules to create the next iteration, repeating. As is typical the controller is not involved in the phase until all processors signal that they have no more bits left in their subarray, at which point the phase ends.

Theory: Graphs

Though the project emphasized the practical task of exploiting VLSI, a few theorems were proved, too. Two research threads are of interest. The first concerned properties of algorithms to convert data-driven programs to loop programs. Data-driven is an asynchronous communication protocol where, for example,



Blue CHiP. Fig. 7 Structure of the Poker parallel programming environment. The “front end” was the portion visible in the graphic user interface; the “back ends” were the “platforms” on which a Poker program could be “run”



Blue CHiP. Fig. 8 The sequence of ten-bit arrays produced using Levialdi’s transformation; pixels are blackened when counted. The algorithm “melts” a component to the lower right corner of its bounding box, where it is counted

reads to a port prior to data arrival stall. The CHiP machine was data driven, but the protocol carries overhead. The question was whether data-driven programs could be more synchronous. The theorems concerned properties of an algorithm to convert from data-driven to synchronous; the results were reported [15] and implemented in Poker, but not heavily used.

The second thread turned out to be rather long. It began with an analysis of yield in VLSI chip fabrication. Called the Tile Salvage Problem, the analysis concerned how the chips on a wafer, some of which had tested faulty, could be grouped into large working blocks [16]. The solution found that matching working pairs is in polynomial time, but finding 2×2 working blocks is NP hard; an optimal-within-2 approximation was developed for 2×2 blocks. When the work was shown to Tom Leighton, he found closely related planar graph matching questions, and added/strengthened some results. He told Peter Shor, who did the same. And finally, David Johnson made still more improvements. The work was finally published as Generalized Planar Matching [17].

Summary

The Blue CHiP Project, which ran for six years, produced a substantial amount of research, most of which is not reviewed here. Much of the research was integrated across multiple topics by applying Mead's "tall, thin man" approach. The results in the hardware domain included a field programmable switch, a programmable communication fabric (lattice), an architecture, wafer-scale integration studies, and a hardware emulator (Pringle) for the machine. The results in the software domain included the Poker Parallel Programming Environment that was built on a graphic workstation, communication graph layouts, programs for the machine, and new parallel algorithms. Related research included studies in wafer scale integration, both theory and design, as well as the extension of the CHiP approach to signal processing and other domains. The work was never directly applied because the technology of the day was not sufficiently advanced; 25 years after the conception of the CHiP architecture, however, the era of multiple-cores-per-chip offers suitable technology.

Apart from the nascent state of the technology, an important conclusion of the project was that the problem in parallel computing is not a hardware problem,

but a software problem. The two key challenges were plain:

- Performance – developing highly parallel computations that *exploit locality*,
- Portability – expressing parallel computations at a high enough level of abstraction that a compiler can target to any MIMD machine.

The two challenges are enormous, and largely remain unsolved into the twenty-first century. Regarding the first, exploiting locality was a serious concern for the CHiP architecture with its tight VLSI connection, making us very sensitive to the issue. But it was also clear that locality is always essential in parallelism, and valuable for all computation. Regarding the second, it was clear in retrospect that the project's programming and algorithm development were very tightly coupled to the machine, as were other projects of the day. Whereas CHiP computations could usually be hosted effectively on other computers, the converse was not true. Shouldn't all code be machine independent? The issue in both cases concerned the parallel programming model.

These conclusions were embodied in a paper known as the "Type Architecture" paper [18]. (Given the many ways "type" is used in computer science, it was not a good name; in this case it meant "characteristic form" as in *type species* in biology.) The paper, among other things, predicts the rise of message passing programming, and criticizes contemporary programming approaches. Most importantly, it defines a machine model – a generic parallel machine – called the CTA. This machine plays the same role that the RAM or von Neumann machine plays in sequential computing. The CTA is visible today in applications ranging from message passing programming to LogP analysis.

Related Entries

- ▶ [CSP \(Communicating Sequential Processes\)](#)
- ▶ [Graph Algorithms](#)
- ▶ [Networks, Direct](#)
- ▶ [Reconfigurable Computer](#)
- ▶ [Routing \(Including Deadlock Avoidance\)](#)
- ▶ [Systolic Arrays](#)
- ▶ [Universality in VLSI Computation](#)
- ▶ [VLSI Computation](#)

Bibliographic Notes and Additional Reading

The focal point of the project, the machine, and its software were strongly influenced by VLSI technology, but the technology was not yet ready for a direct application of the approach; it would take until roughly 2005. The ideas that the project developed – phase-based parallelism, high level parallel language, emphasis on locality, emphasis on data parallelism, etc. – turned out to drive the follow-on research much more than VLSI. Indeed, there were several follow-on efforts to develop high performance, machine independent parallel languages [19], and eventually, ZPL [20]. The problem has not yet been completely solved, but the Chapel [21] language is the latest to embody these ideas.

Bibliography

1. Snyder L (1982) Introduction to the configurable, highly parallel computer. Computer 15(1):47–56, Jan 1982
2. Conway L (1981) The MPC adventures, Xerox PARC Tech. Report VLSI-81-2; also published at <http://ai.eecs.umich.edu/people/conway/VLSI/MPCAdv/MPCAdv.html>
3. Conway L MPC79: a large-scale demonstration of a new way to create systems in silicon, <http://ai.eecs.umich.edu/people/conway/VLSI/MPC79/MPC79Report.pdf>
4. Snyder L (1980) Tree organized processor structure, A VLSI parallel processor design, Yale University Technical Report DCS/TR176
5. Snyder L (1980) Introduction to the configurable, highly parallel computer, Technical Report CSD-TR-351, Purdue University, Nov 1980
6. Snyder L (1981) Programming processor interconnection structures, Technical Report CSD-TR-381, Purdue University, Oct 1981
7. Gannon DB, Panetta J (1986) Restructuring SIMPLE for the CHiP architecture. Parallel Computation 3(4):305–326
8. Kapauan AA, Field JT, Gannon D, Snyder L (1984) The Pringle parallel computer. Proceedings of the 11th international symposium on computer architecture, IEEE, New York, pp 12–20
9. Snyder L (1984) Parallel programming and the Poker programming environment. Computer 17(7):27–36, July 1984
10. Snyder L (1982) The Poker (1.0) programmer's guide, Purdue University Technical Report TR-434, Dec 1982
11. Notkin D, Snyder L, Socha D, Bailey ML, Forstall B, Gates K, Greenlaw R, Griswold WG, Holman TJ, Korry R, Lasswell G, Mitchell R, Nelson PA (1988) Experiences with poker. Proceedings of the ACM/SIGPLAN conference on parallel programming: experience with applications, languages and systems
12. Snyder L, Socha D (1986) Poker on the cosmic cube: the first retargetable parallel programming language and environment. Proceedings of the international conference on parallel processing, Los Alamitos
13. Kung HT, Leiserson CE (1980) Algorithms for VLSI processor arrays. In: Mead C, Conway L (eds) Introduction to VLSI systems, Addison-Wesley, Reading
14. Cypher RE, Sanz JLC, Snyder L (1990) Algorithms for image component labeling on SIMD mesh connected computers. IEEE Trans Comput 39(2):276–281
15. Cuny JE, Snyder L (1983) Compilation of data-driven programs for synchronous execution. Proceedings of the tenth ACM symposium on the principles of programming languages, Austin, pp 197–202
16. Berman F, Leighton FT, Snyder L (1981) Optimal tile salvage, Purdue University Technical Report TR-396, Jan 1981
17. Berman F, Johnson D, Leighton T, Shor P, Snyder L (1990) Generalized planar matching. J Algorithms 11:153–184
18. Snyder L (1986) Type architectures, shared memory, and the corollary of modest potential, Annual review of computer science, vol 1, Annual Reviews, Palo Alto
19. Lin C (1992) The portability of parallel programs across MIMD computers, PhD Dissertation, University of Washington
20. Chamberlain B, Choi S-E, Lewis E, Lin C, Snyder L and Weathersby W (1998) The case for high-level parallel programming in ZPL. IEEE Comput Sci Eng 5(3):76–86, July–Sept 1998
21. Chamberlain BL, Callahan D, Zima HP (2007) Parallel programmability and the Chapel language. Int J High Perform Comput Appl 21(3):291–312, Aug 2007
22. Snyder L (1981) Overview of the CHiP computer. In: Gray JP (ed) VLSI 81, Academic, London, pp 237–246

Blue CHiP Project

► Blue CHiP

Blue Gene/L

► IBM Blue Gene Supercomputer

Blue Gene/P

► IBM Blue Gene Supercomputer

Blue Gene/Q

► IBM Blue Gene Supercomputer

Branch Predictors

ANDRÉ SEZNEC
IRISA/INRIA, Rennes, Rennes, France

Definition

The branch predictor is a hardware mechanism that predicts the address of the instruction following the branch. For respecting the sequential semantic of a program, the instructions should be fetched, decoded, executed, and completed in the order of the program. This would lead to quite slow execution. Modern processors implement many hardware mechanisms to execute concurrently several instructions while still respecting the sequential semantic. Branch instructions cause a particular burden since their result is needed to begin the execution of the subsequent instructions. To avoid stalling the processor execution on every branch, branch predictors are implemented in hardware. The branch predictor predicts the address of the instruction following the branch.

Discussion

Introduction

Most instruction set architectures have a sequential semantic. However, most processors implement pipelining, instruction level parallelism, and out-of-order execution. Therefore on state-of-the-art processors, when an instruction completes, more than one hundred of subsequent instructions may already be in progress in the processor pipeline. Enforcing the semantic of a branch is therefore a major issue: The address of the instruction following a branch instruction is normally unknown before the branch completes. Control flow instructions are quite frequent, up to 30% in some applications. Therefore to avoid stalling the issuing of subsequent instructions until the branch completes, microarchitects invented *branch prediction*,

i.e., the address of the instruction following a branch B is predicted. Then the instructions following the branch can speculatively progress in the processor pipeline without waiting for the completion of the execution of branch B.

Anticipating the address of the next instruction to be executed was recognized as an important issue very early in the computer industry back in the late 1950s. However, the concept of branch prediction was really introduced around 1980 by Smith in [21]. On the occurrence of a branch, the effective information that must be predicted is the address of the next instruction. However, in practice several different informations are predicted in order to predict the address of the next instruction. First of all, it is impossible to know that an instruction is a branch before it has been decoded, that is the branch nature of the instruction must be known or predicted before fetching the next instruction. Second, on taken branches, the branch target is unknown at instruction fetch time, that is the potential target of the branch must be predicted. Third, most branches are conditional, the direction of the branch taken or not-taken must be predicted.

It is important to identify that not all information is of equal importance for performance. Failing to predict that an instruction is a branch means that instructions are fetched in sequence until the branch instruction is decoded. Since decoding is performed early in the pipeline, the instruction fetch stream can be repaired very quickly. Likewise, failing to predict the target of the direct branch is not very dramatic. The effective target of a direct branch can be computed from the instruction codeop and its address, thus the branch can be computed very early in the pipeline — generally, it becomes available at the end of the decode stage. On the other hand, the direction of a conditional branch and the target of an indirect branch are only known when the instruction has been executed, i.e., very late in the pipeline, thus potentially generating very long misprediction penalties, sometimes 30 or 50 cycles.

Since in many applications most branches are conditional and the penalty on a direction misprediction is high, when one refers to branch prediction, one generally refers to predicting directions of conditional branches. Therefore, most of this article is dedicated to conditional branch predictions.

General Hardware Branch Prediction Principle

Some instruction sets have included some software hints to help branch prediction. Hints like “likely taken” and “likely not-taken” have been added to the encoding of the branch instruction. These hints can be inserted by the compilers based on application knowledge, e.g., a loop branch is likely to be taken, or on profiling information. However, the most efficient schemes are essentially hardware and do not rely on any instruction set support.

The general principle that has been used in hardware branch prediction scheme is to predict that the behavior of the branch to be executed will be a replay of its past behavior. Therefore, hardware branch predictors are based on memorization of the past behavior of the branches and some limited hardware logic.

Predicting Branch Natures and Direct Branch Targets

For a given static instruction, its branch nature (is it a branch or not) remains unchanged all along the program life, apart in case of self-modifying code. This stands also for targets of direct branches. The Branch Target Buffer, or BTB [11], is a special cache which aims at predicting whether an instruction is a branch and its potential target. At fetch time, the BTB is checked with the instruction Program Counter. On a hit, the instruction is predicted to be a branch and the address stored in the hitting line of the BTB is assessed to be the potential target of the branch. The nature of the branch (unconditional, conditional, return, indirect jumps) is also read from the BTB.

A branch may miss the BTB, e.g., on its first execution or after its eviction on a conflict miss. In this case, the branch is written in the BTB: its kind and its target are stored in association with its Program Counter. Note that on a BTB hit, the target may be mispredicted on indirect branches and returns. This will shortly be presented in section on “Predicting Indirect Branch Targets”.

Predicting Conditional Branch Directions

Most branches are conditional branches and the penalty on mispredicting the direction of a conditional branch is really high on modern processors. Therefore, predicting the direction of conditional branches has received

a lot of attention from the research community in the 1990s and the early 2000s.

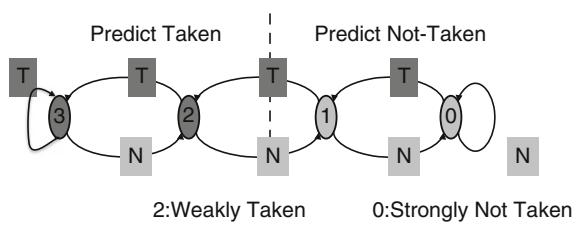
B

PC-Based Prediction Schemes

It is natural to predict a branch based on its program counter. When introducing conditional branch prediction [21], Smith immediately introduced two schemes that capture the most frequent behaviors of the conditional branches.

Since most branches in a program tend to be biased toward taken or not-taken, the simplest scheme consisting to predict that a branch will follow the same direction than the last time it has been executed is quite natural. Hardware implementation of this simple scheme necessitates to store only one bit per branch. When a conditional branch is executed, its direction is recorded, e.g., in the BTB along with the branch target. A single bit is used: 0 encodes not-taken and 1 encodes taken. The next time the branch is fetched, the direction stored in the BTB is used as a prediction. This scheme is surprisingly efficient on most programs often achieving accuracy higher than 90%. However, for branches that privilege one direction and from time to time branch in the other direction, this 1-bit scheme tends to predict the wrong direction twice in a row. This is the case for instance, for loop branches which are taken except the last iteration. The 1-bit scheme fails to predict the first iteration and the last iteration.

Smith [21] proposed a slightly more complex scheme based on a saturated 2-bit counter automaton (Fig. 1): The counter is incremented on a taken branch and decremented on a not-taken branch. The most significant bit of the counter is used as the prediction. On branches exhibiting a strong bias, the 2-bit scheme avoids to encounter two successive mispredictions after one occurrence of the non-bias direction. This 2-bit



Branch Predictors. Fig. 1 A 2-bit counter automaton

counter predictor is often referred to as a 2-bit bimodal predictor.

History-Based Conditional Branch Prediction Schemes

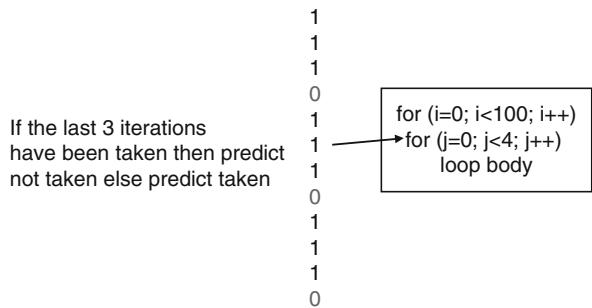
In the early 1990s, branch prediction accuracy became an important issue. The performance of superscalar processors is improved by any reduction of branch misprediction rate. The 1-bit and 2-bit prediction schemes use a very limited history of the behavior of a program to predict the outcome direction of a branch. Yeh and Patt [24] and Pan and So [14] proposed to use more information on the passed behavior of the program, trying to better isolate the context in which the branch is executed.

Two families of predictors were defined. Local history predictors use only the past behavior of the program on the particular branch. Global history predictors use the past behavior of the whole program on all branches.

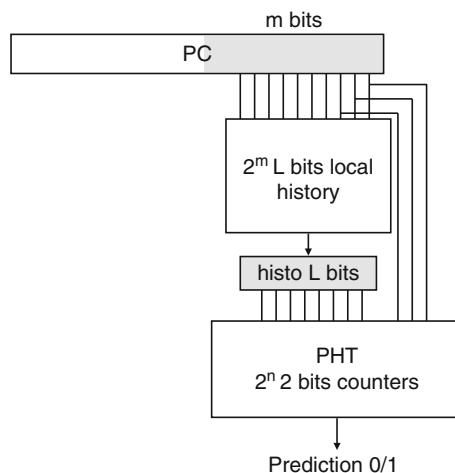
Local History Predictors

Using the past behavior of the branch to be predicted appears attractive. For instance, on the loop nest illustrated on Fig. 2, the number of iterations in the inner loop is fixed (4). When the outcome of the last three occurrences of the branch are known, the outcome of the present occurrence is completely determined: If the last three occurrences of the branch were taken then the current occurrence is the last iteration, therefore is not-taken otherwise the branch is taken. Local history is not limited to capturing the behavior of branches with fixed number of iterations. For instance, it can capture the behavior of any branch exhibiting a periodic behavior as long as the history length is equal or longer than the period.

A local history predictor can be implemented as illustrated Fig. 3. For each branch, the history of its past behavior is recorded as a vector of bits of length L and stored in a local history table. The branch program counter is used as an index to read the local history table. Then the branch history is associated with the program counter to read the prediction table. The prediction table entries are generally saturated 2-bit counters. The history vector must be adapted on each branch occurrence and stored back in the local history table.



Branch Predictors. Fig. 2 A loop with four iterations



Branch Predictors. Fig. 3 A local history predictor

Effective implementation of local history predictors is quite difficult. First the prediction requires to chain two successive table reads, thus creating a difficult latency issue for providing prediction in time for use. Second, in aggressive superscalar processors, several branches (sometimes tens) are progressing speculatively in the pipeline. Several instances of the same branch could have been speculatively fetched and predicted before the first occurrence is finally committed. The branch prediction should be executed using the speculative history. Thus, maintaining correct speculative history for local history predictors is very complex for wide-issue superscalar processors.

Global History Predictors

The outcome of a branch is often correlated with the outcome of branches that have been recently executed.

For instance, in the example illustrated in Fig. 4, the outcome of the third branch is completely determined by the outcome of the first two branches.

First generation global history predictors such as GAs [25] or gshare [12] (Fig. 5) are associating with a fixed length global history vector and the program counter of the branch to index a prediction table. These predictors were shown to suffer from two antagonistic phenomena. First, it was shown that using a very long history is sometimes needed to capture correlation. Second, using a long history results in possible destructive conflicts on a limited size prediction table.

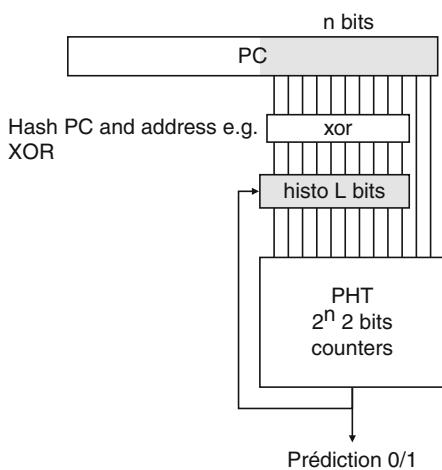
A large body of research in the mid-1990s was dedicated to reduce the impact of these destructive conflicts or aliasing [4, 10, 13, 22].

By the end of the 1990s, these dealiased global history predictors were known to be more accurate at equal storage complexity than local predictors.

Path or global history predictor: In some cases, the global conditional branch history vector does not uniquely determine the instruction path that leads to a particular branch; e.g., an indirect branch or a return

```
B1: if cond1 then ..
B2: if cond2 then ..
B3: if cond1 and cond 2 then ..
```

Branch Predictors. Fig. 4 Branch correlation: outcome of branch 3 is uniquely determined by the outcomes of branches 1 and 2



Branch Predictors. Fig. 5 The gshare predictor

may have occurred and two paths can be represented by the same global history vector. The path vector combining all the addresses of the last control flow instructions that lead to a branch is unique. Using the path history instead of the global history vector generally results in a slightly higher accuracy [15].

Hybrid Predictors

Global history predictors and local history predictors were shown to capture different branch behaviors. In 1993, McFarling [12] proposed to combine several predictors to improve their accuracy. Even combining a global history predictor with a simple bimodal predictor was shown to provide enhanced prediction accuracy.

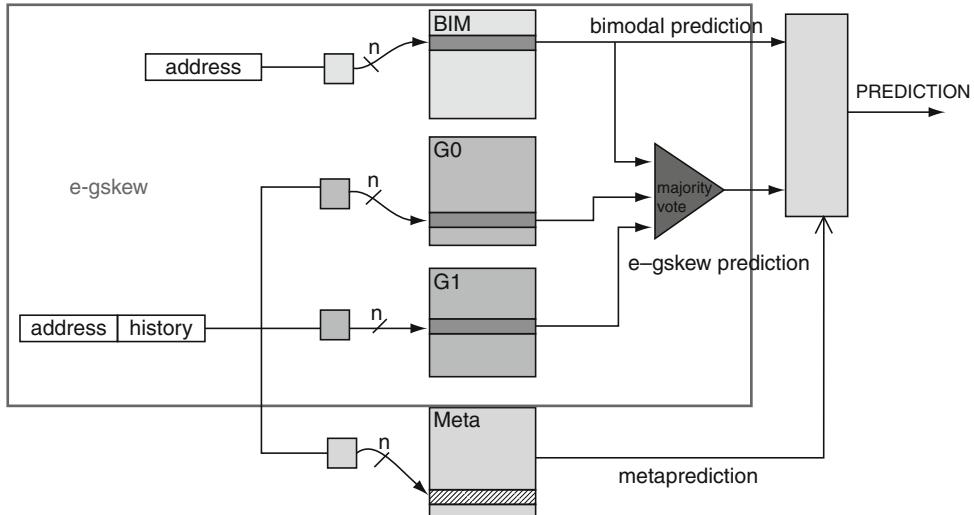
The first propositions of hybrid predictors were to use a metapredictor to select a prediction. The metapredictor is also indexed with the program counter and the branch history. The metapredictor learns which predictor component is more accurate. The *2bc-gskew* predictor 6 proposed for the cancelled Compaq EV8 processor [2] leveraged hybrid prediction to combine several global history predictors including a majority vote *gskew* predictor with different history lengths. However, a metapredictor is not a cost-effective solution to select among more than two predictions [5].

Toward Using Very Long Global History

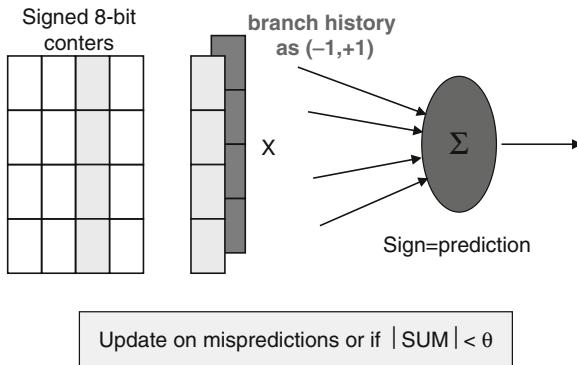
While some applications will be very accurately predicted using a short branch history length, limit studies were showing that some benchmarks would benefit from using very long history in 100s of bit range.

In 2001, Jimenez and Lin [7] proposed to use neural nets inspired techniques to combine predictions. On perceptron predictors, the prediction is computed as the sign of a dot-product of a vector of signed prediction counters read in parallel by the branch history vector (Fig. 7). Perceptron predictors allow to use very long global history vectors, e.g., 50 bits, but suffer from a long latency prediction computation and require huge predictor table volumes.

Building on top of the neural net inspired predictors, the GEometric History Length or GEHL predictor (Fig. 8) was proposed in 2004 [15]. This predictor combines a few global history predictors (typically from 5



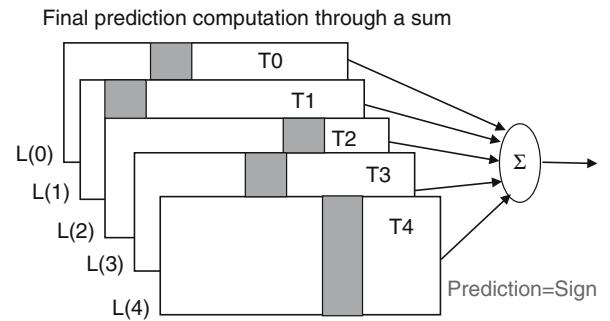
Branch Predictors. Fig. 6 The 2bc-gskew predictor



Branch Predictors. Fig. 7 The perceptron predictor

to 8) indexed with different history lengths. The prediction is computed as the sign of the sum of the read predictions. The set of history lengths forms a geometric series, for instance, 0, 2, 4, 8, 16, 32, 64, 128. The use of such a geometric series allows to concentrate most of the storage budget on short histories while still capturing long-run correlations on very long histories. Using a medium number of predictor tables (4–8) and maximal history length of more than 200 bits was shown to be realistic.

While using an adder tree was shown as an effective final prediction computation function by the GEHL predictor, partial tag matching may be even more storage effective. The TAGE predictor (Fig. 9) proposed in 2006 [19] uses the geometric history length principle,



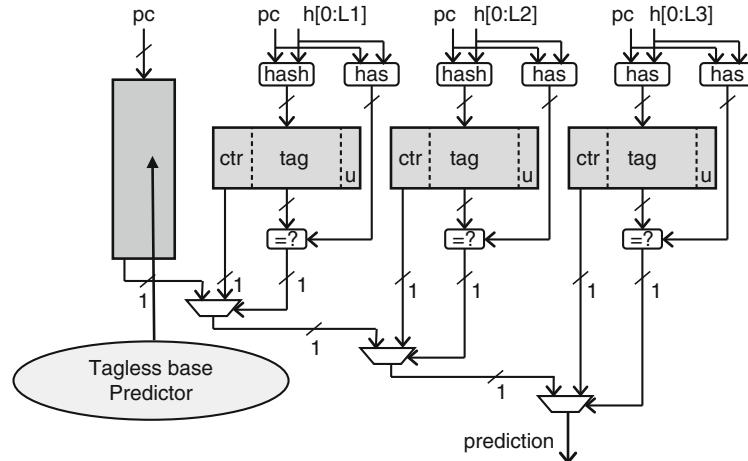
Branch Predictors. Fig. 8 The GEHL predictor: tables are indexed using different global history length that forms a geometric series

but relies on partial tag matching for final prediction computation. Each table in the predictor is read and the prediction is provided by the hitting predictor component with the longest history. If there is no hitting component then the prediction is provided by the default predictor.

Realistic size TAGE currently represents the state of the art in conditional branch predictors [17] and its misprediction rate is within 20% of the currently known limits for conditional branch predictability [16].

Predicting Indirect Branch Targets

Return and indirect jump targets are also only known at execution time and must be predicted.



Branch Predictors. Fig. 9 The TAGE predictor: partial tag match determines the prediction

Predicting Return Targets

Kaeli and Emma [9] remarked that, in most cases and particularly for compiler generated code the return address of procedure calls obey a very simple *call-return* rule: The return address target is the address just following the call instruction. They also remarked that the return addresses of the procedures could be predicted through a simple return address stack (RAS). On a call, the address of the next instruction in sequence is pushed on the top of the stack. On a return, the target of the return is popped from the top of the stack.

When the code is generated following the *call-return* rule and if there is no branch misprediction between the call and the return, an infinite size RAS predicts the return target with a 100% accuracy. However, several difficulties arise for practical implementation. The *call-return* rule is not always respected. The RAS is size limited, but in practice a 32-entry RAS is sufficient for most applications. The main difficulty is associated with the speculative execution: When on the wrong path returns are fetched followed by one or more calls, valid RAS entries are corrupted with wrong information, thus generating mispredictions on the returns on the right path. Several studies [8, 20, 23] have addressed this issue, and the effective accuracy of return target prediction is close to perfect.

Indirect Jump Target Predictions

To predict the targets of indirect jumps, one can use the same kind of information that can be used for

predicting the direction of the conditional branches, i.e., the global branch history or program path history.

First it was proposed to use the last encountered target, i.e., just the entry in the BTB. Chang et al. [1] proposed to use a predictor indexed by the global history and the program counter. Driesen and Holzle [3] proposed to use an hybrid predictor based on tag matching. Finally Seznec and Michand [19] proposed ITTAGE, a multicomponent indirect jump predictor, based on partial tag matching and the use of geometric history length as TAGE.

Conclusion

In the modern processors, the whole set of branch predictors (conditional, returns, indirect jumps, BTB) is an important performance enabler. They are particularly important on deep pipelined wide-issue superscalar processors, but they are also becoming important in low power embedded processors as they are also implementing instruction-level parallelism.

A large body of research has addressed branch prediction during the 1990s and the early 2000s and complex predictors inspired by this research have been implemented in processors during the last decade. Current state-of-the-art branch predictors combine multiple predictions and rely on the use of global history. These predictors cannot deliver a prediction in a single cycle. Since, prediction is needed on the very next cycle, various techniques such as overriding predictors [6] or ahead pipelined predictors [18] were proposed and implemented.

Branch predictor accuracy seems to have reached a plateau since the introduction of TAGE. Radically new predictor ideas, probably new sources of predictability, will probably be needed to further increase the predictor accuracy.

Bibliography

1. Chang P-Y, Hao E, Patt YN (1997) Target prediction for indirect jumps. In: ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture, Denver, 2–4 June 1997. ACM Press, New York, pp 274–283
2. Diefendorff K (1999) Compaq chooses SMT for alpha. Microprocessor Report, Dec 1999
3. Driesen K, Holzle U (1998) The cascaded predictor: economical and adaptive branch target prediction. In: Proceeding of the 31st Annual ACM/IEEE International Symposium on Microarchitecture, Dallas, Dec 1998, pp 249–258
4. Eden AN, Mudge T (1998) The yags branch predictor. In: Proceedings of the 31st Annual International Symposium on Microarchitecture, Dallas, Dec 1998
5. Evers M (2000) Improving branch behavior by understanding branch behavior. Ph.D. thesis, University of Michigan
6. Jiménez D (2003) Reconsidering complex branch predictors. In: Proceedings of the 9th International Symposium on High Performance Computer Architecture, Anaheim, 8–12 Feb 2003. IEEE, Los Alamitos
7. Jiménez DA, Lin C (2001) Dynamic branch prediction with perceptrons. In: HPCA: Proceedings of the 7th International Symposium on High Performance Computer Architecture, Monterrey, 19–24 Jan 2001. IEEE, Los Alamitos, pp 197–206
8. Jourdan S, Hsing T-H, Stark J, Patt YN (1996) The effects of mispredicted-path execution on branch prediction structures. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, Boston, 20–23 Oct 1996. IEEE, Los Alamitos
9. Kaeli DR, Emma PG (1991) Branch history table prediction of moving target branches due to subroutine returns. SIGARCH Comput Archit News 19(3):34–42
10. Lee C-C, Chen I-C, Mudge T (1997) The bi-mode branch predictor. In: Proceedings of the 30th Annual International Symposium on Microarchitecture, Dec 1997
11. Lee J, Smith A (1984) Branch prediction strategies and branch target buffer design. IEEE Comput 17(1):6–22
12. McFarling S (1993) Combining branch predictors, TN 36, DECWRL, Palo Alto, June 1993
13. Michaud P, Seznec A, Uhlig R (1997) Trading conflict and capacity aliasing in conditional branch predictors. In: Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA), Denver, 2–4 June 1997. ACM, New York
14. Pan S, So K, Rahmeh J (1992) Improving the accuracy of dynamic branch prediction using branch correlation. In: Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, 12–15 Oct 1992. ACM, New York
15. Seznec A (2005) Analysis of the o-gehl branch predictor. In: Proceedings of the 32nd Annual International Symposium on Computer Architecture, Madison, 4–8 June 2005. IEEE, Los Alamitos
16. Seznec A (2006) The idealistic gtl predictor. J Instruction Level Parallelism. <http://www.jilp.org/vol9>
17. Seznec A (2006) The l-tage branch predictor. J Instruction Level Parallelism. <http://www.jilp.org/vol9>
18. Seznec A, Fraboulet A (2003) Effective a head pipelining of the instruction address generator. In: Proceedings of the 30th Annual International Symposium on Computer Architecture, San Diego, 9–11 June 2003. IEEE, Los Alamitos
19. Seznec A, Michaud P (2006) A case for (partially)-tagged geometric history length predictors. J Instruction Level Parallelism. <http://www.jilp.org/vol8>
20. Skadron K, Martonosi M, Clark D (2000) Speculative updates of local and global branch history: a quantitative analysis. J Instruction-Level Parallelism 2
21. Smith J (1981) A study of branch prediction strategies. In: Proceedings of the 8th Annual International Symposium on Computer Architecture, May 1981. ACM, New York, pp 135–148
22. Sprangle E, Chappell R, Alsup M, Patt Y (1997) The agree predictor: a mechanism for reducing negative branch history interference. In: 24th Annual International Symposium on Computer Architecture, Denver, 2–4 June 1997. ACM, New York
23. Vandierendonck H, Seznec A (2008) Speculative return address stack management revisited. ACM Trans Archit Code Optim 5(3):1–20
24. Yeh T-Y, Patt Y (1991) Two-level adaptive branch prediction. In: Proceedings of the 24th International Symposium on Microarchitecture, Albuquerque, 18–20 Nov 1991. ACM, New York
25. Yeh T-Y, Patt YN (1991) Two-level adaptive training branch prediction. In: Proceedings of the 24th Annual International Symposium on Microarchitecture, Albuquerque, 18–20 Nov 1991. ACM, New York

Brent's Law

► Brent's Theorem

Brent's Theorem

JOHN L. GUSTAFSON
Intel Corporation, Santa Clara, CA, USA

Synonyms

Brent's law

Definition

Assume a parallel computer where each processor can perform an arithmetic operation in unit time.

Further, assume that the computer has exactly enough processors to exploit the maximum concurrency in an algorithm with N operations, such that T time steps suffice. Brent's Theorem says that a similar computer with fewer processors, P , can perform the algorithm in time

$$T_P \leq T + \frac{N - T}{P},$$

where P is less than or equal to the number of processors needed to exploit the maximum concurrency in the algorithm.

Discussion

Brent's Theorem assumes a PRAM (Parallel Random Access Machine) model [3, 7]. The PRAM model is an idealized construct that assumes any number of processors can access any items in memory instantly, but then take unit time to perform an operation on those items. Thus, PRAM models can answer questions about how much arithmetic parallelism one can exploit in an algorithm if the communication cost were zero. Since many algorithms have very large amounts of theoretical PRAM-type arithmetic concurrency, Brent's Theorem bounds the theoretical time the algorithm would take on a system with fewer processors.

Example

Suppose the task is to solve the following system of two equations in two unknowns, u and v :

$$au + bu = x$$

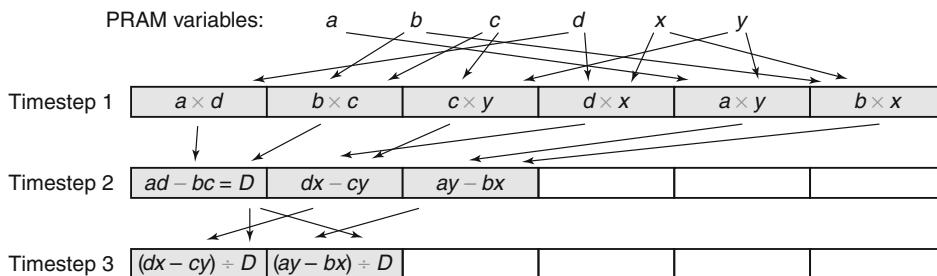
$$cu + du = y$$

One solution method is *Cramer's Rule*, which is far less efficient than Gaussian elimination in general, but exposes so much parallelism that it can take less time on a PRAM computer. Figure 1 shows how a PRAM with six processors can calculate the solution values, u and v in only three time steps:

Thus, $T = 3$ time steps, and $P \leq 6$, the amount of concurrency in the first time step. The total number of arithmetic operations, N , is 11 (six multiplications, followed by three subtractions, followed by two divisions). Brent's Theorem tells us an upper bound on the time a PRAM with *four* processors would take to perform the algorithm:

$$T_P \leq T + \frac{N - T}{P}, \text{ so}$$

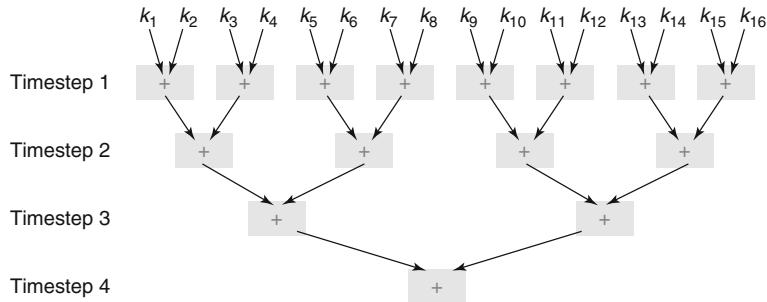
$$T_4 \leq 3 + \frac{11 - 3}{4} = 5.$$



Brent's Theorem. Fig. 1 Concurrency of a solver for two equations in two unknowns

Timestep 1	$a \times d$	$b \times c$	$c \times y$	$d \times x$	
Timestep 2	$a \times y$	$b \times x$			
Timestep 3	$ad - bc = D$	$dx - cy$	$ay - bx$		
Timestep 4	$(dx - cy) \div D$	$(ay - bx) \div D$			

Brent's Theorem. Fig. 2 Linear solver on only four processors



Brent's Theorem. Fig. 3 Binary sum collapse with a maximum parallelism of eight processors

Figure 1 shows that four time steps suffice in this case, fewer than the five in the bound predicted by Brent's Theorem. The figure omits dataflow arrows, for clarity.

Brent's Theorem says the four-processor system should require no more than five time steps, and Fig. 1 shows this is clearly the case.

Asymptotic Example: Parallel Summation

A common use of Brent's Theorem is to place bounds on the applicability of parallelism to problems parameterized by size. For example, to find the sum of a list of n numbers, a PRAM can add the numbers in pairs, then the resulting sums in pairs, and so on until the result is a single summation value. Figure 3 shows such a binary sum collapse for a set of 16 numbers.

In the following discussion, the notation

$\lg(x)$ denotes the logarithm base 2 of x , and the notation $[X]$ denotes the “ceiling” function, the smallest integer larger than x .

If n is exactly a power of 2, such that $n = 2^T$, then the PRAM can complete the sum in T time steps. That is, $T = \lg(n)$. If n is not an integer power of 2, then $T = [\lg(n)]$ because the summation takes one additional step. The total number of arithmetic operations for the summation of n numbers is $n - 1$ additions.

The maximum parallelism occurs in the first time step, when $n/2$ processors can operate on the list concurrently. If the number of processors in a PRAM is P , and P is less than $n/2$, then Brent's Theorem shows [2] that the execution time can be made less than or equal to

$$T_P = \lceil \lg(n) \rceil + \frac{(n - 1) - |\lg(n)|}{P}.$$

For values of n much larger than P , the above formula shows $T_P \approx \frac{n}{P}$.

Proof of Brent's Theorem

Let n_j be the number of operations in step j that the PRAM can execute concurrently, where j is $1, 2, \dots, T$. Assume the PRAM has P processors, where $P \leq \max_j(n_j)$.

A property of the ceiling function is that for any two positive integers k and m ,

$$\left\lceil \frac{k}{m} \right\rceil \leq \frac{k + m - 1}{m}.$$

Hence, the time for the PRAM to perform each time step j has the following bound:

$$\left\lceil \frac{n_j}{P} \right\rceil \leq \frac{n_j + P - 1}{P}.$$

The time for all time steps has the bound

$$\begin{aligned} T_P &\leq \sum_{j=1}^T \left\lceil \frac{n_j}{P} \right\rceil \leq \sum_{j=1}^T \frac{n_j + P - 1}{P} \\ &= \sum_{j=1}^T \frac{n_j}{P} + \sum_{j=1}^T \frac{P}{P} - \sum_{j=1}^T \frac{1}{P}. \end{aligned}$$

Since the sum of the n_j is the total number of operations N , and the sum of 1 for T steps is simply T , this simplifies to

$$T_P \leq \frac{\sum n_j}{P} + \sum_{j=1}^T 1 - \frac{\sum 1}{P} = T + \frac{N - T}{P}.$$

Application to Superlinear Speedup

Ideal speedup is speedup that is linear in the number of processors. Under the strict assumptions of Brent's Theorem, superlinear speedup (or superunitary efficiency,

sometimes erroneously termed superunitary speedup) is impossible.

Superlinear speedup means that the time to perform an algorithm on a single processor is more than P times as long as the time to perform it on P processors. In other words, P processors are more than P times as fast as a single processor. According to Brent's Theorem with $P = 1$,

$$T_1 \leq T + \frac{N - T}{1} = N.$$

Speedup greater than the number of processors means $T_1 > P \cdot T_P$. Brent's Theorem says this is mathematically impossible. However, Brent's Theorem is based on the assumptions of the PRAM model that have little resemblance to real computer design, such as the assumptions that every arithmetic operation requires the same amount of time, and that memory access has zero latency and infinite bandwidth. The debate over the possibility of superlinear speedup first appeared in 1986 [4, 6], and the debate highlighted the oversimplification of the PRAM model for parallel computing performance analysis.

Perspective

The PRAM model used by Brent's Theorem assumes that communication has zero cost, and arithmetic work constitutes all of the work of an algorithm. The opposite is closer to the present state of computer technology (see *Memory Wall*), which greatly diminishes the usefulness of Brent's Theorem in practical problem solving. It is so esoteric that it does not even provide useful upper or lower bounds on how parallel processing might improve execution time, and modern performance analysis seldom uses it. Brent's Theorem is a mathematical model more related to graph theory and partial orderings than to actual computer behavior. When Brent constructed his model in 1974, most computers took longer to perform arithmetic on operands than they took to fetch and store the operands, so the approximation was appropriate. More recent abstract models of parallelism take into account communication costs, both bandwidth and latency, and thus can provide better guidance for the parallel performance bounds of current architectures.

Related Entries

- Bandwidth-Latency Models (BSP, LogP)
- Memory Wall
- PRAM (Parallel Random Access Machines)

B

Bibliographic Notes and Further Reading

As explained above, Brent's Theorem reflects the computer design issues of the 1970s, and readers should view Brent's original 1974 paper in this context. It was in 1986 that the esoteric nature of the PRAM model became clear, in the opposing papers [4] and [6]. Faber, Lubeck, and White [4] assumed the PRAM model to state that one cannot obtain superlinear speedup since computing resources increase linearly with the number of processors. Parkinson [6], having had experience with the ICL Distributed Array Processor (DAP), based his model and experience on premises very different from the hypothetical PRAM model used in Brent's Theorem. Parkinson noted that simple constructs like the sum of two vectors could take place superlinearly faster on P processors because a shared instruction to add, sent to the P processors, does not require the management of a loop counter and address increment that a serial processor requires. The 1989 work *Introduction to Algorithms* by Cormen, Leiserson, and Rivest [3] is perhaps the first textbook to formally recognize that the PRAM model is too simplistic, and thus Brent's Theorem has diminished predictive value.

Bibliography

1. Brent RP (1974) The parallel evaluation of general arithmetic expressions. J ACM 12(2):201–206
2. Cole R (1989) Faster optimal parallel prefix sums and list ranking. Inf Control 81(3):334–352
3. Cormen TH, Leiserson CE, Rivest RL (1989) Introduction to algorithms, MIT Press Cambridge
4. Faber V, Lubeck OM, White AB (1986) Superlinear speedup of an efficient sequential algorithm is not possible. Parallel Comput 3:259–260
5. Helmbold DP, McDowell CE (1989) Modeling speedup (n) greater than n . In: Proceedings of the international conference on parallel processing, 3:219–225
6. Parkinson D (1986) Parallel efficiency can be greater than unity. Parallel Comput 3:261–262
7. Smith JR (1993) The design and analysis of parallel algorithms. Oxford University Press, New York

Broadcast

JESPER LARSSON TRÄFF¹, ROBERT A. VAN DE GEIJN²

¹University of Vienna, Vienna, Austria

²The University of Texas at Austin, Austin, TX, USA

Synonyms

One-to-all broadcast; Copy

Definition

Among a group of processing elements (nodes), a designated *root* node has a data item to be communicated (copied) to all other nodes. The broadcast operation performs this collective communication operation.

Discussion

The reader may consider first visiting the entry on [►Collective Communication](#).

Let p be the number of nodes in the group that participate in the broadcast operation and number these nodes consecutively from 0 to $p - 1$. One node, the root with index r , has a vector of data, x of size n , to be communicated to the remainder $p - 1$ nodes:

Before			After		
Node r	Node 1	Node 2	Node r	Node 1	Node 2
x			x	x	x

All nodes are assumed to explicitly take part in the broadcast operation. It is generally assumed that before its execution all nodes know the index of the designated *root* node as well as the the amount n of data to be broadcast. The data item x may be either a single, atomic unit or divisible into smaller, disjoint pieces. The latter can be exploited algorithmically when n is large.

Broadcast is a key operation on parallel systems with distributed memory. On shared memory systems, broadcast can be beneficial for improving locality and/or avoiding memory conflicts.

Lower Bounds

To obtain some simple lower bounds, it is assumed that

- All nodes can communicate through a communication network.
- Individual nodes perform communication operations that send and/or receive individual messages.
- Communication is through a single *port*, such that a node can be involved in at most one communication operation at a time. Such an operation can be either a send to or a receive from another node (unidirectional communication), a combined send to and receive from another node (bidirectional, telephone like communication), or a send to and receive from two possibly different nodes (simultaneous send-receive, fully bidirectional communication).
- It is assumed that the communication medium is homogeneous and fully connected such that all nodes can communicate with the same costs and any maximal set of pairs of disjoint nodes can communicate simultaneously.
- A reasonable first approximation for the time for transferring a message of size n between (any) two nodes is $\alpha + n\beta$ where α is the start-up cost (latency) and β is the cost per item transferred (inverse of the bandwidth).

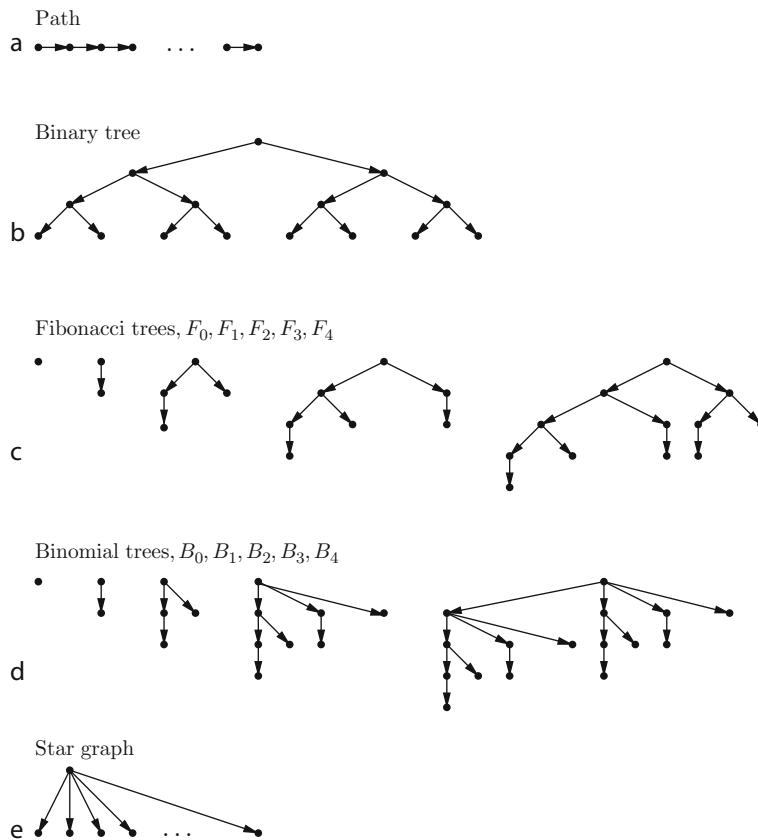
Under these assumptions, two lower bounds for the broadcast operation can be easily justified, the first for the α term and the second for the β term:

- $\lceil \log_2 p \rceil \alpha$. Define a round of communication as a period during which each node can send at most one message and receive at most one message. In each round, the number of nodes that know message x can at most double, since each node that has x can send x to a new node. Thus, a minimum of $\lceil \log_2 p \rceil$ rounds are needed to broadcast the message. Each round costs at least α .
- $n\beta$. If $p > 1$ then the message must leave the root node, requiring a time of at least $n\beta$.

When assumptions about the communication system change, these lower bounds change as well. Lower bounds for mesh and torus networks, hypercubes, and many other communication networks are known.

Tree-Based Broadcast Algorithms

A well-known algorithm for broadcasting is the so-called Minimum Spanning Tree (MST) algorithm



Broadcast. Fig. 1 Commonly used broadcast trees: (a) Path; (b) Binary Tree; (c) Fibonacci trees, the i th Fibonacci tree for $i > 1$ consists of a new root node connected to Fibonacci trees F_{i-1} and F_{i-2} ; (d) Binomial trees, the i th binomial tree for $i > 0$ consists of a new root with children B_{i-1}, B_{i-2}, \dots ; (e) Star

(which is a misnomer, since all broadcast trees are spanning trees and minimum for homogeneous communication media), which can be described as follows:

- Partition the set of nodes into two roughly equal-sized subsets.
- Send x from the root to a node in the subset that does not include the root. The receiving node will become a *local root* in that subset.
- Recursively broadcast x from the (local) root nodes in the two subsets.

Under the stated communication model, the total cost of the MST algorithm is $\lceil \log_2 p \rceil(\alpha + n\beta)$. It achieves the lower bound for the α term but not for the β term for which it is a logarithmic factor off from the lower bound.

If the data transfer between two nodes is represented by a communication edge between the two nodes, the MST algorithm constructs a *binomial* spanning tree over the set of nodes. An equivalent (also recursive) construction is as follows. The 0th binomial tree B_0 consists of a single node. For $i > 0$ the i th binomial tree B_i consists of the root with i children B_j for $j = i - 1, \dots, 0$. The number of nodes in B_i is 2^i . It can be seen that the number of nodes at level i is $\binom{i}{\log_2 p}$ from which the term originates.

The construction and structure of the binomial (MST) tree is shown in Fig. 1d.

Pipelining

For large item sizes, pipelining is a general technique to improve the broadcast cost. Assume first that the nodes

are communicating along a directed path with node 0 being the root and node i sending to node $i + 1$ for $0 \leq i < p - 1$ (node $p - 1$ only receives data). This is shown in Fig. 1a. The message to be broadcast is split into k blocks of size n/k each. In the first round of the algorithm, the first block is sent from node 0 to node 1. In the second round, this block is forwarded to node 2 while the second block is sent to node 1. In this fashion, a pipeline is established that communicates the blocks to all nodes.

Under the prior model, the cost of this algorithm is

$$\begin{aligned} (k+p-2)\left(\alpha + \frac{n}{k}\beta\right) &= (k+p-2)\alpha + \frac{k+p-2}{k}n\beta \\ &= (p-2)\alpha + n\beta + k\alpha + \frac{p-2}{k}n\beta. \end{aligned}$$

It takes $p - 1$ communication rounds for the first piece to reach node $p - 1$, which afterward in each successive round receives a new block. Thus, an additional $k - 1$ rounds are required for a total of $k + p - 2$ rounds. Balancing the $k\alpha$ term against the $\frac{p-2}{k}n\beta$ term gives a minimum time of

$$\left(\sqrt{(p-2)\alpha} + \sqrt{\beta n}\right)^2 = (p-2)\alpha + 2\sqrt{(p-2)n\alpha\beta} + n\beta.$$

and best possible number of blocks $k = \sqrt{\frac{(p-2)\beta n}{\alpha}}$. This meets the lower bound for the β term, but not for the α term. For very large n compared to p the linear pipeline can be a good (practical) algorithm. It is straightforward to implement and has very small extra overhead.

For the common case where p and/or the start-up latency α is large compared to n , the linear pipeline suffers from the nonoptimal $(p-2)\alpha$ term, and algorithms with a shorter longest path from root node to receiving nodes will perform better. Such algorithms can apply pipelining using different, fixed degree trees, as illustrated in Fig. 1a–c. For instance, with a balanced binary tree as shown in Fig. 1b the broadcast time becomes

$$2(\log_2 p - 1) + 4\sqrt{(\log p - 1)\alpha} \sqrt{\beta n} + 2\beta n.$$

The latency of this algorithm is significantly better than the linear pipeline, but the β term is a factor of 2 off from optimal. By using instead skewed trees, the time at which the last node receives the first block of x can be improved which affects both α and β terms. The Fibonacci tree shown in Fig. 1c for instance achieves a

broadcast cost of

$$(\log_\Theta p - 2)\alpha + 2\sqrt{2(\log_\Theta p - 2)\alpha} \sqrt{\beta n} + 2\beta n,$$

where $\Theta = \frac{1+\sqrt{5}}{2}$. Finally it should be noted that pipelining cannot be employed with any advantage for trees with nonconstant degrees like the binomial tree and the degenerated star tree shown in Fig. 1d–e.

A third lower bound for broadcasting of data k items can be justified similarly to the two previously introduced bounds, and is applicable to algorithms that apply pipelining.

- $k - 1 + \lceil \log_2 p \rceil$ rounds. First $k - 1$ items have to leave the root, and the number of rounds for the last item to arrive at some last node is an additional $\lceil \log_2 p \rceil$.

With this bound, the best possible broadcast cost in the linear cost model is therefore $(k - 1 + \log_2 p)(\alpha + \beta n/k)$ when x is (can be) divided into k roughly equal-sized blocks of size at most $\lceil n/k \rceil$. Balancing the $k\alpha$ term against the $(\log_2 p - 1)\beta n/k$ term achieves a minimum time of

$$\begin{aligned} \left(\sqrt{(\log_2 p - 1)\alpha} + \sqrt{\beta n}\right)^2 &= (\log_2 p - 1)\alpha \\ &\quad + 2\sqrt{(\log_2 p - 1)\alpha} \sqrt{\beta n} + \beta n \end{aligned}$$

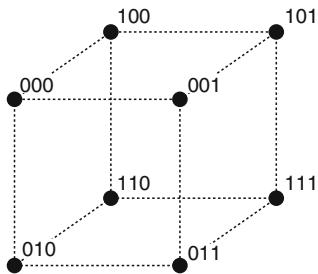
with the best number blocks being $k = \sqrt{\frac{(\log_2 p - 1)\beta n}{\alpha}}$.

Simultaneous Trees

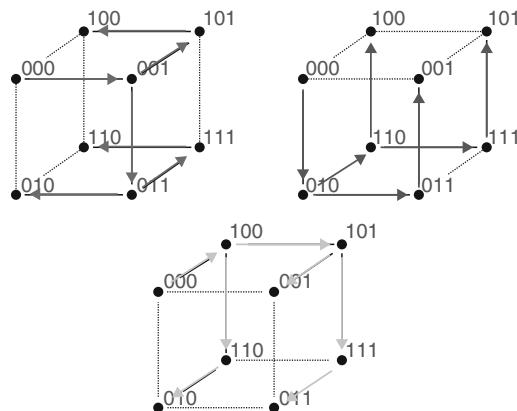
None of the tree-based algorithm were optimal in the sense of meeting the lower bound for both the α and the β terms. A practical consequence of this is that broadcast implementations become cumbersome in that algorithms for different combinations of p and n have to be maintained. The breakthrough results of Johnsson and Ho [17] show how employing multiple, simultaneously active trees can be used to overcome these limitations and yield algorithms that are optimal in the number of communication rounds. The results were at first formulated for hypercubes or fully connected communication networks where p is a power of two, and (much) later extended to communication networks with arbitrary number of nodes. The basic idea can also be employed for meshes and hypercubes.

The idea is to embed simultaneous, edge disjoint spanning trees in the communication network, and use each of these trees to broadcast different blocks of the

input. If this embedding can be organized such that each node has the same number of incoming edges (from different trees) and outgoing edges (to possibly different trees), a form of pipelining can be employed, even if the individual trees are not necessarily fixed degree trees. To illustrate the idea, consider the three-dimensional hypercube, and assume that an infinite number of blocks have to be broadcast.



Let the nodes be numbered in binary with the broadcast root being node 000. The three edge-disjoint trees used for broadcasting blocks $0, 3, 6, \dots; 1, 4, 7, \dots$ and $2, 5, 8, \dots$; respectively, are as shown below:



The trees are in fact constructed as edge disjoint spanning trees excluding node 000 rooted at the root hypercube neighbors 001, 010, and 100. The root is connected to each of these trees. In round 0, block 0 is sent to the first tree, which in rounds 1, 2, and 3 is responsible for broadcasting to its spanning subtree. In round 1, block 1 is sent to the second tree which uses rounds 2, 3, and 4 to broadcast, and in round 2 block 2 is sent to the third tree which uses rounds 3, 4, and 5 to broadcast this block. In round 3, the root sends the third block, the broadcasting of which takes place simultaneously with the broadcasting of the previous

blocks, and so on. The broadcasting is done along a regular pattern, which is symmetric for all nodes. For a d -dimensional hypercube, node i in round t , $t \geq 0$, sends and receives a block to and from the node found by toggling bit $t \bmod d$. As can be seen, each node is a leaf in tree j if bit j of i is 0, otherwise an internal node in tree j . Leaves in tree $j = t \bmod d$ receive block $t - d$ in round d . When node i is an internal node in tree j , the block received in round j is sent to the children of i which are the nodes found by toggling the next, more significant bits after j until the next position in i that is a 1. Thus, to determine which blocks are received and sent by node i in each round, the number of zeroes from each bit position in i until the next, more significant 1 will suffice.

The algorithm for broadcasting k blocks in the optimal $k - 1 + \log_2 p$ number of rounds is given formally in Fig. 2. To turn the description above into an algorithm for an arbitrary, finite number of blocks, the following modifications are necessary. First, any block with a negative number is neither sent nor received. Second, for blocks with a number larger than $k - 1$, block $k - 1$ is taken instead. Third, if $k - 1$ is not a multiple of $\log_2 p$ the broadcast is started at a later round f such that indeed $k + f - 1$ is a multiple of $\log_2 p$. The table $\text{BitDistance}_i[j]$ stores for each node i the distance from bit position j in i to the next 1 to the left of position j (with wrap around after d bits). The root 000 has no ones. This node only sends blocks, and therefore $\text{BitDistance}_{00}[k] = k$ for $0 \leq k < d$. For each i , $0 \leq i < p$ the table can be filled in $O(\log_2 p)$ steps.

Composing from Other Collective Communications

Another approach to implementing the broadcast operation follows from the observation that data can be broadcast to p nodes by scattering p disjoint blocks of x of size n/p across the nodes, and then reassembling the blocks at each node by an allgather operation. On a network that can host a hypercube, the scatter and allgather operations can be implemented at a cost of $(\log_2 p)\alpha + \frac{p-1}{p}n\beta$ each, under the previously used communication cost model (see entries on scatter and allgather). This yields a cost of

$$2\log_2 p\alpha + 2\frac{p-1}{p}n\beta.$$

```

 $f \leftarrow ((k \bmod d) + d - 1) \bmod d$  /* Start round for first phase */
 $t \leftarrow 0$ 
while  $t < k + d - 1$  do
    /* New phase consisting of (up to)  $d$  rounds */
    for  $j \leftarrow f$  to  $d - 1$ 
         $s \leftarrow t - d + (1 - i_j) * \text{BitDistance}_i[j]$  /* block to send */
         $r \leftarrow t - d + i_j * \text{BitDistance}_i[j]$  /* block to receive */
        if  $s \geq k$  then  $s \leftarrow k - 1$ 
        if  $r \geq k$  then  $r \leftarrow k - 1$ 
        par/* simultaneous send-receive with neighbor */
            if  $s \geq 0$  then Send block  $s$  to node ( $i \bmod 2^d$ )
            if  $r \geq 0$  then Receive block  $r$  from node ( $i \bmod 2^d$ )
        end par
         $t \leftarrow t + 1$  /* next round */
    end for
     $f \leftarrow 0$  /* next phases start from 0 */
endwhile

```

Broadcast. Fig. 2 The algorithm for node i , $0 \leq i < p$ for broadcasting k blocks in a d -dimensional hypercube or fully connected network with $p = 2^d$ nodes. The algorithm requires the optimal number of $k - 1 + d$ rounds. The j th bit of i is denoted i_j and is used to determine the hypercube neighbor of node i for round j

The cost of this algorithm is within a factor two of the lower bounds for both the α and β term and is considerably simpler to implement than approaches that use pipelining.

General Graphs

Good broadcast algorithms are known for many different communication networks under various cost models. However, the problem of finding a best broadcast schedule for an arbitrary communication network is a hard problem. More precisely, the following problem of determining whether a given number of rounds suffices to broadcast in an arbitrary, given graph is NP-complete [13, Problem ND49]: Given an undirected graph $G = (V, E)$, a *root* vertex $r \in V$, and an integer k (number of rounds), is there a sequence of vertex and edge subsets $\{r\} = V_0, E_1, V_1, V_2, E_2, \dots, E_k, V_k = V$ with $V_i \subseteq V$, $E_i \subseteq E$, such that each $e \in E_i$ has one endpoint in V_{i-1} and one in V_i , no two edges of E_i share an endpoint, and $V_i = V_{i-1} \cup \{w | (v, w) \in E_i\}$?

Related Entries

- [Allgather](#)
- [Collective Communication](#)
- [Collective Communication, Network Support for](#)
- [Message Passing Interface \(MPI\)](#)
- [PVM \(Parallel Virtual Machine\)](#)

Bibliographic Notes and Further Reading

Broadcast is one of the most thoroughly studied collective communication operations. Classical surveys with extensive treatment of broadcast (and allgather/gossiping) problems under various communication and network assumptions can be found in [11, 14]. For a survey of broadcasting in distributed systems, which raises many issues not discussed here, see [9]. Well-known algorithms that have been implemented in, for instance, Message Passing Interface (MPI) libraries include the MST algorithm, binary trees, and scatter-allgather approaches. Fibonacci trees for broadcast were explored in [6]. Another skewed, pipelined tree structure termed fractional trees that yield close to optimal results for certain ranges of p and n was proposed in [22].

The basic broadcast techniques discussed in this entry date back to the early days of parallel computing [10, 21]. The scatter/allgather algorithm was already discussed in [10] and was subsequently popularized for mesh architectures in [3]. It was further popularized for use in MPI implementations in [25]. A different implementation of this paradigm was given in [26]. Modular construction of hybrid algorithms from MST broadcast and scatter/allgather is discussed in [7].

The classical paper by Johnsson and Ho [17] introduced the simultaneous tree algorithm that was

discussed, which they call the n -ESBT algorithm. This algorithm can be used for networks that can host a hypercube and has been used in practice for hypercubes and fully connected systems. It achieves the lower bound on the number of communication rounds needed to broadcast k blocks of data. The exposition given here is based on [16, 27]. It was for a number of years an open problem how to achieve similar optimality for arbitrary p (and k). The first round-optimal algorithms were given in [2, 18], but these seem not to have been implemented. A different, explicit construction was found and described in [27], and implemented in an MPI library. A very elegant (and practical) extension of the hypercube n -ESBT algorithm to arbitrary p was presented in [16]. That algorithm uses the hypercube algorithm for the largest i such that $2^i \leq p$. Each node that is not in the hypercube is paired up with a hypercube node and the two nodes in each such pair in an alternating fashion jointly carry out the work of a hypercube node. Yet another, optimal to a lower-order term algorithm based on using two edge-disjoint binary trees was given in [23]. Disjoint spanning tree algorithms for multidimensional mesh and torus topologies were given in [28].

The linear model used for modeling communication (transfer) costs is folklore. An arguably more accurate performance model of communication networks is the so-called LogGP model (and its variants), which account more accurately for the time in which processors are involved in data transfers. With this model, yet other broadcast tree structures yield best performance [8, 24]. The so-called postal model in which a message sent at some communication round is received a number of rounds λ later at the destination was used in [1, 5] and gives rise to yet more tree algorithms.

Broadcasting in heterogeneous systems has recently received renewed attention [4, 19, 20].

Finding minimum round schedules for broadcast remain NP-hard for many special networks [15]. General approximation algorithms have been proposed for instance in [12].

Bibliography

1. Bar-Noy A, Kipnis S (1994) Designing broadcasting algorithms in the postal model for message-passing systems. *Math Syst Theory* 27(5):431–452
2. Bar-Noy A, Kipnis S, Schieber B (2000) Optimal multiple message broadcasting in telephone-like communication systems. *Discret Appl Math* 100(1–2):1–15
3. Barnett M, Payne DG, van de Geijn RA, Watts J (1996) Broadcasting on meshes with wormhole routing. *J Parallel Distrib Comput* 35(2):111–122
4. Beaumont O, Legrand A, Marchal L, Robert Y (2005) Pipelining broadcast on heterogeneous platforms. *IEEE Trans Parallel Distrib Syst* 16(4):300–313
5. Bruck J, De Coster L, Dewulf N, Ho C-T, Lauwereins R (1996) On the design and implementation of broadcast and global combine operations using the postal model. *IEEE Trans Parallel Distrib Syst* 7(3):256–265
6. Bruck J, Cypher R, Ho C-T (1992) Multiple message broadcasting with generalized fibonacci trees. In: *Symposium on Parallel and Distributed Processing (SPDP)*. IEEE Computer Society Press. Arlington, Texas, USA, pp 424–431
7. Chan E, Heimlich M, Purkayastha A, van de Geijn RA (1783) Collective communication: theory, practice, and experience. *Concurr Comput* 19(13):1749–1783
8. Culler DE, Karp RM, Patterson D, Sahay A, Santos EE, Schausser KE, Subramonian R, von Eicken T (1996) LogP: A practical model of parallel computation. *Commun ACM* 39(11):78–85
9. Défago X, Schiper A, Urbán P (2004) Total order broadcast and multicast algorithms: taxonomy and survey. *ACM Comput Surveys* 36(4):372–421
10. Fox G, Johnson M, Lyzenga M, Otto S, Salmon J, Walker D (1988) Solving problems on concurrent processors, vol I. Prentice-Hall, Englewood Cliffs
11. Fraigniaud P, Lazard E (1994) Methods and problems of communication in usual networks. *Discret Appl Math* 53(1–3):79–133
12. Fraigniaud P, Vial S (1997) Approximation algorithms for broadcasting and gossiping. *J Parallel Distrib Comput* 43:47–55
13. Garey MR, Johnson DS (1979) Computers and intractability: a guide to the theory of NP-completeness. Freeman, San Francisco (With an addendum, 1991)
14. Hedetniemi SM, Hedetniemi T, Liestman AL (1988) A survey of gossiping and broadcasting in communication networks. *Networks* 18:319–349
15. Jansen K, Müller H (1995) The minimum broadcast time problem for several processor networks. *Theor Comput Sci* 147(1&2):69–85
16. Jia B (2009) Process cooperation in multiple message broadcast. *Parallel Comput* 35(12):572–580
17. Johnsson SL, Ho C-T (1989) Optimum broadcasting and personalized communication in hypercubes. *IEEE Trans Comput* 38(9):1249–1268
18. Kwon O-H, Chwa K-Y (1995) Multiple message broadcasting in communication networks. *Networks* 26:253–261
19. Libeskind-Hadas R, Hartline JRK, Boothe P, Rae G, Swisher J (2001) On multicast algorithms for heterogenous networks of workstations. *J Parallel Distrib Comput* 61:1665–1679
20. Liu P (2002) Broadcast scheduling optimization for heterogeneous cluster systems. *J Algorithms* 42(1):135–152
21. Saad Y, Schultz MH (1989) Data communication in parallel architectures. *Parallel Comput* 11(2):131–150

22. Sanders P, Sibeyn JF (2003) A bandwidth latency tradeoff for broadcast and reduction. *Inf Process Lett* 86(1):33–38
23. Sanders P, Speck J, Träff JL (2001) Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Comput* 35:581–594
24. Santos EE (1999) Optimal and near-optimal algorithms for k-item broadcast. *J Parallel Distrib Comput* 57(2):121–139
25. Thakur R, Gropp WD, Rabenseifner R (2004) Improving the performance of collective operations in MPICH. *Int J High Perform Comput Appl* 19:49–66
26. Träff JL (2004) A simple work-optimal broadcast algorithm for message-passing parallel systems. In: Recent advances in parallel virtual machine and message passing interface. 11th European PVM/MPI users' group meeting. Lecture Notes in Computer Science, vol 3241. Springer, pp 173–180
27. Träff JL, Ripke A (2008) Optimal broadcast for fully connected processor-node networks. *J Parallel Distrib Comput* 68(7): 887–901
28. Watts J, van de Geijn RA (1995) A pipelined broadcast for multi-dimensional meshes. *Parallel Process Lett* 5:281–292

BSP

- [Bandwidth-Latency Models \(BSP, LogP\)](#)
- [BSP \(Bulk Synchronous Parallelism\)](#)

BSP (Bulk Synchronous Parallelism)

ALEXANDER TISKIN

University of Warwick, Coventry, UK

Definition

Bulk-synchronous parallelism is a type of coarse-grain parallelism, where inter-processor communication follows the discipline of strict barrier synchronization. Depending on the context, BSP can be regarded as a computation model for the design and analysis of parallel algorithms, or a programming model for the development of parallel software.

Discussion

Introduction

The model of *bulk-synchronous parallel (BSP) computation* was introduced by Valiant [77] as a “bridging model” for general-purpose parallel computing.

The BSP model can be regarded as an abstraction of both parallel hardware and software, and supports an approach to parallel computation that is both architecture-independent and scalable. The main principles of BSP are the treatment of a communication medium as an abstract fully connected network, and the decoupling of all interaction between processors into point-to-point asynchronous data communication and barrier synchronization. Such a decoupling allows an explicit and independent cost analysis of local computation, communication, and synchronization, all of which are viewed as limited resources.

BSP Computation

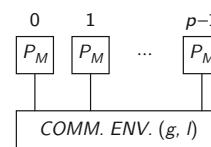
The BSP Model

A *BSP computer* (see Fig. 1) contains

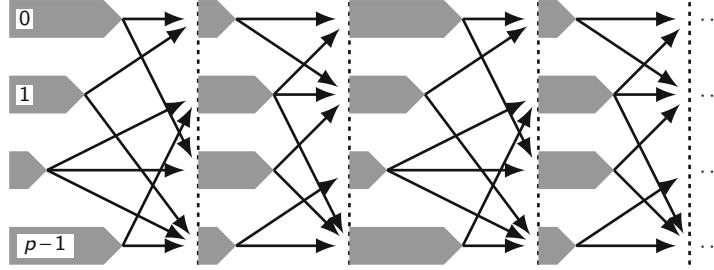
- p *Processors*; each processor has a local memory and is capable of performing an elementary operation or a local memory access every time unit.
- A *communication environment*, capable of accepting a word of data from every processor, and delivering a word of data to every processor, every g time units.
- A *barrier synchronization* mechanism, capable of synchronizing all the processors simultaneously every l time units.

The processors may follow different threads of computation, and have no means of synchronizing with one another apart from the global barriers.

A BSP computation is a sequence of *supersteps* (see Fig. 2). The processors are synchronized between supersteps; the computation within a superstep is completely asynchronous. Consider a superstep in which every processor performs a maximum of w local operations, sends a maximum of h_{out} words of data, and receives a maximum of h_{in} words of data. The value w is the *local computation cost*, and $h = h_{\text{out}} + h_{\text{in}}$ is the *communication cost* of the superstep. The total superstep cost is defined



BSP (Bulk Synchronous Parallelism). Fig. 1 The BSP computer



BSP (Bulk Synchronous Parallelism). Fig. 2 BSP computation

as $w + h \cdot g + l$, where the *communication gap* g and the *latency* l are parameters of the communication environment. For a computation comprising S supersteps with local computation costs w_s and communication costs h_s , $1 \leq s \leq S$, the total cost is $W + H \cdot g + S \cdot l$, where

- $W = \sum_{1 \leq s \leq S} w_s$ is the total *local computation cost*.
- $H = \sum_{s=1 \leq s \leq S} h_s$ is the total *communication cost*.
- S is the *synchronization cost*.

The values of W , H , and S typically depend on the number of processors p and on the problem size.

In order to utilize the computer resources efficiently, a typical BSP program regards the values p , g , and l as configuration parameters. Algorithm design should aim to minimize local computation, communication, and synchronization costs for any realistic values of these parameters. The main BSP design principles are

- Load balancing, which helps to minimize both the local computation cost W and the communication cost H
- Data locality, which helps to minimize the communication cost H
- Coarse granularity, which helps to minimize (or sometimes to trade off) the communication cost H and the synchronization cost S

The term “data locality” refers here to placing a piece of data in the local memory of a processor that “needs it most.” It has nothing to do with the locality of a processor in a specific network topology, which is actively discouraged from use in the BSP model. To distinguish these concepts, some authors use the terms “strict locality” [38] or “co-locality” [67].

The values of network parameters g , l for a specific parallel computer can be obtained by *benchmarking*. The benchmarking process is described in [8]; the

resulting lists of machine parameters can be found in [8, 63].

BSP vs Traditional Parallel Models

Traditionally, much of theoretical research in parallel algorithms has been done using the Parallel Random Access Machine (PRAM), proposed initially in [25]. This model contains

- A potentially unlimited number of *processors*, each capable of performing an elementary operation every time unit
- Global shared memory, providing uniform access for every processor to any location in one time unit

A PRAM computation proceeds in a sequence of synchronous parallel *steps*, each taking one time unit. Concurrent reading or writing of a memory location by several processors within the same step may be allowed or disallowed. The number of processors is potentially unbounded, and is often considered to be a function of the problem size. If the number of processors p is fixed, the PRAM model can be viewed as a special case of the BSP model, with $g = l = 1$ and communication realized by reading from/writing to the shared memory.

Since the number of processors in a PRAM can be unbounded, a common approach to PRAM algorithm design is to associate a different processor with every data item. Often, the processing of every item is identical; in this special case, the computation is called *data-parallel*. Programming models and languages designed for data-parallel computation can benefit significantly from the BSP approach to cost analysis; see [39] for a more detailed discussion.

It has long been recognized that in order to be practical, a model has to impose a certain structure on the

communication and/or synchronization patterns of a parallel computation. Such a structure can be provided by defining a restricted set of collective communication primitives, called *skeletons*, each with an associated cost model (see, e.g., [16]). In this context, a BSP superstep can be viewed as a simple generalized skeleton (see also [39]). However, current skeleton proposals concentrate on more specialized, and somewhat more arbitrary, skeleton sets (see, e.g., [17]).

The CTA/Phase Abstractions model [1], which underlies the ZPL language, is close in spirit to skeletons. A CTA computation consists of a sequence of *phases*, each with a simple and well-structured communication pattern. Again, the BSP model takes this approach to the extreme, where a superstep can be viewed as a phase allowing a single generic asynchronous communication pattern, with the associated cost model.

Memory Efficiency

The original definition of BSP does not account for memory as a limited resource. However, the model can be easily extended by an extra parameter m , representing the maximum capacity of each processor's local memory. Note that this approach also limits the amount of communication allowed within a superstep: $h \leq m$. One of the early examples of memory-sensitive BSP algorithm design is given by [59].

An alternative approach to reflecting memory cost is given by the model CGM, proposed in [20]. A CGM is essentially a memory-restricted BSP computer, where memory capacity and maximum superstep communication are determined by the size of the input/output: $h \leq m = O(N/p)$. A large number of algorithms have been developed for the CGM, see, e.g., [68].

Memory Management

The BSP model does not directly support shared memory, a feature that is often desirable for both algorithm design and programming. Furthermore, the BSP model does not properly address the issue of input/output, which can also be viewed as accessing an external shared memory. Virtual shared memory can be obtained by PRAM simulation, a technique introduced in [77]. An efficient simulation on a p -processor BSP computer is possible if the simulated virtual PRAM has at least

$p \log p$ processors, a requirement known as *slackness*. Memory access in the randomized simulation is made uniform by *address hashing*, which ensures a nearly random and independent distribution of virtual shared memory cells.

In the *automatic mode* of BSP programming proposed in [77] (see also [24]), shared memory simulation completely hides the network and processors' local memories. The algorithm designer and the programmer can enjoy the benefits of virtual shared memory; however, data locality is destroyed, and, as a result, performance may suffer. A useful compromise is achieved by the BSPRAM model, proposed in [70]. This model can be seen as a hybrid of BSP and PRAM, where each processor keeps its local memory, but in addition there is a uniformly accessible shared (possibly external) memory. Automatic memory management can still be achieved by the address-hashing technique of [77]; additionally, there are large classes of algorithms, identified by [70], for which simpler, slackness-free solutions are possible.

In contrast to the standard BSP model, the BSPRAM is meaningful even with a single processor ($p = 1$). In this case, it models a sequential computation that has access both to main and external memory (or the cache and the main memory). Further connections between parallel and external-memory computations are explored, e.g., in [66].

Paper [19] introduced a more elaborate model EM-BSP, where each processor, in addition to its local memory, can have access to several external disks, which may be private or shared. Paper [78] proposed a restriction of BSPRAM, called BSPGRID, where only the external memory can be used for persistent data storage between supersteps – a requirement reflecting some current trends in processor architecture. Paper [64] proposed a shared-memory model QSM, where normal processors have communication gap g , and each shared memory cell is essentially a “mini-processor” with communication gap d . Naturally, in a superstep every such “mini-processor” can “send” or “receive” at most p words (one for each normal processor), hence the model is similar to BSPRAM with communication gap g and latency $dp + l$.

Virtual shared memory is implemented in several existing or proposed programming environments offering BSP-like functionality (see, e.g., [49, 54]).

Heterogeneity

In the standard BSP model, all processors are assumed to be identical. In particular, all have the same local processing speed (which is an implicit parameter of the model), and the same communication gap g . In practice, many parallel architectures are heterogeneous, i.e., include processors with different speeds and communication performances. This fact has prompted heterogeneous extensions of the BSP model, such as HBSP [79], and HCGM [61]. Both these extended models introduce a processor's speed as an explicit parameter. Each processor has its own speed and communication gap; these two parameters can be either independent or linked (e.g., proportional). The barrier structure of a computation is kept in both models.

Other Variants of BSP

The BSP^* model [7] is a refinement of the BSP model with an alternative cost formula for small-sized h -relations. Recognizing the fact that communication of even a small amount of data incurs a constant-sized overhead, the model introduces a parameter b , defined as the minimum communication cost of any, even zero-sized, h -relation. Since the overhead reflected by the parameter b can also be counted as part of superstep latency, the BSP^* computer with communication gap g and latency l is asymptotically equivalent to a standard BSP computer with gap g and latency $l + b$.

The E-BSP model [46] is another refinement of the BSP model, where the cost of a superstep is parameterized separately by the maximum amount of data sent, the maximum amount of data received, the total volume of communicated data, and the network-specific maximum distance of data travel. The OBSP * model [10] is an elaborate extension of BSP, which accounts for varying computation costs of individual instructions, and allows the processors to run asynchronously while maintaining “logical supersteps”. While E-BSP and OBSP * may be more accurate on some architectures (in [46], a linear array and a 2D mesh are considered), they lack the generality and simplicity of pure BSP. A simplified version of E-BSP, asymptotically equivalent to pure BSP, is defined in [47].

The PRO approach [26] is introduced as another parallel computation model, but can perhaps be better understood as an alternative BSP algorithm design philosophy. It requires that algorithms are work-optimal,

disregards point-to-point communication efficiency by setting $g = 1$, and instead puts the emphasis on synchronization efficiency and memory optimality.

BSP Algorithms

Basic Algorithms

As a simple parallel computation model, BSP lends itself to the design of efficient, well-structured algorithms. The aim of BSP algorithm design is to minimize the resource consumption of the BSP computer: the local computation cost W , the communication cost H , the synchronization cost S , and also sometimes the memory cost M . Since the aim of parallel computation is to obtain speedup over sequential execution, it is natural to require that a BSP algorithm should be work-optimal relative to a particular “reasonable” sequential algorithm, i.e., the local computation cost W should be proportional to the running time of the sequential algorithm, divided by p . It is also natural to allow a reasonable amount of slackness: an algorithm only needs to be efficient for $n \gg p$, where n is the problem size. The asymptotic dependence of the minimum value of n on the number of processors p must be clearly specified by the algorithm.

Assuming that a designated processor holds a value a , the *broadcasting* problem asks that a copy of a is obtained by every processor. Two natural solutions are possible: either direct or binary tree broadcast.

In the direct broadcast method, a designated processor makes $p - 1$ copies of a and sends them directly to the destinations. The BSP costs are $W = O(p)$, $H = O(p)$, $S = O(1)$. In the binary tree method, initially, the designated processor is defined as being *awake*, and the other processors as *sleeping*. The processors are woken up in $\log p$ rounds. In every round, every awake processor makes a copy of a and sends it to a sleeping processor, waking it up. The BSP costs are $W = O(p)$, $H = O(\log p)$, $S = O(\log p)$. Thus, there is a trade-off between the direct and the binary tree broadcast methods, and no single method is optimal (see [30, 77]).

The *array broadcasting* problem asks, instead of a single value, to broadcast an array a of size $n \geq p$. In contrast with the ordinary broadcast problem, there exists an optimal method for array broadcast, known as two-phase broadcast (a folklore result, described, e.g., in [8]). In this method, the array is partitioned into p

blocks of size n/p . The blocks are scattered across the processors; then, a total-exchange of the blocks is performed. Assuming sufficient slackness, the BSP costs are $W = O(n/p)$, $H = O(n/p)$, $S = O(1)$.

Many computational problems can be described as computing a directed acyclic graph (dag), which characterizes the problem's data dependencies. From now on, it is assumed that a BSP algorithm's input and output are stored in the external memory. It is also assumed that all problem instances have sufficient slackness.

The *balanced binary tree dag* of size n consists of $2n - 1$ nodes, arranged in a rooted balanced binary tree with n leaves. The direction of all edges can be either top-down (from root to leaves), or bottom-up (from leaves to root). By partitioning into appropriate blocks, the balanced binary tree dag can be computed with BSP costs $W = O(n/p)$, $H = O(n/p)$, $S = O(1)$ (see [8, 77]).

The *butterfly dag* of size n consists of $n \log n$ nodes, and describes the data dependencies of the Fast Fourier Transform algorithm on n points. By partitioning into appropriate blocks, the butterfly dag can be computed with BSP costs $W = O(n \log n/p)$, $H = O(n/p)$, $S = O(1)$ (see [8, 77]).

The *ordered 2D grid dag* of size n consists of n^2 nodes arranged in an $n \times n$ grid, with edges directed top-to-bottom and left-to-right. The computation takes $2n$ inputs to the nodes on the left and top borders, and returns $2n$ outputs from the nodes on the right and bottom borders. By partitioning into appropriate blocks, the ordered 2D grid dag can be computed with BSP costs $W = O(n^2/p)$, $H = O(n)$, $S = O(p)$ (see [57]).

The *ordered 3D grid dag* of size n consists of n^3 nodes arranged in an $n \times n \times n$ grid, with edges directed top-to-bottom, left-to-right, and front-to-back. The computation takes $3n^2$ inputs to the nodes on the front, left, and top faces, and returns $3n^2$ outputs from the nodes on the back, right, and bottom faces. By partitioning into appropriate blocks, the ordered 3D grid dag can be computed with BSP costs $W = O(n^3/p)$, $H = O(n^2/p^{1/2})$, $S = O(p^{1/2})$ (see [57]).

Further Algorithms

The design of efficient BSP algorithms has become a well-established topic. Some examples of BSP algorithms proposed in the past include list and tree contraction [29], sorting [28, 30, 32, 70, 77], convex hull

computation [15, 21, 23, 32, 45, 65, 73], and selection [28, 76]. In the area of matrix algorithms, some examples of the proposed algorithms include matrix–vector and matrix–matrix multiplication [9, 56, 59], Strassen-type matrix multiplication [59], triangular system solution and several versions of Gaussian elimination [30, 56, 75], and orthogonal matrix decomposition [13, 75]. In the area of graph algorithms, some examples of the proposed algorithms include Boolean matrix multiplication [69], minimum spanning tree [12], transitive closure [2], the algebraic path problem and the all-pairs shortest path problems [71], and graph coloring [27]. In the area of string algorithms, some examples of the proposed algorithms include the longest common subsequence and edit distance problems [3–5, 51, 53, 74], and the longest increasing subsequence problem [52].

BSP Programming

The BSPlib Standard

Based on the experience of early BSP programming tools [34, 58, 60], the BSP programming community agreed on a common library standard *BSPlib* [42]. The aim of BSPlib is to provide a set of BSP programming primitives, striking a reasonable balance between simplicity and efficiency. BSPlib is based on the *single program/multiple data* (SPMD) programming model, and contains communication primitives for *direct remote memory access* (DRMA) and *bulk-synchronous message passing* (BSMP). Experience shows that DRMA, due to its simplicity and deadlock-free semantics, is the method of choice for all but the most irregular applications. Routine use of DRMA is made possible by the barrier synchronization structure of BSP computation.

The two currently existing major implementations of BSPlib are *the Oxford BSP toolset* [63] and *the PUB library* [11]. Both provide a robust environment for the development of BSPlib applications, including mechanisms for optimizing communication [22, 43], load balancing, and fault tolerance [40]. The PUB library also provides a few additional primitives (oblivious synchronization, processor subsets, multithreading). Both the Oxford BSP toolset and the PUB library include tools for performance analysis and prediction. Recently, new approaches have been developed to BSPlib implementation [48, 50, 62] and performance analysis [41, 80].

Beyond BSPlib

The Message Passing Interface (MPI) is currently the most widely accepted standard of distributed-memory parallel programming. In contrast to BSPlib, which is based on a single programming paradigm, MPI provides a diverse set of parallel programming patterns, allowing the programmer to pick-and-choose a paradigm most suitable for the application. Consequently, the number of primitives in MPI is an order of magnitude larger than in BSPlib, and the responsibility to choose the correct subset of primitives and to structure the code rests with the programmer. It is not surprising that a carefully chosen subset of MPI can be used to program in the BSP style; an example of such an approach is given by [8].

The ZPL language [14] is a global-view array language based on a BSP-like computation structure. As such, it can be considered to be one of the earliest high-level BSP programming tools. Another growing trend is the integration of the BSP model with modern programming environments. A successful example of integrating BSP with Python is given by the package Scientific Python [44], which provides high-level facilities for writing BSP code, and performs communication by calls to either BSPlib or MPI. Tools for BSP programming in Java have been developed by projects NestStep [49] and JBSP [35]; a Java-like multi-threaded BSP programming model is proposed in [72]. A functional programming model for BSP is given by the BSMLlib library [36]; a constraint programming approach is introduced in [6]. Projects InteGrade [31] and GridNestStep [55] are aiming to implement the BSP model using Grid technology.

Related Entries

- ▶ [Bandwidth-Latency Models \(BSP, LogP\)](#)
- ▶ [Collective Communication](#)
- ▶ [Dense Linear System Solvers](#)
- ▶ [Functional Languages](#)
- ▶ [Graph Algorithms](#)
- ▶ [Load Balancing, Distributed Memory](#)
- ▶ [Linear Algebra, Numerical](#)
- ▶ [Models of Computation, Theoretical](#)
- ▶ [Parallel Skeletons](#)
- ▶ [PGAS \(Partitioned Global Address Space\) Languages](#)
- ▶ [PRAM \(Parallel Random Access Machines\)](#)

► Reduce and Scan

► Sorting

► SPMD Computational Model

► Synchronization

► ZPL

Bibliographic Notes and Further Reading

A detailed treatment of the BSP model, BSP programming, and several important BSP algorithms is given in the monograph [8]. Collection [18] includes several chapters dedicated to BSP and related models.

Bibliography

1. Alverson GA, Griswold WG, Lin C, Notkin D, Snyder L (1998) Abstractions for portable, scalable parallel programming. *IEEE Trans Parallel Distribut Syst* 9(1):71–86
2. Alves CER, Cáceres EN, Castro Jr AA, Song SW, Szwarcfiter JL (2003) Efficient parallel implementation of transitive closure of digraphs. In: Proceedings of EuroPVM/MPI, Venice. Lecture notes in computer science, vol 2840. Springer, pp 126–133
3. Alves CER, Cáceres EN, Dehne F, Song SW (2002) Parallel dynamic programming for solving the string editing problem on a CGM/BSP. In: Proceedings of the 14th ACM SPAA, Winnipeg, pp 275–281
4. Alves CER, Cáceres EN, Dehne F, Song SW (2003) A parallel wavefront algorithm for efficient biological sequence comparison. In: Proceedings of ICCSA, Montreal. Lecture notes in computer science, vol 2668. Springer, Berlin, pp 249–258
5. Alves CER, Cáceres EN, Song SW (2006) A coarse-grained parallel algorithm for the all-substrings longest common subsequence problem. *Algorithmica* 45(3):301–335
6. Ballereau O, Hains G, Lallouet A (2002) BSP constraint programming. In: Gorlatch S, Lengauer C (eds) Constructive methods for parallel programming, vol 10. Advances in computation: Theory and practice. Nova Science, New York, Chap 7
7. Bäumker A, Dittrich W, Meyer auf der Heide F (1998) Truly efficient parallel algorithms: l-optimal multisearch for an extension of the BSP model. *Theor Comput Sci* 203(2):175–203
8. Bisseling RH (2004) Parallel scientific computation: A structured approach using BSP and MPI. Oxford University Press, New York
9. Bisseling RH, McColl WF (1993) Scientific computing on bulk synchronous parallel architectures. Preprint 836, Department of Mathematics, University of Utrecht, December 1993
10. Blanco V, González JA, León C, Rodríguez C, Rodríguez G, Printista M (2004) Predicting the performance of parallel programs. *Parallel Comput* 30:337–356
11. Bonorden O, Juurlink B, von Otte I, Rieping I (2003) The Paderborn University BSP (PUB) library. *Parallel Comput* 29(2): 187–207

12. Cáceres EN, Dehne F, Mongelli H, Song SW, Szwarcfiter JL (2004) A coarse-grained parallel algorithm for spanning tree and connected components. In: Proceedings of Euro-Par, Pisa. Lecture notes in computer science, vol 3149. Springer, Berlin, pp 828–831
13. Calinescu R, Evans DJ (1997) Bulk-synchronous parallel algorithms for QR and QZ matrix factorisation. *Parallel Algorithms Appl* 11:97–112
14. Chamberlain BL, Choi S-E, Lewis EC, Lin C, Snyder L, Weathersby WD (2000) ZPL: A machine independent programming language for parallel computers. *IEEE Trans Softw Eng* 26(3): 197–211
15. Cinque L, Di Maggio C (2001) A BSP realisation of Jarvis' algorithm. *Pattern Recogn Lett* 22(2):147–155
16. Cole M (1999) Algorithmic skeletons. In: Hammond K, Michaelson G (eds) Research Directions in Parallel Functional Programming. Springer, London, pp 289–303
17. Cole M (2004) Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Comput* 30:389–406
18. Corrêa R et al (eds) (2002) Models for parallel and distributed computation: theory, algorithmic techniques and applications, vol 67. Applied Optimization. Kluwer, Dordrecht
19. Dehne F, Dittrich W, Hutchinson D (2003) Efficient external memory algorithms by simulating coarse-grained parallel algorithms. *Algorithmica* 36(2):97–122
20. Dehne F, Fabri A, Rau-Chaplin A (1996) Scalable parallel computational geometry for coarse grained multicomputers. *Int J Comput Geom* 6:379–400
21. Diallo M, Ferreira A, Rau-Chaplin A, Ubéda S (1999) Scalable 2D convex hull and triangulation algorithms for coarse grained multicomputers. *J Parallel Distrib Comput* 56:47–70
22. Donaldson SR, Hill JMD, Skillicorn D (1999) Predictable communication on unpredictable networks: Implementing BSP over TCP/IP and UDP/IP. *Concurr Pract Exp* 11(II):687–700
23. Dymond P, Zhou J, Deng X (2001) A 2D parallel convex hull algorithm with optimal communication phases. *Parallel Comput* 27(3):243–255
24. Fantozzi C, Pietracaprina A, Pucci G (2003) A general PRAM simulation scheme for clustered machines. *Int J Foundations Comput Sci* 14(6):1147–1164
25. Fortune S, Wyllie J (1978) Parallelism in random access machines. In: Proceedings of ACM STOC, San Diego, pp 114–118
26. Gebremedhin AH, Essaïdi M, Lassous GI, Gustedt J, Telle JA (2006) PRO: A model for the design and analysis of efficient and scalable parallel algorithms. *Nordic J Comput* 13(4):215–239
27. Gebremedhin AH, Manne F (2000) Scalable parallel graph coloring algorithms. *Concurr Pract Exp* 12(12):1131–1146
28. Gerbessiotis AV, Siniolakis CJ (1996) Deterministic sorting and randomized median finding on the BSP model. In: Proceedings of the 8th ACM SPAA, Padua, pp 223–232
29. Gerbessiotis AV, Siniolakis CJ, Tiskin A (2002) Parallel priority queue and list contraction: The BSP approach. *Comput Informatics* 21:59–90
30. Gerbessiotis AV, Valiant LG (1994) Direct bulk-synchronous parallel algorithms. *J Parallel Distrib Comput* 33(2):251–267
31. Goldchleger A, Kon F, Goldman A, Finger M, Bezerra GC. InteGrade: Object-oriented Grid middleware leveraging idle computing power of desktop machines. *Concurr Comput Pract Exp* 16:449–454
32. Goodrich M (1996) Communication-efficient parallel sorting. In: Proceedings of the 28th ACM STOC, Philadelphia, pp 247–256
33. Gorlatch S, Lengauer C (eds) (2002) Constructive methods for parallel programming, vol 10. Advances in computation: Theory and practice. Nova Science, New York
34. Goudreau MW, Lang K, Rao SB, Suel T, Tsantilas T (1999) Portable and efficient parallel computing using the BSP model. *IEEE Trans Comput* 48(7):670–689
35. Gu Y, Lee B-S, Cai W (2001) JBSP: A BSP programming library in Java. *J Parallel Distrib Comput* 61:1126–1142
36. Hains G, Loulergue F (2002) Functional bulk synchronous parallel programming using the BSMLlib library. In: Gorlatch S, Lengauer C (eds) Constructive methods for parallel programming, vol 10. Advances in computation: Theory and practice. Nova Science, New York, Chap 11
37. Hammond K, Michaelson G (eds) (1999) Research directions in parallel functional programming. Springer, London
38. Heywood T, Ranka S (1992) A practical hierarchical model of parallel computation I: The model. *J Parallel Distrib Comput* 16(3):212–232
39. Hill J (1999) Portability of performance in the BSP model. In: Hammond K, Michaelson G (eds) Research directions in parallel functional programming. Springer, London, pp 267–287
40. Hill JMD, Donaldson SR, Lanfear T (1998) Process migration and fault tolerance of BSPlib programs running on a network of workstations. In: Pritchard D, Reeve J (eds) Proceedings of Euro-Par, Southampton. Lecture notes in computer science, vol 1470. Springer, Berlin, pp 80–91
41. Hill JMD, Jarvis SA, Siniolakis C, Vasilev VP (1998) Analysing an SQL application with a BSPlib call-graph profiling tool. In: Pritchard D, Reeve J (eds) Proceedings of Euro-Par. Lecture notes in computer science, vol 1470. Springer, Berlin, pp 157–165
42. Hill JMD, McColl WF, Stefanescu DC, Goudreau MW, Lang K, Rao SB, Suel T, Tsantilas T, Bisseling RH (1998) BSPlib: The BSP programming library. *Parallel Comput* 24(14):1947–1980
43. Hill JMD, Skillicorn DB (1998) Lessons learned from implementing BSP. *Future Generation Comput Syst* 13(4–5):327–335
44. Hinsen K (2003) High-level parallel software development with Python and BSP. *Parallel Process Lett* 13(3):473–484
45. Ishimizu T, Fujiwara A, Inoue M, Masuzawa T, Fujiwara H (2002) Parallel algorithms for selection on the BSP and BSP* models. *Syst Comput Jpn* 33(12):97–107
46. Juurlink BHH, Wijshoff HAG (1996) The E-BSP model: Incorporating general locality and unbalanced communication into the BSP model. In: Bougé et al (eds) Proceedings of Euro-Par (Part II), Lyon. Lecture notes in computer science, vol 1124. Springer, Berlin, pp 339–347
47. Juurlink BHH, Wijshoff HAG (1998) A quantitative comparison of parallel computation models. *ACM Trans Comput Syst* 16(3): 271–318

48. Kee Y, Ha S (2002) An efficient implementation of the BSP programming library for VIA. *Parallel Process Lett* 12(1):65–77
49. Keßler CW (2000) NestStep: Nested parallelism and virtual shared memory for the BSP model. *J Supercomput* 17:245–262
50. Kim S-R, Park K (2000) Fully-scalable fault-tolerant simulations for BSP and CGM. *J Parallel Distrib Comput* 60(12):1531–1560
51. Krusche P, Tiskin A (2006) Efficient longest common subsequence computation using bulk-synchronous parallelism. In: Proceedings of ICCSA, Glasgow. Lecture notes in computer science, vol 3984. Springer, Berlin, pp 165–174
52. Krusche P, Tiskin A (2010) Longest increasing subsequences in scalable time and memory. In: Proceedings of PPAM 2009, Revised Selected Papers, Part I. Lecture notes in computer science, vol 6067. Springer, Berlin, pp 176–185
53. Krusche P, Tiskin A (2010) New algorithms for efficient parallel string comparison. In: Proceedings of ACM SPAA, Santorini, pp 209–216
54. Lecomber DS, Siniolakis CJ, Sujithan KR (2000) PRAM programming: in theory and in practice. *Concurr Pract Exp* 12:211–226
55. Mattsson H, Kessler CW (2004) Towards a virtual shared memory programming environment for grids. In: Proceedings of PARA, Copenhagen. Lecture notes in computer science, vol 3732. Springer-Verlag, pp 519–526
56. McCol W F (1995) Scalable computing. In: van Leeuwen J (ed) Computer science today: Recent trends and developments. Lecture notes in computer science, vol 1000. Springer, Berlin, pp 46–61
57. McCol W F (1996) Universal computing. In: L. Bougé et al (eds) Proceedings of Euro-Par (Part I), Lyon. Lecture notes in computer science, vol 1123. Springer, Berlin, pp 25–36
58. McCol W F, Miller Q (1995) Development of the GPL language. Technical report (ESPRIT GEPPCOM project), Oxford University Computing Laboratory
59. McCol W F, Tiskin A (1999) Memory-efficient matrix multiplication in the BSP model. *Algorithmica* 24(3/4):287–297
60. Miller R (1999) A library for bulk-synchronous parallel programming. In: Proceeding of general purpose parallel computing, British Computer Society, pp 100–108
61. Morin P (2000) Coarse grained parallel computing on heterogeneous systems. In: Proceedings of ACM SAC, Como, pp 628–634
62. Nibhanupudi MV, Szymanski BK (1998) Adaptive bulk-synchronous parallelism on a network of non-dedicated workstations. In: High performance computing systems and applications. Kluwer, pp 439–452
63. The Oxford BSP Toolset (1998) <http://www.bsp-worldwide.org/implmnts/oxtool>
64. Ramachandran V (1999) A general purpose shared-memory model for parallel computation. In: Heath MT, Ranade A, Schreiber RS (eds) Algorithms for parallel processing. IMA volumes in mathematics and applications, vol 105. Springer-Verlag, New York
65. Saukas LG, Song SW (1999) A note on parallel selection on coarse-grained multicomputers. *Algorithmica*, 24(3–4):371–380, 1999
66. Sibeyn J, Kaufmann M (1997) BSP-like external-memory computation. In: Bongiovanni GC, Bovet DP, Di Battista G (eds) Proceedings of CIAC, Rome. Lecture notes in computer science, vol 1203. Springer, Berlin, pp 229–240
67. Skillicorn DB (2002) Predictable parallel performance: The BSP model. In: Corrêa R et al (eds) Models for parallel and distributed computation: theory, algorithmic techniques and applications, vol 67. Applied Optimization. Kluwer, Dordrecht, pp 85–115
68. Song SW (2002) Parallel graph algorithms for coarse-grained multicomputers. In: Corrêa R et al (eds) Models for parallel and distributed computation: theory, algorithmic techniques and applications, vol 67, Applied optimization. Kluwer, Dordrecht, pp 147–178
69. Tiskin A (1998) Bulk-synchronous parallel multiplication of Boolean matrices. In: Proceedings of ICALP Aalborg. Lecture notes in computer science, vol 1443. Springer, Berlin, pp 494–506
70. Tiskin A (1998) The bulk-synchronous parallel random access machine. *Theor Comput Sci* 196(1–2):109–130
71. Tiskin A (2001) All-pairs shortest paths computation in the BSP model. In: Proceedings of ICALP, Crete. Lecture notes in computer science, vol 2076. Springer, Berlin, pp 178–189
72. Tiskin A (2001) A new way to divide and conquer. *Parallel Process Lett* 11(4):409–422
73. Tiskin A (2002) Parallel convex hull computation by generalised regular sampling. In: Proceedings of Euro-Par, Paderborn. Lecture notes in computer science, vol 2400. Springer, Berlin, pp 392–399
74. Tiskin A (2005) Efficient representation and parallel computation of string-substring longest common subsequences. In: Proceedings of ParCo Malaga. NIC Series, vol 33. John von Neumann Institute for Computing, pp 827–834
75. Tiskin A (2007) Communication-efficient parallel generic pairwise elimination. *Future Generation Comput Syst* 23:179–188
76. Tiskin A (2010) Parallel selection by regular sampling. In: Proceedings of Euro-Par (Part II), Ischia. Lecture notes in computer science, vol 6272. Springer, pp 393–399
77. Valiant LG (1990) A bridging model for parallel computation. *Commun ACM* 33(8):103–111
78. Vasilev V (2003) BSPGRID: Variable resources parallel computation and multiprogrammed parallelism. *Parallel Process Lett* 13(3):329–340
79. Williams TL, Parsons RJ (2000) The heterogeneous bulk synchronous parallel model. In: Rolim J et al (eds) Proceedings of IPDPS workshops Cancun. Lecture notes in computer science, vol 1800. Springer, Berlin, pp 102–108
80. Zheng W, Khan S, Xie H (1999) BSP performance analysis and prediction: Tools and applications. In: Malyshkin V (ed) Proceedings of PaCT, Newport Beach. Lecture notes in computer science, vol 1662. Springer, Berlin, pp 313–319

Bulk Synchronous Parallelism (BSP)

► [BSP \(Bulk Synchronous Parallelism\)](#)

Bus: Shared Channel

► Buses and Crossbars

RAJEEV BALASUBRAMONIAN¹, TIMOTHY M. PINKSTON²

¹University of Utah, Salt Lake City, UT, USA

²University of Southern California, Los Angeles, CA, USA

Synonyms

Bus: Shared channel; Shared interconnect; Shared-medium network; Crossbar; Interconnection network; Point-to-point switch; Switched-medium network

Definition

Bus: A bus is a shared interconnect used for connecting multiple components of a computer on a single chip or across multiple chips. Connected entities either place signals on the bus or listen to signals being transmitted on the bus, but signals from only one entity at a time can be transported by the bus at any given time. Buses are popular communication media for broadcasts in computer systems.

Crossbar: A crossbar is a non-blocking switching element with N inputs and M outputs used for connecting multiple components of a computer where, typically, $N = M$. The crossbar can simultaneously transport signals on any of the N inputs to any of the M outputs as long as multiple signals do not compete for the same input or output port. Crossbars are commonly used as basic switching elements in switched-media network routers.

Discussion

Introduction

Every computer system is made up of numerous components such as processor chips, memory chips, peripherals, etc. These components communicate with each other via interconnects. One of the simplest interconnects used in computer systems is the *bus*. The bus is a shared medium (usually a collection of electrical

wires) that allows one sender at a time to communicate with all sharers of that medium. If the interconnect must support multiple simultaneous senders, more scalable designs based on switched-media must be pursued. The *crossbar* represents a basic switched-media building block for more complex but scalable networks.

In traditional multi-chip multiprocessor systems (c. 2000), buses were primarily used as off-chip interconnects, for example, front-side buses. Similarly, crossbar functionality was implemented on chips that were used mainly for networking. However, the move to multi-core technology has necessitated the use of networks even within a mainstream processor chip to connect its multiple cores and cache banks. Therefore, buses and crossbars are now used within mainstream processor chips as well as chip sets. The design constraints for on-chip buses are very different from those of off-chip buses. Much of this discussion will focus on on-chip buses, which continue to be the subject of much research and development.

Basics of Bus Design

A bus comprises a shared medium with connections to multiple entities. An interface circuit allows each of the entities either to place signals on the medium or sense (listen to) the signals already present on the medium. In a typical communication, one of the entities acquires ownership of the bus (the entity is now known as the *bus master*) and places signals on the bus. Every other entity senses these signals and, depending on the content of the signals, may choose to accept or discard them. Most buses today are *synchronous*, that is, the start and end of a transmission are clearly defined by the edges of a shared clock. An *asynchronous* bus would require an acknowledgment from the receiver so the sender knows when the bus can be relinquished.

Buses often are collections of electrical wires, (Alternatively, buses can be a collection of optical waveguides over which information is transmitted phototonically [9]) where each wire is typically organized as “data,” “address,” or “control.” In most systems, networks are used to move messages among entities on the *data* bus; the *address* bus specifies the entity that must receive the message; and the *control* bus carries auxiliary signals such as arbitration requests and error correction codes. There is another nomenclature that readers may also encounter. If the network is used to implement a

cache coherence protocol, the protocol itself has three types of messages: (1) *DATA*, which refers to blocks of memory; (2) *ADDRESS*, which refers to the memory block's address; and (3) *CONTROL*, which refers to auxiliary messages in the protocol such as acknowledgments. Capitalized terms as above will be used to distinguish message types in the coherence protocol from signal types on the bus. For now, it will be assumed that all three protocol message types are transmitted on the *data* bus.

Arbitration Protocols

Since a bus is a shared medium that allows a single master at a time, an arbitration protocol is required to identify this bus master. A simple arbitration protocol can allow every entity to have ownership of the bus for a fixed time quantum, in a round-robin manner. Thus, every entity can make a local decision on when to transmit. However, this wastes bus bandwidth when an entity has nothing to transmit during its turn.

The most common arbitration protocol employs a central arbiter; entities must send their bus requests to the arbiter and the arbiter sends explicit messages to grant the bus to requesters. If the requesting entity is not aware of its data bus occupancy time beforehand, the entity must also send a bus release message to the arbiter after it is done. The request, grant, and release signals are part of the control network. The request signal is usually carried on a dedicated wire between an entity and the arbiter. The grant signal can also be implemented similarly, or as a shared bus that carries the ID of the grantee. The arbiter has state to track data bus occupancy, buffers to store pending requests, and policies to implement priority or fairness. The use of pipelining to hide arbitration delays will be discussed shortly.

Arbitration can also be done in a distributed manner [5], but such methods often incur latency or bandwidth penalties. In one example, a shared arbitration bus is implemented with wired-OR signals. Multiple entities can place a signal on the bus; if any entity places a “one” on the bus, the bus carries “one,” thus using wires to implement the logic of an *OR* gate. To arbitrate, all entities place their IDs on the arbitration bus; the resulting signal is the OR of all requesting IDs. The bus is granted to the entity with the largest ID and this is determined by having each entity sequentially drop

out if it can determine that it is not the largest ID in the competition.

B

Pipelined Bus

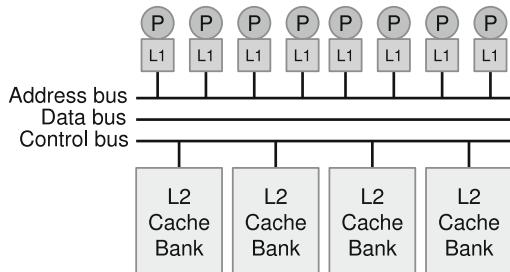
Before a bus transaction can begin, an entity must arbitrate for the bus, typically by contacting a centralized arbiter. The latency of request and grant signals can be hidden with pipelining. In essence, the arbitration process (that is handled on the *control* bus) is overlapped with the data transmission of an earlier message. An entity can send a bus request to the arbiter at any time. The arbiter buffers this request, keeps track of occupancy on the *data* bus, and sends the grant signal one cycle before the *data* bus will be free. In a heavily loaded network, the *data* bus will therefore rarely be idle and the arbitration delay is completely hidden by the wait for the *data* bus. In a lightly loaded network, pipelining will not hide the arbitration delay, which is typically at least three cycles: one cycle for the request signal, one cycle for logic at the arbiter, and one cycle for the grant signal.

Case Study: Snooping-Based Cache Coherence Protocols

As stated earlier, the bus is a vehicle for transmission of messages within a higher-level protocol such as a cache coherence protocol. A single *transaction* within the higher-level protocol may require multiple messages on the bus. Very often, the higher-level protocol and the bus are codesigned to improve efficiency. Therefore, as a case study, a snooping bus-based coherence protocol will be discussed.

Consider a single-chip multiprocessor where each processor core has a private L1 cache, and a large L2 cache is shared by all the cores. The multiple L1 caches and the multiple banks of the L2 cache are the entities connected to a shared bus (Fig. 1). The higher-level coherence protocol ensures that data in the L1 and L2 caches is kept coherent, that is, a data modification is eventually seen by all caches and multiple updates to one block are seen by all caches in exactly the same order.

A number of coherence protocol operations will now be discussed. When a core does not find its data in its local L1 cache, it must send a request for the data block to other L1 caches and the L2 cache. The core's L1 cache first sends an arbitration request for the bus to the



Buses and Crossbars. Fig. 1 Cores and L2 cache banks connected with a bus. The bus is composed of wires that handle data, address, and control

arbiter. The arbiter eventually sends the grant signal to the requesting L1. The arbitration is done on the *control* portion of the bus. The L1 then places the ADDRESS of the requested data block on the *data* bus. On a synchronous bus, we are guaranteed that every other entity has seen the request within one bus cycle. Each such “snooping” entity now checks its L1 cache or L2 bank to see if it has a copy of the requested block. Since every lookup may take a different amount of time, a wired-AND signal is provided within the *control* bus so everyone knows that the snoop is completed. This is an example of bus and protocol codesign (a protocol CONTROL message being implemented on the bus’ *control* bus). The protocol requires that an L1 cache respond with data if it has the block in “modified” state, else, the L2 cache responds with data. This is determined with a wired-OR signal; all L1 caches place the outcome of their snoop on this wired-OR signal and the L2 cache accordingly determines if it must respond. The responding entity then fetches data from its arrays and places it on the *data* bus. Since the bus is not released until the end of the entire coherence protocol transaction, the responder knows that the *data* bus is idle and need not engage in arbitration (another example of protocol and bus codesign). *Control* signals let the requester know that the data is available and the requester reads the cache block off the bus.

The use of a bus greatly simplifies the coherence protocol. It serves as a serialization point for all coherence transactions. The timing of when an operation is visible to everyone is well known. The broadcast of operations allows every cache on the bus to be self-managing. Snooping bus-based protocols are therefore

much simpler than directory-based protocols on more scalable networks.

As described, each coherence transaction is handled atomically, that is, one transaction is handled completely before the bus is released for use by other transactions. This means that the *data* bus is often idle while caches perform their snoops and array reads. Bus utilization can be improved with a *split transaction bus*. Once the requester has placed its request on the *data* bus, the *data* bus is released for use by other transactions. Other transactions can now use the *data* bus for their requests or responses. When a transaction’s response is ready, the *data* bus must be arbitrated for. Every request and response must now carry a small tag so responses can be matched up to their requests. Additional tags may also be required to match the wired-OR signals to the request.

The split transaction bus design can be taken one step further. Separate buses can be implemented for ADDRESS and DATA messages. All requests (ADDRESS messages) and corresponding wired-OR CONTROL signals are carried on one bus. This bus acts as the serialization point for the coherence protocol. Responders always use a separate bus to return DATA messages. Each bus has its own separate arbiter and corresponding control signals.

Bus Scalability

A primary concern with any bus is its lack of scalability. First, if many entities are connected to a bus, the bus speed reduces because it must drive a heavier load over a longer distance. In an electrical bus, the higher capacitive load from multiple entities increases the RC-delay; in an optical bus, the reduced photons received at photodetectors from dividing the optical power budget among multiple entities likewise increases the time to detect bus signals. Second, with many entities competing for the shared bus, the wait-time to access the bus increases with the number of entities. Therefore, conventional wisdom states that more scalable switched-media networks are preferred when connecting much more than 8 or 16 entities [4]. However, the simplicity of bus-based protocols (such as the snooping-based cache coherence protocol) make it attractive for small- or medium-scale symmetric multiprocessing (SMP) systems. For example, the IBM POWER7™ processor chip supports 8 cores on its SMP bus [12]. Buses

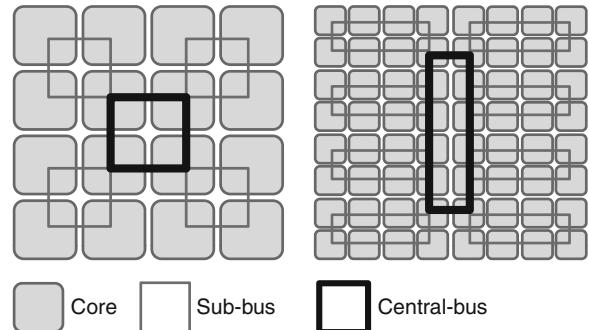
are also attractive because, unlike switched-media networks, they do not require energy-hungry structures such as buffers and crossbars. Researchers have considered multiple innovations to extend the scalability of buses, some of which are discussed next.

One way to scale the number of entities connected using buses is, simply, to provide multiple buses, for example, dual-independent buses or quad-independent buses. This mitigates the second problem listed above regarding the high rate of contention on a single bus, but steps must still be taken to maintain cache coherency via snooping on the buses. The Sun Starfire multiprocessor [3], for example, uses four parallel buses for ADDRESS requests, wherein each bus handles a different range of addresses. Tens of dedicated buses are used to connect up to 32 IBM POWER7TM processor chips in a coherent SMP system [12]. While this option has high cost for off-chip buses because of pin and wiring limitations, a multi-bus for an on-chip network is not as onerous because of plentiful metal area budgets.

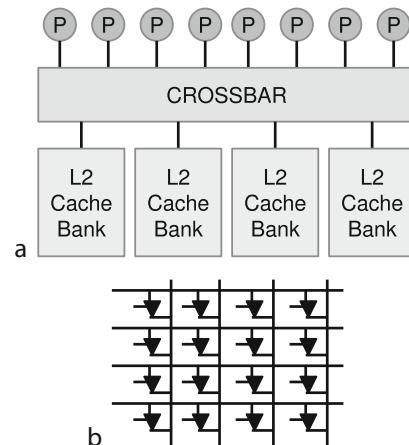
Some recent works have highlighted the potential of bus-based on-chip networks. Das et al. [6] argue that buses should be used within a relatively small cluster of cores because of their superior latency, power, and simplicity. The buses are connected with a routed mesh network that is employed for communication beyond the cluster. The mesh network is exercised infrequently because most applications exhibit locality. Udupi et al. [14] take this hierarchical network approach one step further. As shown in Fig. 2, the intra-cluster buses are themselves connected with an inter-cluster bus. Bloom filters are used to track the buses that have previously handled a given address. When coherence transactions are initiated for that address, the Bloom filters ensure that the transaction is broadcasted only to the buses that may find the address relevant. Locality optimizations such as page coloring help ensure that bus broadcasts do not travel far, on average. Udupi et al. also employ multiple buses and low-swing wiring to further extend bus scalability in terms of performance and energy.

Crossbars

Buses are used as a shared fabric for communication among multiple entities. Communication on a bus



Buses and Crossbars. Fig. 2 A hierarchical bus structure that localizes broadcasts to relevant clusters



Buses and Crossbars. Fig. 3 (a) A "dance-hall" configuration of processors and memory. (b) The circuit for a 4 × 4 crossbar

is always broadcast-style, *i.e.*, even though a message is going from entity-A to entity-B, all entities see the message and no other message can be simultaneously in transit. However, if the entities form a "dance hall" configuration (Fig. 3a) with processors on one side and memory on the other side, and most communication is between processors and memory, a crossbar interconnect becomes a compelling choice. Although crossbars incur a higher wiring overhead than buses, they allow multiple messages simultaneously to be in transit, thus increasing the network bandwidth. Given this, crossbars serve as the basic switching element within switched-media network routers.

A crossbar circuit takes N inputs and connects each input to any of the M possible outputs. As shown in Fig. 3b, the circuit is organized as a grid of wires, with inputs on the left, and outputs on the bottom. Each wire can be thought of as a bus with a unique master, that is, the associated input port. At every intersection of wires, a pass transistor serves as a *crosspoint connector* to short the two wires, if enabled, connecting the input to the output. Small buffers can also be located at the crosspoints in *buffered crossbar* implementations to store messages temporarily in the event of contention for the intended output port. A crossbar is usually controlled by a centralized arbiter that takes output port requests from incoming messages and computes a viable assignment of input port to output port connections. This, for example, can be done in a crossbar *switch allocation* stage prior to a crossbar *switch traversal* stage for message transport. Multiple messages can be simultaneously in transit as long as each message is headed to a unique output and each emanates from a unique input. Thus, the crossbar is *non-blocking*. Some implementations allow a single input message to be routed to multiple output ports.

A crossbar circuit has a cost that is proportional to $N \times M$. The circuit is replicated W times, where W represents the width of the link at one of the input ports. It is therefore not a very scalable circuit. In fact, larger centralized switches such as Butterfly and Benes switch fabrics are constructed hierarchically from smaller crossbars to form *multistage indirect networks* or *MINs*. Such networks have a cost that is proportional to $N \log(M)$ but have a more restrictive set of messages that can be routed simultaneously without blocking.

A well-known example of a large-scale on-chip crossbar is the Sun Niagara processor [10]. The crossbar connects eight processors to four L2 cache banks in a “dance-hall” configuration. A recent example of using a crossbar to interconnect processor cores in other switched point-to-point configurations is the Intel QuickPath Interconnect [7]. More generally, crossbars find extensive use in network routers. Meshes and tori, for example, implement a 5×5 crossbar in router switches where the five input and output ports correspond to the North, South, East, West neighbors and the node connected to each router. The mesh-connected Tilera Tile-GxTM 100-core processor is a recent example [13].

Related Entries

- ▶ Cache Coherence
- ▶ Collective Communication
- ▶ Interconnection Networks
- ▶ Networks, Direct
- ▶ Network Interfaces
- ▶ Networks, Multistage
- ▶ PCI-Express
- ▶ Routing (Including Deadlock Avoidance)
- ▶ Switch Architecture
- ▶ Switching Techniques

Bibliographic Notes and Further Reading

For more details on bus design and other networks, readers are referred to the excellent textbook by Dally and Towles [5]. Recent papers in the architecture community that focus on bus design include those by Udupi et al. [14], Das et al. [6], and Kumar et al. [11]. Kumar et al. [11] articulate some of the costs of implementing buses and crossbars in multi-core processors and argue that the network must be codesigned with the core and caches for optimal performance and power. A few years back, S. Borkar made a compelling argument for the widespread use of buses within multi-core chips that is highly thought provoking [1, 2]. The paper by Charlesworth [3] on Sun’s Starfire, while more than a decade old, is an excellent reference that describes considerations when designing a high-performance bus for a multi-chip multiprocessor. Future many-core processors may adopt photonic interconnects to satisfy the high memory bandwidth demands of the many cores. A single photonic waveguide can carry many wavelengths of light, each carrying a stream of data. Many receiver “rings” can listen to the data transmission, each ring contributing to some loss in optical energy. The Corona paper by Vantrease et al. [15] and the paper by Kirman et al. [9] are excellent references for more details on silicon photonics, optical buses, and optical crossbars.

The basic crossbar circuit has undergone little change over the last several years. However, given the recent interest in high-radix routers which increase the input/output-port degree of the crossbar used as the internal router switch, Kim et al. [8] proposed hierarchical crossbar and buffered crossbar organizations to facilitate scalability. Also, given the relatively recent shift

in focus to energy-efficient on-chip networks, Wang et al. [16] proposed techniques to reduce the energy usage within crossbar circuits. They introduced a cut-through crossbar that is optimized for traffic that travels in a straight line through a mesh network's router. The design places some restrictions on the types of message turns that can be simultaneously handled. Wang et al. also introduce a segmented crossbar that prevents switching across the entire length of wires when possible.

Bibliography

1. Borkar S (2007) Networks for multi-core chips – a contrarian view, ISLPED Keynote. www.islped.org/X2007/BorkarISLPED07.pdf
2. Borkar S (2006) Networks for multi-core chips – a controversial view. In: Workshop on on- and off-chip interconnection networks for multicore systems (OCIN), Stanford
3. Charlesworth A (1998) Starfire: extending the SMP envelope. *IEEE Micro* 18(1):39–49
4. Dally W, Towles B (2001) Route packets, not wires: on-chip interconnection networks. In: Proceedings of DAC, Las Vegas
5. Dally W, Towles B (2003) Principles and practices of interconnection networks, 1st edn. Morgan Kaufmann, San Francisco
6. Das R, Eachempati S, Mishra AK, Vijaykrishnan N, Das CR (2009) Design and evaluation of hierarchical on-chip network topologies for next generation CMPs. In: Proceedings of HPCA, Raleigh
7. Intel Corp. An introduction to the Intel QuickPath interconnect. <http://www.intel.com/technology/quickpath/introduction.pdf>
8. Kim J, Dally W, Towles B, Gupta A (2005) Microarchitecture of a high-radix router. In: Proceedings of ISCA, Madison
9. Kirman N, Kyrman M, Dokania R, Martinez J, Apsel A, Watkins M, Albonesi D (2006) Leveraging optical technology in future bus-based chip multiprocessors. In: Proceedings of MICRO, Orlando
10. Kongetira P (2004) A 32-way multithreaded SPARC processor. In: Proceedings of hot chips 16, Stanford. <http://www.hotchips.org/archives/>
11. Kumar R, Zyuban V, Tullsen D (2005) Interconnections in multicore architectures: understanding mechanisms, overheads, and scaling. In: Proceedings of ISCA, Madison
12. Tendler JM (2010) POWER7 processors: the beat goes on. <http://www.ibm.com/developerworks/wikis/download/attachments/104533501/POWER7++The+Beat+Goes+On.pdf>
13. Tilera. TILE-Gx processor family product brief. http://www.tilera.com/sites/default/files/productbriefs/PB025_Processor_A_v3.pdf
14. Udipi A, Muralimanohar N, Balasubramonian R (2010) Towards scalable, energy-efficient, bus-based on-chip networks. In: Proceedings of HPCA, Bangalore
15. Vantrease D, Schreiber R, Monchiero M, McLaren M, Jouppi N, Fiorentino M, Davis A, Binkert N, Beausoleil R, Ahn J-H (2008) Corona: system implications of emerging nanophotonic technology. In: Proceedings of ISCA, Beijing
16. Wang H-S, Peh L-S, Malik S (2003) Power-driven design of router microarchitectures in on-chip networks. In: Proceedings of MICRO, San Diego

Butterfly

► Networks, Multistage

C

C*

GUY L. STEELE JR.
Oracle Labs, Burlington, MA, USA

Definition

C* (pronounced “see-star”) refers to two distinct data-parallel dialects of C developed by Thinking Machines Corporation for its Connection Machine supercomputers. The first version (1987) is organized around the declaration of *domains*, similar to classes in C++, but when code associated with a domain is activated, it is executed in parallel within *all* instances of the domain, not just a single designated instance. Compound assignment operators such as `+ =` are extended in C* to perform parallel reduction operations. An elaborate theory of control flow allows use of C control statements in a MIMD-like, yet predictable, fashion despite the fact that the underlying execution model is SIMD. The revised version (1991) replaces domains with *shapes* that organize processors into multidimensional arrays and abandons the MIMD-like control-flow theory.

Discussion

Of the four programming languages (*Lisp, C*, CM Fortran, and CM-Lisp) provided by Thinking Machines Corporation for Connection Machine Systems, C* was the most clever (indeed, perhaps too clever) in trying to extend features of an already existing sequential language for parallel execution. To quote the language designers:

- ▶ C* is an extension of the C programming language designed to support programming in the data parallel style, in which the programmer writes code as if a processor were associated with every data element. C* features a single new data type (based on classes in C++), a synchronous execution model, and a minimal number of extensions to C statement and expression

syntax. Rather than introducing a plethora of new language constructs to express parallelism, C* relies on existing C operators, applied to parallel data, to express such notions as broadcasting, reduction, and interprocessor communication in both regular and irregular patterns [5, 6, 8].

The original proposed name for the language was *C, not only by analogy with *Lisp, but with a view for the potential of making a similarly data-parallel extension to the C++ language, which would then naturally be called *C++. However, the marketing department of Thinking Machines Corporation decided that “C*” sounded better. This inconsistency in the placement of the “*” did confuse many Thinking Machines customers and others, resulting in frequent references to “*C” anyway, and even to “Lisp*” on occasion.

The Initial Design of C*

The basic idea was to start with the C programming language and then augment it with the ability to declare something like a C++ class, but with the keyword `class` replaced with the keyword `domain`. As in C++, a domain could have functions as well as variables as members. However, the notion of method invocation (calling a member function on a single specific instance) was replaced by the notion of domain activation (calling a member function, or executing code, on *all instances* simultaneously and synchronously). Everything else in the language was driven by that one design decision, that one metaphor.

Two new keywords, `mono` and `poly`, were introduced to describe in which memory data resided – the front-end computer or the Connection Machine processors, respectively. Variables declared within sequential code were `mono` by default, and variables declared within parallel code were `poly` by default, so the principal use of these keywords was in describing pointer types; for example, the declaration

```
mono int *poly p;
```

indicates that p is a poly pointer to a mono int (i.e., p holds many values, all stored in the Connection Machine processors, one in each instance of the current domain; and each of these values is a pointer to an integer that resides in the front-end processor, but each of these pointers might point to a different front-end integer).

C^* systematically extended the standard operators in C to have parallel semantics by applying two rules: (1) if a binary operator has a scalar operand and a parallel operand, the scalar value is automatically replicated to form a parallel value (an idea previously seen in both APL and Fortran 8x), and (2) an operator applied to parallel operands is executed for all active processors *as if* in some serial order. In this way, binary operators such as $+$ and $-$ and $\%$ can be applied elementwise to many sets of operands at once, and binary operators with side effects – the compound assignment operators – are guaranteed to have predictable sensible behavior; e.g., if a is a scalar variable and b is a parallel variable, then the effect of $a += b$ is to add every active element of b into a , because it must behave as if each active element of b were added into a in some sequential order. (In practice, the implementation used a parallel algorithm to sum the elements of b and then added that sum into a .)

C^* makes two extensions to the standard set of C operators, both motivated by a desire to extend the parallel functionality provided by the standard operators in a consistent manner. Two common arithmetic operations, \min and \max , are provided in C as preprocessor macros rather than as operators; one disadvantage of a macro is that it cannot be used as part of a compound assignment operator, and so one cannot write $a \max= b$ in the same way that one can write $a += b$. C^* introduces operators $<?$ and $>?$ to serve as \min and \max operations. The designers commented:

- ▶ They may be understood in terms of their traditional macro definitions

```
a <? b means ((a) < (b)) ? (a) : (b)
a >? b means ((a) > (b)) ? (a) : (b)
```

but of course the operators, unlike the macro definitions, evaluate each argument exactly once. The operators $<?$ and $>?$ are intended as mnemonic reminders of these definitions. (Such mnemonic reminders are important. The original design of C^* used $><$ and $<>$

for the maximum and minimum operators. We quickly discovered that users had some difficulty remembering which was which.) [5, 6]

In addition, most of the binary compound operators are pressed into service as unary reduction operators. Thus $+ = b$ computes the sum of all the active values in b and returns that sum as a mono value; similarly $>=? b$ finds the largest active value in b .

C^* also added a modulus operator $\% \%$ because of the great utility of modulus, as distinguished from the remainder operator $\%$, in performing array index calculations: when k is zero, the expression $(k - 1) \% n$ produces a much more useful result, namely, $k - 1$, than does the expression $(k - 1) \% n$, which produces -1 .

Because a processor can contain pointers to variables residing in other processors, interprocessor communication can be expressed simply by dereferencing such pointers. Thus if p is a poly pointer to a poly int, then $*p$ causes each active processor to fetch an int value through its own p pointer, and $*p = b$ causes each active processor to store its b value indirectly through p (thus “sending a message” to some other processor). The “combining router” feature of the Connection Machine could be invoked by an expression such as $*p += b$, which might cause the b values to be sent to some smaller number of destinations, resulting in the summing (in parallel) of various subsets of the b values into the various individual destinations.

C^* has an elaborate theory of implicitly synchronized control flow that allows the programmer to code, for the most part, as if each Connection Machine processor were executing its own copy of parallel code independently as if it were ordinary sequential C code. The idea is that each processor has its own program counter (a “virtual PC”), but can make no progress until a “master PC” arrives at that point in the code, at which point that virtual PC (as well every other virtual PC waiting at that particular code location) becomes active, joining the master PC and participating in SIMD execution. Whenever the master PC reaches a conditional branch, each active virtual PC that takes the branch becomes inactive (thus, e.g., in an `if` statement, after evaluation of the test expression, processors that need to execute the `else` part take an implicit branch to the `else` part and wait while other processors possibly

proceed to execute the “then” part). Whenever the master PC reaches the end of a statement, every virtual PC becomes inactive, and the master PC is transferred from the current point to a new point in the code, namely, the *earliest* point that has waiting program counters, within the *innermost* statement that has waiting program counters within it, that contains the current point. Frequently this new point is the same as the current point, and frequently this fact can be determined at compile time; but in general the effect is to keep trying to pull lagging program counters forward, in such a way that once the master PC enters a block, it does not leave the block until every virtual PC has left the block. Thus this execution rule respects block structure (and subroutine call structure) in a natural manner, while allowing the programmer to make arbitrary use of `goto` statements if desired. (This theory of control flow was inspired by earlier investigations of a SIMD-like theory giving rise to a superficially MIMD-like control structure in Connection Machine Lisp [7].)

A compiler for this version of C* was independently implemented at the University of New Hampshire [4] for a MIMD multicomputer (an N-Cube 3200, whose processors communicated by message passing through a hypercube network).

Figure 1 shows an early (1987) example of a C* program that identifies prime numbers by the method of the Sieve of Eratosthenes, taken (with one minor correction) from [10]. In typical C style, `N` is defined to be a preprocessor name that expands to the integer literal `100000`. The name `bit` is then defined to be a synonym for the type of 1-bit integers. The `domain` declaration defines a parallel processing domain named `SIEVE` that has a field named `prime` of type `bit`, and then declares an array named `sieve` of length `N`, each element of which is an instance of this domain. (This single statement could instead have been written as two statements:

```
domain SIEVE { bit prime; };
domain SIEVE sieve[N];
```

In this respect `domain` declarations are very much like C `struct` declarations.) The function `find_primes` is declared within the domain `SIEVE`, so when it is called, its body is executed within every instance of that domain, with a distinct virtual processor of the Connection Machine containing and processing

each such instance. The “magic identifier” `this` is a pointer to the current domain instance, and its value is therefore different within each executing instance; subtracting from it a pointer to the first instance in the array, namely, `&sieve[0]`, produces the index of that instance within the `sieve` array, by the usual process of C pointer arithmetic. Local variables `value` and `candidate` are declared within each instance of the domain, and therefore are parallel values. The `while` statement behaves in such a way that different domain instances can execute different number of iterations, as appropriate to the data within that instance; when an active instance computes a zero (false) value for the test expression `candidate`, that instance simply becomes inactive. When all instances have become inactive, then the `while` statement completes after making active exactly those instances that had been active when execution of the `while` statement began. The `mono` storage class keyword indicates that a variable should be allocated just once (in the front-end processor), not once within each domain instance. The unary operator `<?=` is the minimum-reduction operator, so the expression `(<?= value)` returns the smallest integer that any active domain instance holds in its `value` variable. The array `sieve` can be indexed in the normal C fashion, by either a `mono` (front-end) index, as in this example code, or by a `poly` (parallel) value, which can be used to perform inter-domain (i.e., inter-processor) communication. The `if` statement is handled in much the same way as the `while` statement: If an active instance computes a zero (false) value for the test expression `candidate`, that instance simply becomes inactive during execution of the “then” part of the `if` statement, and then becomes active again. (If an `if` statement has an `else` part, then processors that had been active for the “then” part become temporarily inactive during execution of the `else` part.).

The Revised Design of C*

In 1991, the C* language was revised substantially [1] to produce version 6.0; this version was initially implemented for the Connection Machine model CM-2 [9, 10] and later for the model CM-200 as well as the model CM-5 [11, 56]. This revised design was intended to be closer in style to ANSI C than to C++. The biggest change was to introduce the notion of a *shape*,

```

#define N 100000

typedef int bit:1;

domain SIEVE { bit prime; } sieve[N];

void SIEVE::find_primes() {
    int value = this - &sieve[0];
    bit candidate = (value >= 2);
    prime = 0;
    while (candidate) {
        mono int next_prime = (<?= value);
        sieve[next_prime].prime = 1;
        if (value % next_prime == 0) candidate = 0;
    }
}

```

C*. Fig. 1 Example version 5 C* program for identifying prime numbers

which essentially describes a (possibly multidimensional) array of virtual processors. An ordinary variable declaration may be tagged with the name of a shape, which indicates that the declaration is replicated for parallel processing, one instance for each position in the shape. Where the initial design of C* required that *all* instances of a domain be processed in parallel, the new design allows declaration of several different shapes (parallel arrays) having the same data layout, and different shapes may be chosen at different times for parallel execution. Parallel execution is initiated using a newly introduced *with* statement that specifies a shape:

with (shape) statement

The *statement* is executed with the specified *shape* as the “current shape” for parallel execution. C operators may be applied to parallel data much as in the original version of C*, but such data must have the current shape (a requirement enforced by the compiler at compile time).

Positions within shapes may be selected by indexing. In order to distinguish shape indexing from ordinary array indexing, shape subscripts are written to the left rather than to the right. Given these declarations:

```

shape [16] [16] [16] cube;
float:cube z;
int:cube a[10] [10];

```

then *cube* is a three-dimensional shape having 4,096 ($16 \times 16 \times 16$) distinct positions, *z* is a parallel variable consisting of one *float* value at each of these 4,096 shape positions, and *a* is a parallel array variable that has a 10×10 array of *int* values at each of 4,096 shape positions (for a total of 4,09,600 distinct *int* values). Then [3] [13] [5] *z* refers to one particular *float* value within *z* at position (3,13,5) within the *cube* shape, and [3] [13] [5] *a* [4] [7] refers to one particular *int* element within the particular 10×10 array at that same position. One may also write [3] [13] [5] *a* to refer to (the address of) that same 10×10 array. (This use of left subscripts to index “across processors” and right subscripts to index “within a processor” may be compared to the use of square brackets and parentheses to distinguish two sorts of subscript in Co-Array Fortran [3].)

Although pointers can be used for interprocessor communication exactly as in earlier versions of C*, such communication is more conveniently expressed in C* version 6.0 by shape indexing (writing [i] [j] [k] *z* = *b* rather than **p* = *b*, for example), thus affording a more array-oriented style of programming. Furthermore, version 6.0 of C* abandons the entire “master program counter” execution model that allowed all C control structures to be used for parallel execution. Instead, statements such as *if*, *while*, and *switch* are restricted to test nonparallel values, and a

newly introduced `where` statement tests parallel values, behaving very much like the `where` statement in Fortran 90. The net effect of all these revisions is to give version 6.0 of C* an execution model and programming style more closely resembling those of *Lisp and CM Fortran.

Figure 2 shows a rewriting of the code in Fig. 1 into C* version 6.0. The domain declaration is replaced by a `shape` declaration that specifies a one-dimensional shape named `SIEVE` of size `N`. The global variable `prime` of type `bit` is declared within shape `SIEVE`. The function `find_primes` is no longer declared within a domain. When it is called, the `with` statement establishes `SIEVE` as the current shape for parallel execution; conceptually, a distinct virtual processor of the Connection Machine is associated with each position in the current shape. The function `pcoord` takes an integer specifying an axis number and returns, at each position of the current shape, an integer indicating the index of the position along the specified axis; thus in this example `pcoord` returns values ranging from 0 to 99999. Local variables `value` and `candidate` are explicitly declared as belonging to shape `SIEVE`,

and therefore are parallel values. The `while` statement of Fig. 1 becomes a `while` statement and a `where` statement in Fig. 2; the new `while` statement uses an explicit OR-reduction operator `|=` to decide whether there are any remaining candidates; if there are, the `where` statement makes inactive every position in the shape for which its `candidate` value is zero. The declaration of local variable `next_prime` does not mention a `shape`, so it is not a parallel variable (note that the `mono` keyword is no longer used). A left subscript is used to assign 1 to a single value of `prime` at the position within shape `SIEVE` indicated by the value of `next_prime`.

The need to split what was a single `while` statement in earlier versions of C* into two nested statements (a `while` with an OR-reduction operator containing a `where` that typically repeats the same expression) has been criticized, as well as the syntax of type and shape declarations and the lack of nested parallelism [14].

Version 7.2 of C* [13] introduced a “global/local programming” feature, allowing C* to be used for MIMD programming on the model CM-5. A special “prototype file” is used to specify which functions are local

```
#define N 100000

typedef int bit:1;

shape [N]SIEVE;
bit:SIEVE prime;

void find_primes() {
    with (SIEVE) {
        int:SIEVE value = pcoord(0);
        bit:SIEVE candidate = (value >= 2);
        prime = 0;
        while (|= candidate) {
            where (candidate) {
                int next_prime = (<?= value);
                [next_prime]prime = 1;
                if (value % next_prime == 0) candidate = 0;
            }
        }
    }
}
```

C*. Fig. 2 Example version 6 C* program for identifying prime numbers

and the calling interface to be used when global code calls each local function. The idea is that a local function executes on a single processing node but can use all the facilities of C*, including parallelism (which might be implemented on the model CM-5 through SIMD vector accelerator units). This facility is very similar to “local subprograms” in High Performance Fortran [2].

Related Entries

- ▶ [Coarray Fortran](#)
- ▶ [Connection Machine](#)
- ▶ [Connection Machine Fortran](#)
- ▶ [Connection Machine Lisp](#)
- ▶ [HPF \(High Performance Fortran\)](#)
- ▶ [*Lisp](#)

Bibliography

1. Frankel JL (1991) A reference description of the C* language. Technical Report TR-253, Thinking Machines Corporation, Cambridge, MA
2. Koelbel CH, Loveman DB, Schreiber RS, Steele GL Jr, Zosel ME (1994) The High Performance Fortran handbook. MIT Press, Cambridge, MA
3. Numrich RW, Reid J (1998) Co-array Fortran for parallel programming. SIGPLAN Fortran Forum 17(2):1–31
4. Quinn MJ, Hatcher PJ (1990) Data-parallel programming on multicomputers. IEEE Softw 7(5):69–76
5. Rose JR, Steele GL Jr (1986) C*: An extended C language for data parallel programming. Technical Report PL 87-5, Thinking Machines Corporation, Cambridge, MA
6. Rose JR, Steele GL Jr (1987) C*: An extended C language for data parallel programming. In: Supercomputing '87: Proceedings of the second international conference on supercomputing, vol II: Industrial supercomputer applications and computations. International Supercomputing Institute, Inc., St. Petersburg, Florida, pp 2–16
7. Steele GL Jr, Daniel Hillis W (1986) Connection Machine Lisp: fine-grained parallel symbolic processing. In: LFP '86: Proc. 1986 ACM conference on LISP and functional programming, ACM SIGPLAN/SIGACT/SIGART, ACM, New York, pp 279–297, Aug 1986
8. Thinking Machines Corporation (1987) Connection Machine model CM-2 technical summary. Technical report HA87-4, Cambridge, MA
9. Thinking Machines Corporation (1990) C* programming guide, version 6.0 Pre-Beta. Cambridge, MA
10. Thinking Machines Corporation (1990) C* user's guide, version 6.0 Pre-Beta. Cambridge, MA

11. Thinking Machines Corporation (1993) C* programming guide. Cambridge, MA
12. Thinking Machines Corporation (1993) Connection Machine CM-5 technical summary, 3rd edn. Cambridge, MA
13. Thinking Machines Corporation (1994) C* 7.2 Alpha release notes. Cambridge, MA
14. Tichy WF, Philippson M, Hatcher P (1992) A critique of the programming language C*. Commun ACM 35(6):21–24

Cache Affinity Scheduling

- ▶ [Affinity Scheduling](#)

Cache Coherence

XIAOWEI SHEN
IBM Research, Armonk, NY, USA

Definition

A shared-memory multiprocessor system provides a global address space in which processors can exchange information and synchronize with one another. When shared variables are cached in multiple caches simultaneously, a memory store operation performed by one processor can make data copies of the same variable in other caches out of date. Cache coherence ensures a coherent memory image for the system so that each processor can observe the semantic effect of memory access operations performed by other processors in time.

Discussion

The cache coherence mechanism plays a crucial role in the construction of a shared-memory system, because of its profound impact on the overall performance and implementation complexity. It is also one of the most complicated problems in the design, because an efficient cache coherence protocol usually incorporates various optimizations.

Cache Coherence and Memory Consistency

The cache coherence protocol of a shared-memory multiprocessor system implements a memory consistency model that defines the semantics of memory access instructions. The essence of memory consistency is the

correspondence between each load instruction and the store instruction that supplies the data retrieved by the load instruction. The memory consistency model of uniprocessor systems is intuitive: a load operation returns the most recent value written to the address, and a store operation binds the value for subsequent load operations. In parallel systems, however, notions such as “the most recent value” can become ambiguous since multiple processors access memory concurrently.

An ideal memory consistency model should allow efficient implementations while still maintaining simple semantics for the architect and the compiler writer to reason about. Sequential consistency [1] is a dominant memory model in parallel computing for decades due to its simplicity. A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Sequential consistency is easy for programmers to understand and use, but it often prohibits many architectural and compiler optimizations. The desire to achieve higher performance has led to relaxed memory models, which can provide more implementation flexibility by exposing optimizing features such as instruction reordering and data caching. Modern microprocessors [2, 3] support selected relaxed memory consistency models that allow memory accesses to be reordered, and provide memory fences that can be used to ensure proper memory access ordering constraints whenever necessary.

It is worth noting that, as a reaction to ever-changing memory models and their complicated and imprecise definitions, there is a desire to go back to the simple, easy-to-understand sequential consistency, even though there are a plethora of problems in its high-performance implementation [4]. Ingenious solutions have been devised to maintain the sequential consistency semantics so that programmers cannot detect if and when the memory accesses are out of order or nonatomic. For example, advances in speculative execution may permit memory access reordering without affecting the semantics of sequential consistency.

Sometimes people may get confused between memory consistency models and cache coherence protocols. A memory consistency model defines the semantics of memory operations, in particular, for each memory load operation, the data value that should be provided by the memory system. The memory consistency model is a critical part of the semantics of the Instruction-Set Architecture of the system, and thus should be exposed to the system programmer. A cache coherence protocol, in contrast, is an implementation-level protocol that defines how caches should be kept coherent in a multiprocessor system in which data of a memory address can be replicated in multiple caches, and thus should be made transparent to the system programmer. Generally speaking, in a shared-memory multiprocessor system, the underlying cache coherence protocol, together with some proper memory operation reordering constraint often enforced when memory operations are issued, implements the semantics of the memory consistency model defined for the system.

Snoopy Cache Coherence

A symmetric multiprocessor (SMP) system generally employs a snoopy mechanism to ensure cache coherence. When a processor reads an address not in its cache, it broadcasts a read request on the bus or network, and the memory or the cache with the most up-to-date copy can then supply the data. When a processor broadcasts its intention to write an address which it does not own exclusively, other caches need to invalidate or update their copies.

With snoopy cache coherence, when a cache miss occurs, the requesting cache sends a cache request to the memory and all its peer caches. When a peer cache receives the cache request, it performs a cache snoop operation and produces a cache snoop response indicating whether the requested data is found in the peer cache and the state of the corresponding cache line. A combined snoop response can be generated based on cache snoop responses from all the peer caches. If the requested data is found in a peer cache, the peer cache can source the data to the requesting cache via a cache-to-cache transfer, which is usually referred to as a cache intervention. The memory is responsible for supplying the requested data if the combined snoop

response shows that the data cannot be supplied by any peer cache.

Example: The MESI Cache Coherence Protocol

A number of snoopy cache coherence protocols have been proposed. The MESI coherence protocol and its variations have been widely used in SMP systems. As the name suggests, MESI has four cache states, modified (M), exclusive (E), shared (S), and invalid (I).

- I (invalid): The data is not valid. This is the initial state or the state after a snoop invalidate hit.
- S (shared): The data is valid, and can also be valid in other caches. This state is entered when the data is sourced from the memory or another cache in the modified state, and the corresponding snoop response shows that the data is valid in at least one of the other caches.
- E (exclusive): The data is valid, and has not been modified. The data is exclusively owned, and cannot be valid in another cache. This state is entered when the data is sourced from the memory or another cache in the modified state, and the corresponding snoop response shows that the data is not valid in another cache.
- M (modified): The data is valid and has been modified. The data is exclusively owned, and cannot be valid in another cache. This state is entered when a store operation is performed on the cache line.

With the MESI protocol, when a read cache miss occurs, if the requested data is found in another cache and the cache line is in the modified state, the cache with the modified data supplies the data via a cache intervention (and writes the most up-to-date data back to the memory). However, if the requested data is found in another cache and the cache line is in the shared state, the cache with the shared data does not supply the requested data, since it cannot guarantee from the shared state that it is the only cache that is to source the data. In this case, the memory will supply the data to the requesting cache. When a write cache miss occurs, if data of the memory address is cached in one or more other caches in the shared state, all those cached copies in other caches are invalidated before the write operation can be performed in the local cache.

It should be pointed out that the MESI protocol described above is just an exemplary protocol to show the essence of cache coherence operations. It can be modified or tailored in various ways for implementation optimization. For example, one can imagine that a cache line in a shared state can provide data for a read cache miss, rather than letting the memory provide the data. This may provide better response time for a system in which cache-to-cache data transfer is faster than memory-to-cache data transfer. Since there can be more than one cache with the requested data in the shared state, the cache coherence protocol needs to specify which cache in the shared state should provide the data, or how extra data copies should be handled in case multiple caches provide the requested data at the same time.

Example: An Enhanced MESI Cache Coherence Protocol

In modern SMP systems, when a cache miss occurs, if the requested data is found in both the memory and a cache, supplying the data via a cache intervention is often preferred over supplying the data from the memory, because cache-to-cache transfer latency is usually smaller than memory access latency. Furthermore, when caches are on the same die or in the same package module, there is usually more bandwidth available for cache-to-cache transfers, compared with the bandwidth available for off-chip DRAM accesses.

The IBM Power-4 system [5], for example, enhances the MESI coherence protocol to allow more cache interventions. Compared with MESI, an enhanced coherence protocol allows data of a shared cache line to be sourced via a cache intervention. In addition, if data of a modified cache line is sourced from one cache to another, the modified data does not have to be written back to the memory immediately. Instead, a cache with the most up-to-date data can be held responsible for memory update when it becomes necessary to do so. An exemplary enhanced MESI protocol employing seven cache states is as follows.

- I (invalid): The data is invalid. This is the initial state or the state after a snoop invalidate hit.
- SL (shared, can be sourced): The data is valid and may also be valid in other caches. The data can

be sourced to another cache in the same module via a cache intervention. This state is entered when the data is sourced from another cache or from the memory.

- S (shared): The data is valid, and may also be valid in other caches. The data cannot be sourced to another cache. This state is entered when a snoop read hit from another cache in the same module occurs on a cache line in the SL state.
- M (modified): The data is valid, and has been modified. The data is exclusively owned, and cannot be valid in another cache. The data can be sourced to another cache. This state is entered when a store operation is performed on the cache line.
- ME (exclusive): The data is valid, and has not been modified. The data is exclusively owned, and cannot be valid in another cache.
- MU (unsolicited modified): The data is valid and is considered to have been modified. The data is exclusively owned and cannot be valid in another cache.
- T (tagged): The data is valid and has been modified. The modified data has been sourced to another cache. This state is entered when a snoop read hit occurs on a cache line in the M state.

When data of a memory address is shared in multiple caches in a single module, the single module can include at most one cache in the SL state. The cache in the SL state is responsible for supplying the shared data via a cache intervention when a cache miss occurs in another cache in the same module. At any time, the particular cache that can source the shared data is fixed, regardless of which cache has issued the cache request. When data of a memory address is shared in more than one module, each module can include a cache in the SL state. A cache in the SL state can source the data to another cache in the same module, but cannot source the data to a cache in a different module.

In systems in which a cache-to-cache transfer can take multiple message-passing hops, sourcing data from different caches can result in different communication latency and bandwidth consumption. When a cache miss occurs in a requesting cache, if requested data is shared in more than one peer cache, a peer cache that is closest to the requesting cache is preferred to supply the

requested data to reduce communication latency and bandwidth consumption of cache intervention. Thus, it is probably desirable to enhance cache coherence mechanisms with cost-conscious cache-to-cache transfers to improve overall performance in SMP systems.

Broadcast-Based Cache Coherence Versus Directory-Based Cache Coherence

A major drawback of broadcast-based snoopy cache coherence protocols is that a cache request is usually broadcast to all caches in the system. This can cause serious problems to overall performance, system scalability, and power consumption, especially for large-scale multiprocessor systems. Further, broadcasting cache requests indiscriminately may consume enormous network bandwidth, while snooping peer caches unnecessarily may require excessive cache snoop ports. It is worth noting that servicing a cache request may take more time than necessary when far away caches are snooped unnecessarily.

Unlike broadcast-based snoopy cache coherence protocols, a directory-based cache coherence protocol maintains a directory entry to record the cache sites in which each memory block is currently cached [6]. The directory entry is often maintained at the site in which the corresponding physical memory resides. Since the locations of shared copies are known, the protocol engine at each site can maintain coherence by employing point-to-point protocol messages. The elimination of broadcast overcomes a major limitation on scaling cache coherent machines to large-scale multiprocessor systems.

Typical directory-based protocols maintain a directory entry for each memory block to record the caches in which the memory block is currently cached. With a full-map directory structure, for example, each directory entry comprises one bit for each cache in the system, indicating whether the cache has a data copy of the memory block. Given a memory address, its directory entry is usually maintained in a node in which the corresponding physical memory resides. This node is often referred to as the home of the memory address. When a cache miss occurs, the requesting cache sends a cache request to the home, which generates appropriate point-to-point coherence messages according to the directory information.

However, directory-based cache coherence protocols have various shortcomings. For example, maintaining a directory entry for each memory block usually results in significant storage overhead. Alternative directory structures may reduce the storage overhead with performance compromises. Furthermore, accessing directory can be time consuming since directory information is usually stored in DRAM. Caching recently used directory entries can potentially reduce directory access latencies but with increased implementation complexity.

Accessing directory causes three or four message-passing hops to service a cache request, compared with two message-passing hops with snoopy cache coherence protocols. Consider a scenario in which a cache miss occurs in a requesting cache, while the requested data is modified in another cache. To service the cache miss, the requesting cache sends a cache request to the corresponding home. When the home receives the cache request, it forwards the cache request to the cache that contains the modified data. When the cache with the modified data receives the forwarded cache request, it sends the requested data to the requesting cache (an alternative is to send the requested data to the home, which will forward the requested data to the requesting cache).

Cache Coherence for Network-Based Multiprocessor Systems

In a modern shared-memory multiprocessor system, caches can be interconnected with each other via a message-passing network instead of a shared bus to improve system scalability and performance. In a bus-based SMP system, the bus behaves as a central arbitrator that serializes all bus transactions. This ensures a total order of bus transactions. In a network-based multiprocessor system, in contrast, when a cache broadcasts a message, the message is not necessarily observed atomically by all the receiving caches. For example, it is possible that cache A multicasts a message to caches B and C, cache B receives the broadcast message and then sends a message to cache C, and cache C receives cache B's message before receiving cache A's multicast message.

Protocol correctness can be compromised when multicast messages can be received in different orders

at different receiving caches. Appropriate mechanisms are needed to guarantee correctness of cache coherence for network-based multiprocessor systems [7, 8].

Bibliography

1. Lamport L (1979) How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans Comput* C-28(9):690–691
2. May C, Silha E, Simpson R, Warren H (1994) The powerPC architecture: a specification for a new family of RISC processors. Morgan Kaufmann, San Francisco
3. Intel Corporation (1999) IA-64 application developer's architecture guide
4. Gniady C, Falsafi B, Vijaykumar T (1999) Is SC+ILP=RC? In: Proceedings of the 26th annual international symposium on computer architecture (ISCA 1999), Atlanta, 2–4 May 1999, pp 162–17
5. Tendler J, Dodson J, Fields J, Le H, Sinharoy B (2002) POWER-4 system microarchitecture. *IBM J Res Dev* 46(1):5
6. Chaiken D, Fields C, Kurihara K, Agarwal A (1990) Directory-based cache coherence in large-scale multiprocessors. *Computer* 23(6):49–58
7. Martin M, Hill M, Wood D (2003) Token coherence: decoupling performance and corrections. In: Proceedings of the 30th annual international symposium on computer architecture international symposium on computer architecture, San Diego, 9–11 June 2003
8. Strauss K, Shen X, Torrellas J (2007) Uncorq: unconstrained snoop request delivery in embedded-ring multiprocessors. In: Proceedings of the 40th annual IEEE/ACM international symposium on microarchitecture, Chicago, pp 327–342, 1–5 Dec 2007

Cache-Only Memory Architecture (COMA)

JOSEP TORRELLAS

University of Illinois at Urbana-Champaign, Urbana, IL, USA

Synonyms

[COMA \(Cache-only memory architecture\)](#)

Definition

A Cache-Only Memory Architecture (COMA) is a type of cache-coherent nonuniform memory access (CC-NUMA) architecture. Unlike in a conventional CC-NUMA architecture, in a COMA, every shared-memory module in the machine is a cache, where each memory line has a tag with the line's address and state.

As a processor references a line, it transparently brings it to both its private cache(s) and its nearby portion of the NUMA shared memory (*Local Memory*) – possibly displacing a valid line from its local memory. Effectively, each shared-memory module acts as a huge cache memory, giving the name COMA to the architecture. Since the COMA hardware automatically replicates the data and migrates it to the memory module of the node that is currently accessing it, COMA increases the chances of data being available locally. This reduces the possibility of frequent long-latency memory accesses. Effectively, COMA dynamically adapts the shared data layout to the application's reference patterns.

Discussion

Basic Concepts

In a conventional CC-NUMA architecture, each node contains one or more processors with private caches and a memory module that is part of the NUMA shared memory. A page allocated in the memory module of one node can be accessed by the processors of all other nodes. The physical page number of the page specifies the node where the page is allocated. Such node is referred to as the *Home Node* of the page. The physical address of a memory line includes the physical page number and the offset within that page.

In large machines, fetching a line from a remote memory module can take several times longer than fetching it from the local memory module. Consequently, for an application to attain high performance, the local memory module must satisfy a large fraction of the cache misses. This requires a good placement of the program pages across the different nodes. If the program's memory access patterns are too complicated for the software to understand, individual data structures may not end up being placed in the memory module of the node that access them the most. In addition, when a page contains data structures that are read and written by different processors, it is hard to attain a good page placement.

In a COMA, the hardware can transparently eliminate a certain class of remote memory accesses. COMA does this by turning memory modules into large caches called *Attraction Memory* (AM). When a processor requests a line from a remote memory, the line is

inserted in both the processor's cache and the node's AM. A line can be evicted from an AM if another line needs the space. Ideally, with this support, the processor dynamically attracts its working set into its local memory module. The lines the processor is not accessing overflow and are sent to other memories. Because a large AM is more capable of containing a node's current working set than a cache is, more of the cache misses are satisfied locally within the node.

There are three issues that need to be addressed in COMA, namely finding a line, replacing a line, and dealing with the memory overhead. In the rest of this article, these issues are described first, then different COMA designs are outlined, and finally further readings are suggested.

Finding a Memory Line

In a COMA, the address of a memory line is a global identifier, not an indicator of the line's physical location in memory. Just like a normal cache, the AM keeps a tag with the address and state of the memory line currently stored in each memory location. On a cache miss, the memory controller has to look up the tags in the local AM to determine whether or not the access can be serviced locally. If the line is not in the local AM, a remote request is issued to locate the block.

COMA machines have a mechanism to locate a line in the system so that the processor can find a valid copy of the line when a miss occurs in the local AM. Different mechanisms are used by different classes of COMA machines.

One approach is to organize the machine hierarchically, with the processors at the leaves of the tree. Each level in the hierarchy includes a directory-like structure, with information about the status of the lines present in the subtree extending from the leaves up to that level of the hierarchy. To find a line, the processing node issues a request that goes to successively higher levels of the tree, potentially going all the way to the root. The process stops at the level where the subtree contains the line. This design is called Hierarchical COMA [2, 7].

Another approach involves assigning a home node to each memory line, based on the line's physical address. The line's home has the directory entry for the line. Memory lines can freely migrate, but directory entries do not. Consequently, to locate a memory line,

a processor interrogates the directory in the line's home node. The directory always knows the state and location of the line and can forward the request to the right node. This design is called Flat COMA [12].

Replacing a Memory Line

The AM acts as a cache, and lines can be displaced from it. When a line is displaced in a plain cache, it is either overwritten (if it is unmodified) or written back to its home memory module, which guarantees a place for the line.

A memory line in COMA does not have a fixed backup location where it can be written to if it gets displaced from an AM. Moreover, even an unmodified line can be the only copy of that memory line in the system, and it must not be lost on an AM displacement. Therefore, the system must keep track of the last copy of a line. As a result, when a modified or otherwise unique line is displaced from an AM, it must be relocated into another AM.

To guarantee that at least one copy of an unmodified line remains in the system, one of the line's copies is denoted as the *Master* copy. All other shared copies can be overwritten if displaced, but the master copy must always be relocated to another AM. When a master copy or a modified line is relocated, the problem is deciding which node should take the line in its AM. If other nodes already have one or more other shared copies of the line, one of them becomes the master copy. Otherwise, another node must accept the line. This process is called *Line Injection*.

Different line injection algorithms are possible. One approach is for the displacing node to send requests to other nodes asking if they have space to host the line [7]. Another approach is to force one node to accept the line. This, however, may lead to another line displacement. A proposed solution is to relocate the new line to the node that supplied the line that caused the displacement in the first place [8].

Dealing with Memory Overhead

A CC-NUMA machine can allocate all memory to application or system pages. COMA, however, leaves a portion of the memory unallocated to facilitate automatic data replication and migration. This unallocated space supports the replication of lines across AMs.

It also enhances line migration to the AMs of the referencing nodes because less line relocation traffic is needed.

Without unallocated space, every time a line is inserted in the AM, another line would have to be relocated. The ratio between the allocated data size and the total size of the AMs is called the *Memory Pressure*. If the memory pressure is 80%, then 20% of the AM space is available for data replication. Both the relocation traffic and the number of AM misses increase with the memory pressure [8]. For a given memory size, choosing an appropriate memory pressure is a trade-off between the effect on page faults, AM misses, and relocation traffic.

Different Cache-Only Memory Architecture Designs

Hierarchical COMA

The first designs of COMA machines follow what has been called Hierarchical COMA. These designs organize the machine hierarchically, connecting the processors to the leaves of the tree. These machines include the KSR-1 [2] from Kendall Square Research, which has a hierarchy of rings, and the Data Diffusion Machine (DDM) [7] from the Swedish Institute of Computer Science, which has a hierarchy of buses.

Each level in the tree hierarchy includes a directory-like structure, with information about the status of the lines extending from the leaves up to that level of the hierarchy. To find a line, the processing node issues a request that goes to successively higher levels of the tree, potentially going all the way to the root. The process stops at the level where the subtree contains the line.

In these designs, substantial latency occurs as the memory requests go up the hierarchy and then down to find the desired line. It has been argued that such latency can offset the potential gains of COMA relative to conventional CC-NUMA architectures [12].

Flat COMA

A design called Flat COMA makes it easy to locate a memory line by assigning a home node to each memory line [12] – based on the line's physical address. The line's home has the directory entry for the line, like in a conventional CC-NUMA architecture. The memory lines can freely migrate, but the directory entries of the memory lines are fixed in their home nodes. At a miss on a

line in an AM, a request goes to the node that is keeping the directory information about the line. The directory redirects the request to another node if the home does not have a copy of the line. In Flat COMA, unlike in a conventional CC-NUMA architecture, the home node may not have a copy of the line even though no processor has written to the line. The line has simply been displaced from the AM in the home node.

Because Flat COMA does not rely on a hierarchy to find a block, it can use any high-speed network.

Simple COMA

A design called Simple COMA (S-COMA) [10] transfers some of the complexity in the AM line displacement and relocation mechanisms to software. The general coherence actions, however, are still maintained in hardware for performance reasons. Specifically, in S-COMA, the operating system sets aside space in the AM for incoming memory blocks on a page- granularity basis. The local Memory Management Unit (MMU) has mappings only for pages in the local node, not for remote pages. When a node accesses for the first time a shared page that is already in a remote node, the processor suffers a page fault. The operating system then allocates a page frame locally for the requested line. Thereafter, the hardware continues with the request, including locating a valid copy of the line and inserting it, in the correct state, in the newly allocated page in the local AM. The rest of the page remains unused until future requests to other lines of the page start filling it. Subsequent accesses to the line get their mapping directly from the MMU. There are no AM address tags to check if the correct line is accessed.

Since the physical address used to identify a line in the AM is set up independently by the MMU in each node, two copies of the same line in different nodes are likely to have different physical addresses. Shared data needs a global identity so that different nodes can communicate. To this end, each node has a translation table that converts local addresses to global identifiers and vice versa.

Multiplexed Simple COMA

S-COMA sets aside memory space in page-sized chunks, even if only one line of each page is present.

Consequently, S-COMA suffers from memory fragmentation. This can cause programs to have inflated working sets that overflow the AM, inducing frequent page replacements and resulting in high operating system overhead and poor performance.

Multiplexed Simple COMA (MS-COMA) [1] eliminates this problem by allowing multiple virtual pages in a given node to map to the same physical page at the same time. This mapping is possible because all the lines on a virtual page are not used at the same time. A given physical page can now contain lines belonging to different virtual pages if each line has a short virtual page ID. If two lines belonging to different pages have the same page offset, they displace each other from the AM. The overall result is a compression of the application's working set.

Further Readings

There are several papers that discuss COMA and related topics. Dahlgren and Torrellas present a more in-depth survey of COMA machine issues [3]. There are several designs that combine COMA and conventional CC-NUMA architecture features, such as NUMA with Remote Caches (NUMA-RC) [9], Reactive NUMA [5], Excel-NUMA [14], the Sun Microsystems' WildFire multiprocessor design [6], the IBM Prism architecture [4], and the Illinois I-ACOMA architecture [13]. A model for comparing the performance of COMA and conventional CC-NUMA architectures is presented by Zhang and Torrellas [15]. Soundarajan et al. [11] describe the trade-offs related to data migration and replication in CC-NUMA machines.

Bibliography

1. Basu S, Torrellas J (1998) Enhancing memory use in simple coma: multiplexed simple coma. In: International symposium on high-performance computer architecture, Las Vegas, February 1998
2. Burkhardt H et al (1992) Overview of the KSRI computer system. Technical Report 9202001, Kendall Square Research, Waltham, February 1992
3. Dahlgren F, Torrellas J (1999) Cache-only memory architectures. IEEE Computer Magazine 32(6):72–79, June 1999
4. Ekanadham K, Lim B-H, Pattnaik P, Snir M (1998) PRISM: an integrated architecture for scalable shared memory. In: International symposium on high-performance computer architecture, Las Vegas, February 1998

5. Falsafi B, Wood D (1997) Reactive NUMA: a design for unifying S-COMA and CC-NUMA. In: International symposium on computer architecture, Denver, June 1997
6. Hagersten E, Koster M (1992) WildFire: a scalable path for SMPs. In: International symposium on high-performance computer architecture, Orlando, January 1999
7. Hagersten E, Landin A, Haridi S (1992) DDM – a cache-only memory architecture. *IEEE Computer* 25(9):44–54
8. Joe T, Hennessy J (1994) Evaluating the memory overhead required for COMA architectures. In: International symposium on computer architecture, Chicago, April 1994, pp 82–93
9. Moga A, Dubois M (1998) The effectiveness of SRAM network caches in clustered DSMs. In: International symposium on high-performance computer architecture, Las Vegas, February 1998
10. Saulsbury A, Wilkinson T, Carter J, Landin A (1995) An argument for simple COMA. In: International symposium on high-performance computer architecture, Raleigh, January 1995, pp 276–285
11. Soundararajan V, Heinrich M, Verghese B, Gharachorloo K, Gupta A, Hennessy J (1998) Flexible use of memory for replication/migration in cache-coherent DSM multiprocessors. In: International symposium on computer architecture, Barcelona, June 1998
12. Stenstrom P, Joe T, Gupta A (1992) Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In: International symposium on computer architecture, Gold Coast, Australia, May 1992, pp 80–91
13. Torrellas J, Padua D (1996) The illinois aggressive coma multiprocessor project (I-ACOMA). In: Symposium on the frontiers of massively parallel computing, Annapolis, October 1996
14. Zhang Z, Cintra M, Torrellas J (1999) Excel-NUMA: toward programmability, simplicity, and high performance. *IEEE Trans Comput* 48(2):256–264. Special Issue on Cache Memory, February 1999
15. Zhang Z, Torrellas J (1997) Reducing remote conflict misses: NUMA with remote cache versus COMA. In: International symposium on high-performance computer architecture, San Antonio, February 1997, pp 272–281

Caches, NUMA

- [NUMA Caches](#)

Calculus of Mobile Processes

- [Pi-Calculus](#)

Carbon Cycle Research

- [Terrestrial Ecosystem Carbon Modeling](#)

Car-Parrinello Method

MARK TUCKERMAN¹, ERIC J. BOHM², LAXMIKANT V. KALÉ², GLENN MARTYNA³

¹New York University, New York, NY, USA

²University of Illinois at Urbana-Champaign, Urbana, IL, USA

³IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA

Synonyms

Ab initio molecular dynamics; First-principles molecular dynamics

Definition

A Car–Parrinello simulation is a molecular dynamics based calculation in which the finite-temperature dynamics of a system of N atoms is generated using forces obtained directly from electronic structure calculations performed “on the fly” as the simulation proceeds. A typical Car–Parrinello simulation employs a density functional description of the electronic structure, a plane-wave basis expansion of the single-particle orbitals, and periodic boundary conditions on the simulation cell. The original paper has seen an exponential rise in the number of citations, and the method has become a workhorse for studying systems which undergo nontrivial electronic structure changes.

Discussion

Introduction

Atomistic modeling of many systems in physics, chemistry, biology, and materials science requires explicit treatment of chemical bond-breaking and forming events. The methodology of *ab initio* molecular dynamics (AIMD), in which the finite-temperature dynamics of a system of N atoms is generated using forces obtained directly from the electronic structure calculations performed “on the fly” as the simulation proceeds, can describe such processes in a manner that

is both general and transferable from system to system. The Car–Parrinello method is one type of AIMD simulation.

Assuming that a physical system can be described classically in terms of its N constituent atoms, having masses M_1, \dots, M_N and charges Z_1e, \dots, Z_Ne , the classical microscopic state of the system is completely determined by specifying the Cartesian positions $\mathbf{R}_1, \dots, \mathbf{R}_N$ of its atoms and their corresponding conjugate momenta $\mathbf{P}_1, \dots, \mathbf{P}_N$ as functions of time. In a standard molecular dynamics calculation, the time evolution of the system is determined by solving Hamilton's equations of motion

$$\dot{\mathbf{R}}_I = \frac{\mathbf{P}_I}{M_I}, \quad \dot{\mathbf{P}}_I = \mathbf{F}_I$$

which can be combined into the second-order differential equations

$$M_I \ddot{\mathbf{R}}_I = \mathbf{F}_I$$

Here, $\dot{\mathbf{R}}_I = d\mathbf{R}_I/dt$, $\dot{\mathbf{P}}_I = d\mathbf{P}_I/dt$ are the first derivatives with respect to time of position and momentum, respectively, and $\ddot{\mathbf{R}}_I = d^2\mathbf{R}_I/dt^2$ is the second derivative of position with respect to time, and \mathbf{F}_I is the total force on atom I due to all of the other atoms in the system. The force \mathbf{F}_I is a function $\mathbf{F}_I(\mathbf{R}_1, \dots, \mathbf{R}_N)$ of all of the atomic positions, hence Newton's equations of motion constitute a set of $3N$ coupled second-order ordinary differential equations.

Any molecular dynamics calculation requires the functional form $\mathbf{F}_I(\mathbf{R}_1, \dots, \mathbf{R}_N)$ of the forces as an input to the method. In most molecular dynamics calculations, the forces are modeled using simple functions that describe bond stretching, angle bending, torsion, van der Waals, and Coulombic interactions and a set of parameters for these interactions that are fit either to experiment or to high-level *ab initio* calculations. Such models are referred to as *force fields*, and while force fields are useful for many types of applications, they generally are unable to describe chemical bond-breaking and forming events and often neglect electronic polarization effects. Moreover, the parameters cannot be assumed to remain valid in thermodynamic states very different from the one for which they were originally fit. Consequently, most force fields are unsuitable for studying chemical processes under varying external conditions.

An AIMD calculation circumvents the need for an explicit force field model by obtaining the forces $\mathbf{F}_I(\mathbf{R}_1, \dots, \mathbf{R}_N)$ at a given configuration $\mathbf{R}_1, \dots, \mathbf{R}_N$ of the nuclei from a quantum mechanical electronic structure calculation performed at this particular nuclear configuration. To simplify the notation, let \mathbf{R} denote the complete set $\mathbf{R}_1, \dots, \mathbf{R}_N$ nuclear coordinates. Suppose the forces on the Born–Oppenheimer electronic ground state surface are sought. Let $|\Psi_0(\mathbf{R})\rangle$ and $\hat{H}_{\text{el}}(\mathbf{R})$ denote, respectively, the ground-state electronic wave function and electronic Hamiltonian at the nuclear configuration \mathbf{R} . If the system contains N_e electrons with position operators $\hat{\mathbf{r}}_1, \dots, \hat{\mathbf{r}}_{N_e}$, then the electronic Hamiltonian in atomic units ($e = 1, \hbar = 1, m_e = 1, c = 1$) is

$$\hat{H}_{\text{el}} = -\frac{1}{2} \sum_{i=1}^{N_e} \nabla_i^2 + \sum_{i>j} \frac{1}{|\hat{\mathbf{r}}_i - \hat{\mathbf{r}}_j|} - \sum_{i=1}^{N_e} \sum_{I=1}^N \frac{Z_I}{|\hat{\mathbf{r}}_i - \mathbf{R}_I|}$$

where the first, second, and third terms are the electron kinetic energy, the electron–electron Coulomb repulsion, and the electron–nuclear Coulomb attraction, respectively. The interatomic forces are given exactly by

$$\begin{aligned} \mathbf{F}_I(\mathbf{R}) = & -\langle \Psi_0(\mathbf{R}) | \nabla_I \hat{H}_{\text{el}}(\mathbf{R}) | \Psi_0(\mathbf{R}) \rangle \\ & + \sum_{J \neq I} Z_I Z_J \frac{(\mathbf{R}_I - \mathbf{R}_J)}{|\mathbf{R}_I - \mathbf{R}_J|^3} \end{aligned}$$

by virtue of the Hellman–Feynman theorem.

In practice, it is usually not possible to obtain the exact ground-state wave function $|\Psi_0(\mathbf{R})\rangle$, and, therefore, an approximate electronic structure method is needed. The approximation most commonly employed in AIMD calculations is the Kohn–Sham formulation of density functional theory. In the Kohn–Sham theory, the full electronic wave function is replaced by a set $\psi_s(\mathbf{r})$, $s = 1, \dots, N_s$ of mutually orthogonal single-electron orbitals (denoted collectively as $\psi(\mathbf{r})$) and the corresponding electron density

$$n(\mathbf{r}) = \sum_{s=1}^{N_s} f_s |\psi_s(\mathbf{r})|^2$$

where f_s is the occupation number of the state s . In closed-shell calculations $N_s = N_e/2$ with $f_s = 2$, while for open-shell calculations $N_s = N_e$ with $f_s = 1$. The Kohn–Sham energy functional gives the total energy of the system as

$$E[\psi, \mathbf{R}] = T_s[\psi] + E_H[n] + E_{\text{ext}}[n] + E_{\text{xc}}[n]$$

where

$$\begin{aligned} T_s[\psi] &= -\frac{1}{2} \sum_{s=1}^{N_s} f_s \int d\mathbf{r} \psi_s^*(\mathbf{r}) \nabla^2 \psi_s(\mathbf{r}) \\ E_H[n] &= \frac{1}{2} \int d\mathbf{r} d\mathbf{r}' \frac{n(\mathbf{r})n(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} \\ E_{\text{ext}}[n] &= -\sum_{I=1}^N \int d\mathbf{r} \frac{Z_I n(\mathbf{r})}{|\mathbf{r} - \mathbf{R}_I|} \end{aligned}$$

are the single-particle kinetic energy, the Hartree energy, and the external energy, respectively. The functional dependence of the term $E_{xc}[n]$, known as the exchange and correlation energy, is not known and must, therefore, be approximated. One of the most commonly used approximations is referred to as the *generalized gradient approximation*

$$E_{xc}[n] \approx \int d\mathbf{r} f(n(\mathbf{r}), \nabla n(\mathbf{r}))$$

where f is a scalar function of the density and its gradient. When the Kohn–Sham functional is minimized with respect to the electronic orbitals subject to the orthogonality condition $\langle \psi_s | \psi_{s'} \rangle = \delta_{ss'}$, then the interatomic forces are given by

$$\mathbf{F}_I(\mathbf{R}) = -\nabla_I E[\psi^{(0)}, \mathbf{R}] + \sum_{J \neq I} Z_I Z_J \frac{(\mathbf{R}_I - \mathbf{R}_J)}{|\mathbf{R}_I - \mathbf{R}_J|^3}$$

where $\psi^{(0)}$ denotes the set of orbitals obtained by the minimization procedure.

Most AIMD calculations use a basis set for expanding the Kohn–Sham orbitals $\psi_s(\mathbf{r})$. Because periodic boundary conditions are typically employed in molecular dynamics simulations, a useful basis set is a simple plane-wave basis. In fact, when the potential is periodic, the Kohn–Sham orbitals must be represented as Bloch functions, $\psi_s^{\mathbf{k}}(\mathbf{r}) = \exp(i\mathbf{k} \cdot \mathbf{r}) u_s(\mathbf{r})$, where \mathbf{k} is a vector in the first Brillouin zone. However, if a large enough simulation cell is used, \mathbf{k} can be taken to be $(0, 0, 0)$ (the Gamma-point) for many chemical systems, as will be done here. In this case, the plane-wave expansion of $\psi_s(\mathbf{r})$ becomes the simple Fourier representation

$$\psi_s(\mathbf{r}) = \frac{1}{\sqrt{V}} \sum_{\mathbf{g}} C_s(\mathbf{g}) e^{i\mathbf{g} \cdot \mathbf{r}}$$

where $\mathbf{g} = 2\pi\mathbf{in}/V^{1/3}$, with \mathbf{n} a vector of integers, denotes the Fourier-space vector corresponding to a cubic box of volume V , and the $\{C_s(\mathbf{g})\}$ is the set

of expansion coefficients. At the Gamma-point, the orbitals are purely real, and the coefficients satisfy the condition $C_s^*(\mathbf{g}) = C_s(-\mathbf{g})$, which means that only half of the reciprocal space is needed to reconstruct the full set of Kohn–Sham orbitals. A similar expansion

$$n(\mathbf{r}) = \frac{1}{V} \sum_{\mathbf{g}} \tilde{n}(\mathbf{g}) e^{i\mathbf{g} \cdot \mathbf{r}}$$

is employed for the electronic density. Note that the coefficients $\tilde{n}(\mathbf{g})$ depend on the orbital coefficients $C_s(\mathbf{g})$. Because the density is real, the coefficients $\tilde{n}(\mathbf{g})$ satisfy $\tilde{n}^*(\mathbf{g}) = \tilde{n}(-\mathbf{g})$. Again, this condition means that the full density can be reconstructed using only half of the reciprocal space. In order to implement these expansions numerically, they must be truncated. The truncation criterion is based on the plane-wave kinetic energy $|\mathbf{g}|^2/2$. Specifically, the orbital expansion is truncated at a value E_{cut} such that $|\mathbf{g}|^2/2 < E_{\text{cut}}$, and the density, being determined from the squares of the orbitals, is truncated using the condition $|\mathbf{g}|^2/2 < 4E_{\text{cut}}$. When the above two plane-wave expansions are substituted into the Kohn–Sham energy functional, the resulting energy is an ordinary function of the orbital expansion coefficients that must be minimized with respect to these coefficients subject to the orthonormality constraint $\sum_{\mathbf{g}} C_s^*(\mathbf{g}) C_{s'}(\mathbf{g}) = \delta_{ss'}$.

An alternative to explicit minimization of the energy was proposed by Car and Parrinello (see bibliographic notes) based on the introduction of an artificial dynamics for the orbital coefficients

$$\begin{aligned} \mu \ddot{C}_s(\mathbf{g}) &= -\frac{\partial E}{\partial C_s^*(\mathbf{g})} - \sum_{s'} \Lambda_{ss'} C_{s'}(\mathbf{g}) \\ M_I \ddot{\mathbf{R}}_I &= -\frac{\partial E}{\partial \mathbf{R}_I} \end{aligned}$$

In the above equations, known as the *Car–Parrinello equations*, μ is a mass-like parameter (having units of $\text{energy} \times \text{time}^2$) that determines the time scale on which the coefficients evolve, and $\Lambda_{ss'}$ is a matrix of Lagrange multipliers for enforcing the orthogonality condition as a constraint. The mechanism of the Car–Parrinello equations is the following: An explicit minimization of the energy with respect to the orbital coefficients is carried out at a given initial configuration of the nuclei. Following this, the Car–Parrinello equations are integrated numerically with a value of μ small enough to

ensure that the coefficients respond as quickly as possible to the motion of the nuclei. In addition, the orbital coefficients must be assigned a fictitious kinetic energy satisfying

$$\mu \sum_s \sum_{\mathbf{g}} |\dot{C}_s(\mathbf{g})|^2 \ll \sum_I M_I \dot{\mathbf{R}}_I^2$$

in order to ensure that the coefficients remain close to the ground-state Born–Oppenheimer surface throughout the simulation.

In a typical calculation containing 10^2 – 10^3 nuclei and a comparable number of Kohn–Sham orbitals, the number of coefficients per orbital is often in the range 10^5 – 10^7 , depending on the atomic types present in the system. Although this may seem like a large number of coefficients, the number would be considerably larger without a crucial approximation applied to the external energy $E_{\text{ext}}[n]$. Specifically, electrons in low-lying energy states, also known as *core electrons*, are eliminated in favor of an *atomic pseudopotential* that requires replacing the exact $E_{\text{ext}}[n]$ by a functional of the following form

$$\begin{aligned} E_{\text{ext}}[n, \psi] &\approx \sum_{I=1}^N \int d\mathbf{r} n(\mathbf{r}) v_{\bar{l}}(\mathbf{r} - \mathbf{R}_I) + E_{\text{nl}}[\psi] \\ &\equiv E_{\text{loc}}[n] + E_{\text{nl}}[\psi] \end{aligned}$$

where $E_{\text{loc}}[n]$ is a purely local energy term, and $E_{\text{nl}}[\psi]$ is an orbital-dependent functional known as the *nonlocal energy* given by

$$E_{\text{nl}}[\psi] = \sum_s \sum_I \sum_{l=0}^{\bar{l}-1} \sum_{m=-l}^l w_l |Z_{sIlm}|^2$$

with

$$Z_{sIlm} = \int d\mathbf{r} F_{lm}(\mathbf{r} - \mathbf{R}_I) \psi_s(\mathbf{r})$$

Here $F_{lm}(\mathbf{r})$ is a function particular to the pseudopotential, and l and m label angular momentum channels. The value of l is summed up to a maximum $\bar{l} - 1$. Typically, $\bar{l} = 1$ or 2 depending on the chemical composition of the system, but higher values can be included when necessary. Despite the pseudopotential approximation, AIMD calculations in a plane-wave basis are computationally very intensive and can benefit substantially from massive parallelization. In addition, a typical value of the integration time step in a Car–Parrinello AIMD simulation is 0.1 fs, which means that 10^5 – 10^6 or more electronic structure calculations are needed to generate

a trajectory of just 10–100 ps. Thus, the parallelization scheme must, therefore, be efficient and scale well with the number of processors. The remainder of this entry will be devoted to the discussion of the algorithm and parallelization techniques for it.



Outline of the Algorithm

The calculation of the total energy and its derivatives with respect to orbital expansion coefficients and nuclear positions consists of the following phases:

- Phase I: Starting with the orbital coefficients in reciprocal space, the electron kinetic energy and its coefficient derivatives are evaluated using the formula
- $$T_s = \frac{1}{2} \sum_{s=1}^{N_s} f_s \sum_{\mathbf{g}} |\mathbf{g}|^2 |C_s(\mathbf{g})|^2$$
- Phase II: The orbital coefficients are transformed from Fourier space to real space. This operation requires N_s three-dimensional fast Fourier transforms (FFTs).
 - Phase III: The real-space coefficients are squared and summed to generate the density $n(\mathbf{r})$.
 - Phase IV: The real-space density $n(\mathbf{r})$ is used to evaluate the exchange-correlation functional and its functional derivatives with respect to the density. Note that $E_{\text{xc}}[n]$ is generally evaluated on a regular mesh, hence, the functional derivatives are replaced by ordinary derivatives at the mesh points.
 - Phase V: The density is Fourier transformed to reciprocal space, and the coefficients $\tilde{n}(\mathbf{g})$ are used to evaluate the Hartree and purely local part of the pseudopotential energy using the formulas

$$E_H = \frac{1}{V} \sum_{\mathbf{g} \neq (0,0,0)} \frac{2\pi}{|\mathbf{g}|^2} |\tilde{n}(\mathbf{g})|^2$$

$$E_{\text{loc}} = \frac{1}{V} \sum_{I=1}^N \sum_{\mathbf{g}} \tilde{n}^*(\mathbf{g}) \tilde{v}_{\bar{l}}(\mathbf{g}) e^{-i\mathbf{g} \cdot \mathbf{R}_I}$$

and their derivatives and nuclear position derivatives, where $\tilde{v}_{\bar{l}}(\mathbf{g})$ is the Fourier transform of the potential $v_{\bar{l}}(\mathbf{r})$.

- Phase VI: The derivatives from Phase VI are Fourier transformed to real space and combined with the derivatives from Phase V. The combined functional

derivatives are multiplied against the real-space orbital coefficients to produce part of the orbital forces in real space.

7. Phase VII: The reciprocal-space orbital coefficients are used to evaluate the nonlocal pseudopotential energy and its derivatives.
8. Phase VIII: The forces from Phase VI are combined and are then transformed back into reciprocal space, an operation that requires N_s FFTs.
9. Phase IX: The nuclear–nuclear Coulomb repulsion energy and its position derivatives are evaluated using standard Ewald summation.
10. Phase X: The reciprocal space forces are combined with those from Phases I and VII to yield the total orbital forces. These, together with the nuclear forces, are fed into a numerical solver in order to advance the nuclear positions and reciprocal-space orbital coefficients to the next time step, and the process returns to Phase I. As part of this phase, the condition of orthogonality

$$\langle \psi_s | \psi_{s'} \rangle = \sum_{\mathbf{g}} C_s^*(\mathbf{g}) C_{s'}(\mathbf{g}) = \delta_{ss'}$$

is enforced as a holonomic constraint on the dynamics.

Parallelization

Two basic strategies for parallelization of *ab initio* molecular dynamics will be outlined. The first is a hybrid state/grid-level parallelization scheme useful on machines with a modest number of fast processors having large memory but with slow communication. This scheme does not require a parallel fast Fourier transform (FFT) and can be coded up relatively easily starting from an optimized serial code. The second scheme is a fine-grained parallelization approach based on parallelization of all operations and is useful for massively parallel architectures. The tradeoff of this scheme is its considerable increase in coding complexity. An intermediate scheme between these builds a parallel FFT into the hybrid scheme as a means of reducing the memory and some of the communication requirements. In all such schemes if the FFT used is a complex-to-complex FFT and the orbitals are real, then the states can be double packed into the FFT routine and transformed two at a time, which increases the overall efficiency of the algorithms to be presented below.

Hybrid scheme – Let `ncoef` represent the number of coefficients used to represent each Kohn–Sham orbital, and let `nstate` represent the number of orbitals (also called “states”). In a serial calculation, the coefficients are then stored in two arrays `a(ncoef, state)` and `b(ncoef, nstate)` holding the real and imaginary parts of the coefficients, respectively. Alternatively, complex data typing could be used for the coefficient array. Let `nproc` be the number of processors available for the calculation. In the hybrid parallel scheme, the coefficients are stored in one of two ways at each point in the calculation. The default storage mode is called “transposed” form in which the coefficient arrays are dimensioned as `a(ncoef/nproc, nstate)` and `b(ncoef/nproc, nstate)`, so that each processor has a portion of the orbitals for all of the states. At certain points in the calculation, the coefficients are transformed to “normal” form in which the arrays are dimensioned as `a(ncoef, nstate/nproc)` and `b(ncoef, nstate/nproc)`, so that each processor has a fraction of the orbitals but each of the orbitals is complete on the processor.

In the hybrid scheme, operations on the density, both in real and reciprocal spaces, are carried out in parallel. These terms include the Hartree and local pseudopotential energies, which are carried out on a spherical reciprocal-space grid, and the exchange-correlation energy, which is carried out on a rectangular real-space grid. The Hartree and local pseudopotential terms require arrays `gdens_a(ndens/nproc)` and `gdens_b(ndens/nproc)` that hold, on each processor, a portion of the real and imaginary reciprocal-space density coefficients $\tilde{n}(\mathbf{g})$. Here, `ndens` is the number of reciprocal-space density coefficients. The exchange-correlation energy makes use of an array `rdens(ngrid/nproc)` that holds, on each processor, a portion of the real-space density $n(\mathbf{r})$. Here, `ngrid` is the number of points on the rectangular grid.

Given this division of data over the processors, the algorithm proceeds in the following steps:

1. With the coefficients in transposed form, each processor calculates its contribution to the electronic kinetic energy and the corresponding forces on the coefficients it holds. A reduction is performed to obtain the total kinetic energy.

2. The coefficient arrays are transposed into normal form, and each processor must subsequently perform $0.5A * nstate / nproc$ three-dimensional serial FFTs on its subset of states to obtain the corresponding orbitals in real space. These orbitals are stored as `creal(ngrid, nstate/nproc)`. Each processor sums the square of this orbital over $nstate / nproc$ at each grid point.
3. Each processor performs a serial FFT on its portion of the density to obtain its contribution to the reciprocal-space density coefficients $\tilde{n}(\mathbf{g})$.
4. `Reduce_scatter` operations are used to sum each processor's contributions to the real and reciprocal-space densities and distribute `ngrid / nproc` and `ndens / nproc` real and reciprocal-space density values on each processor.
5. Each processor calculates its contribution to the Hartree and local pseudopotential energies, Kohn-Sham potential, and nuclear forces using its reciprocal-space density coefficients and exchange-correlation energies and Kohn-Sham potential using its real-space density coefficients.
6. As the full Kohn-Sham potential is needed on each processor, `Allgather` operators are used to collect the reciprocal and real-space potentials. The reciprocal-space potential is additionally transformed to real space by a single FFT.
7. With the coefficients in normal form, the Kohn-Sham potential contributions to the coefficient forces are computed via the product $V_{KS}(\mathbf{r})\psi_s(\mathbf{r})$. Each processor computes this product for the states it has.
8. Each processor transforms its coefficient force contributions $V_{KS}(\mathbf{r})\psi_s(\mathbf{r})$ back to reciprocal space by performing $0.5 * nstate / nproc$ serial FFTs.
9. With the coefficients in normal form, each processor calculates its contribution to the nonlocal pseudopotential energy, coefficient forces, and nuclear forces for its states.
10. Each processor adds its contributions to the nonlocal forces to those from Steps 9 and 1 to obtain the total coefficient forces in normal form. The energies and nuclear forces are reduced over processors.
11. The coefficients and coefficient forces are transformed back to transposed form, and the equations of motion are integrated with the coefficients in transposed form.
12. In order to enforce the orthogonality constraints, various partially integrated coefficient and coefficient velocity arrays are multiplied together in transposed form on each processor. In this way, each processor has a set of $N_s \times N_s$ matrices that are reduced over processors to obtain the corresponding $N_s \times N_s$ matrices needed to obtain the Lagrange multipliers. The Lagrange multiplier matrix is broadcast to all of the processors and each processor applies this matrix on its subset of the coefficient or coefficient velocity array.

Intermediate scheme – The storage requirements of this scheme are the same as those of the hybrid approach except that the coefficient arrays are only used in their transposed form. Since all FFTs are performed in parallel, a full grid is never required, which cuts down on the memory and scratch-space requirements. The key to this scheme is having a parallel FFT capable of achieving good load balance in the presence of the spherical truncation of the reciprocal-space coefficient and density grids.

This algorithm is carried out as follows:

1. With the coefficients in transposed form, each processor calculates its contribution to the electronic kinetic energy and the corresponding forces on the coefficients it holds. A reduction is performed to obtain the total kinetic energy.
2. A set of $0.5 * nstate$ three-dimensional parallel FFTs is performed in order to transform the coefficients to corresponding orbitals in real space. These orbitals are stored in an array with dimensions `creal(ngrid/nproc, nstate)`. Since each processor has a full set of state, the real-space orbitals can simply be squared so that each processor has the full density on its subset of `ngrid / nproc` grid points.
3. A parallel FFT is performed on the density to obtain the reciprocal-space density coefficients $\tilde{n}(\mathbf{g})$, which are also divided over the processors. Each processor will have `ndens / nproc` coefficients.
4. Each processor calculates its contribution to the Hartree and local pseudopotential energies, Kohn-Sham potential, and nuclear forces using its reciprocal-space density coefficients. Each processor

also calculates its contribution to the exchange-correlation energies and Kohn–Sham potential using its real-space density coefficients.

5. A parallel FFT is used to transform the reciprocal-space Kohn–Sham potential to real space.
6. With the coefficients in transposed form, the Kohn–Sham potential contributions to the coefficient forces are computed via the product $V_{KS}(\mathbf{r})\psi_s(\mathbf{r})$. Each processor computes this product at the grid points it has.
7. The coefficient force contributions $V_{KS}(\mathbf{r})\psi_s(\mathbf{r})$ on each processor are transformed back to reciprocal space via a set of $0.5 * nstate$ three-dimensional parallel FFTs.
8. With the coefficients in transposed form, each processor calculates the nonlocal pseudopotential energy, coefficient forces, and nuclear forces using its subset of reciprocal-space grid points.
9. Each processor adds its contributions to the non-local forces to those from Steps 8 and 1 to obtain the total coefficient forces in transposed form. The energies and nuclear forces are reduced over processors.
10. In order to enforce the orthogonality constraints, various partially integrated coefficient and coefficient velocity arrays are multiplied together in transposed form on each processor. In this way, each processor has a set of $N_s \times N_s$ matrices that are reduced over processors to obtain the corresponding $N_s \times N_s$ matrices needed to obtain the Lagrange multipliers. The Lagrange multiplier matrix is broadcast to all of the processors and each processor applies this matrix on its subset of the coefficient or coefficient velocity array.

3D parallel FFTs and fine-grained data decomposition – Any fine-grained parallelization scheme for Car–Parrinello molecular dynamics requires a scalable three-dimensional FFT and a data decomposition strategy that allows parallel operations to scale up to large numbers of processors. Much of this is true even for the intermediate scheme discussed above.

Starting with the 3D parallel FFT, when the full set of indices of the coefficients $C_s(\mathbf{g})$ are displayed, the array appears as $C_s(g_x, g_y, g_z)$, where the reciprocal-space points lie within a sphere of radius $\sqrt{2}E_{cut}$. A common approach for transforming this array

to real space is based on data transposes. First, a set of one-dimensional FFTs is computed to yield $C_s(g_x, g_y, g_z) \longrightarrow \tilde{C}_s(g_x, g_y, z)$. Since the data is dense in real space, this operation transforms the sphere into a cylinder. Next, a transpose is performed on the z index to parallelize it and collect complete planes of g_x and g_y along the cylindrical axis. Once this operation is complete, the remaining two one-dimensional FFTs can be performed. The first maps the cylinder onto a rectangular slab, and the final FFT transforms the slab into a dense real-space array $\psi_s(x, y, z)$.

Given this strategy for the 3D FFT, the optimal data decomposition scheme is based on dividing the state and density arrays into planes both in reciprocal space in and in real space. Briefly, it is useful to think of dividing the coefficient array into data objects represented as $G(s, p)$, where p indexes the (g_x, g_y) planes. The planes are optimally grouped in such a way that complete lines along the g_z axis are created. This grouping is important due to the spherical truncation of reciprocal space. This type of virtualization allows parallelization tools such as the Charm++ software to be employed as a way of mapping the data objects to the physical processors as discussed by Bohm et al. in [9]. The granularity of the object $G(s, p)$ is something that can be tuned for each given architecture.

Once the data decomposition is accomplished, the planes must be mapped to the physical processors. A simple map that allocates all planes of a state to the same processor, for example, would make all 3D FFT transpose operations local, resulting in good performance. However, such a mapping is not scalable because massively parallel architectures will have many more processors than states. Another extreme would map all planes of the same rank in all of the states to the same processor. Unfortunately, this causes the transposes to be highly nonlocal and this leads to a communication bottleneck. Thus, the optimal mapping is a compromise between these two extremes, mapping collections of planes in a state partition to the physical processors, where the size of these collections depends on the number of processors and communication bandwidth. This mapping then enables the parallel computation of overlap matrices obtained by summing objects with different state indices s and s' over reciprocal space.

This entry is focused on the parallelization of plane-wave based Car–Parrinello molecular dynamics.

However, other basis sets offer certain advantages over plane waves. These are localized real-space basis sets useful for chemical applications where the orbitals $\psi_s(\mathbf{r})$ can be easily transformed into a maximally spatially localized form. Future work will focus on developing parallelization strategies for *ab initio* molecular dynamics calculations with such basis sets.

Bibliographic Notes and Further Reading

As noted in the introduction, the Car–Parrinello method was introduced in Ref. [1] and is discussed in greater detail in a number of review articles [2, 3] and a recent book [4]. Further details on the algorithms presented in this entry and their implementation in the open-source package PINY_MD can be found in Refs. [5, 6]. A detailed discussion of the Charm++ runtime environment alluded to in the “Parallelization” section can be found in Ref. [7]. Finally, fine-grained algorithms leveraging the Charm++ runtime environment in the manner described in this article are described in more detail in Refs. [8, 9]. These massively parallel techniques have been implemented in the Charm++ based open-source *ab initio* molecular dynamics package OpenAtom [10].

Bibliography

1. Car R, Parrinello M (1985) Unified approach for molecular dynamics and density-functional theory. *Phys Rev Lett* 55:2471
2. Marx D, Hutter J (2000) *Ab initio* molecular dynamics: theory and implementation in modern methods and algorithms of quantum chemistry. In: Grotendorst J (ed) Forschungszentrum, Jülich, NIC Series, vol 1. NIC Directors, Jülich, pp 301–449
3. Tuckerman ME (2002) *Ab initio* molecular dynamics: basic concepts, current trends, and novel applications. *J Phys Condens Mat* 14:R1297
4. Marx D, Hutter J (2009) *Ab initio* molecular dynamics: basic theory and advanced methods. Cambridge University Press, New York
5. Tuckerman ME, Yarne DA, Samuelson SO, Hughes AL, Martyna GJ (2000) Exploiting multiple levels of parallelism in molecular dynamics based calculations via modern techniques and software paradigms on distributed memory computers. *Comp Phys Commun* 128:333
6. The open-source package PINY MD is freely available for download via the link <http://homepages.nyu.edu/~mt33/PINYMD/PINY.html>
7. Kale LV, Krishnan S (1996) Charm++: parallel programming with message-driven objects. In: Wilson GV, Lu P

(eds) Parallel programming using C++. MIT, Cambridge, pp 175–213

8. Vadali RV, Shi Y, Kumar S, Kale LV, Tuckerman ME, Martyna GJ (2004) Scalable fine-grained parallelization of plane-wave-based *ab initio* molecular dynamics for large supercomputers. *J Comp Chem* 25:2006
9. Bohm E, Bhatele A, Kale LV, Tuckerman ME, Kumar S, Gunnels JA, Martyna GJ (2008) Fine-grained parallelization of the Car–Parrinello *ab initio* molecular dynamics method on the IBM Blue Gene/L supercomputer. *IBM J Res Dev* 52:159
10. OpenAtom is freely available for download via the link <http://charm.cs.uiuc.edu/OpenAtom>

CDC 6600

► [Control Data 6600](#)

Cedar Multiprocessor

PEN-CHUNG YEW

University of Minnesota at Twin-Cities, Minneapolis, MN, USA

Definition

The Cedar multiprocessor was designed and built at the Center for Supercomputing Research and Development (CSR) in the University of Illinois at Urbana-Champaign (UIUC) in 1980s. The project brought together a group of researchers in computer architecture, parallelizing compilers, parallel algorithms/applications, and operating system, to develop a scalable, hierarchical, shared-memory multiprocessor. It was the largest machine building effort in academia since ILLIAC-IV. The machine became operational in 1989 and decommissioned in 1994. Some pieces of the boards are still displayed in the Department of Computer Science at UIUC.

Discussion

Introduction

The Cedar multiprocessor was the first scalable, cluster-based, hierarchical shared-memory multiprocessor of its kind in 1980s. It was designed and built at the

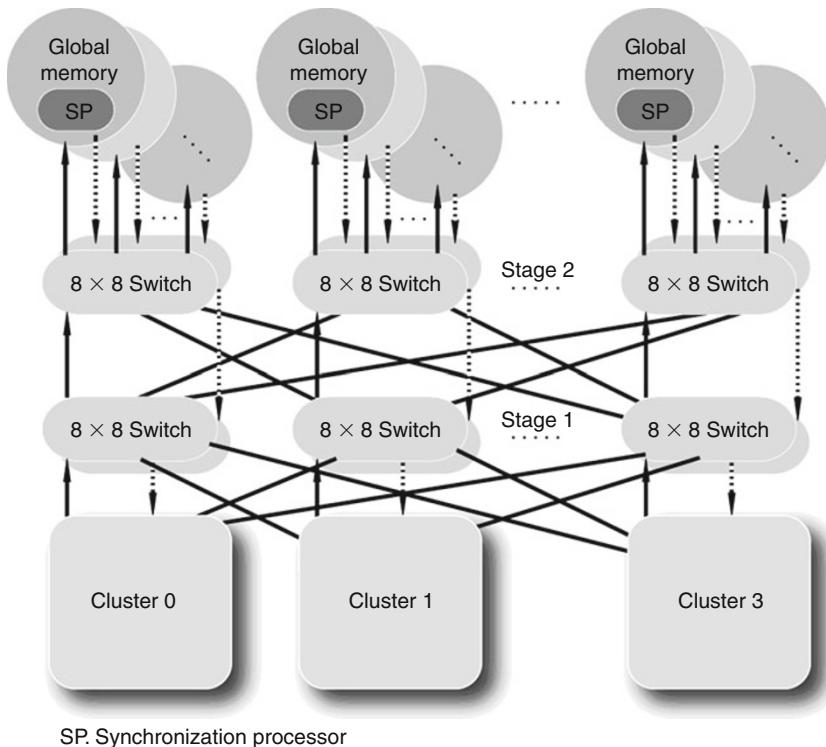
Center for Supercomputing Research and Development (CSRD) in the University of Illinois at Urbana-Champaign (UIUC). The project succeeded in building a complete scalable shared-memory multiprocessor system with a working 4-cluster (32-processor) hardware prototype [9], a parallelizing compiler for Cedar Fortran [2, 3, 8], and an operating system, called Xylem, for scalable multiprocessor [5]. Real application programs were ported and run on Cedar [6, 7] with performance studies presented in [9]. The Cedar project was started in 1984 and the prototype became functional in 1989.

Cedar had many features that later used extensively in large-scale multiprocessor systems, such as software-managed cache memory to avoid very expensive cache coherence hardware support; vector data prefetching to cluster memories for hiding long memory latency; parallelizing compiler techniques that take sequential applications and extract task-level parallelism from their loop structures; language extensions that include memory attributes of the data variables to allow programmers and compilers to manage data locality more easily; and parallel dense/sparse

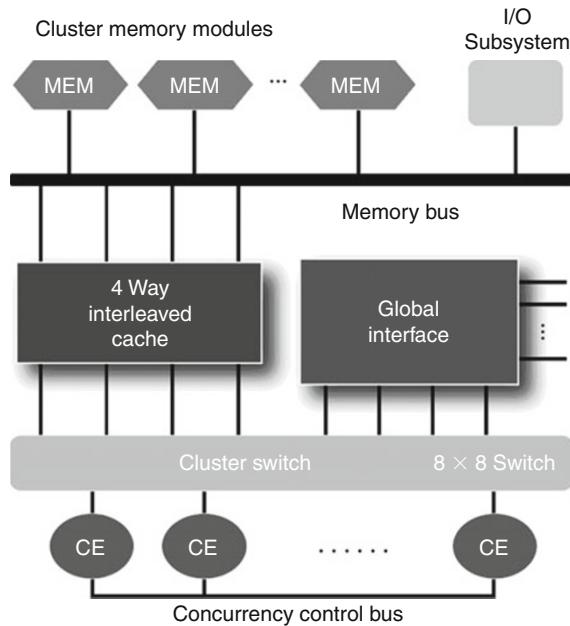
matrix algorithms that could speed up the most time consuming part of many linear systems in large-scale scientific applications.

Machine Organization of Cedar

The organization of Cedar consists of multiple *clusters* of processors connected through two high-bandwidth single-directional *global interconnection networks* (GINs) to a globally shared memory system (GSM) (see Fig. 1). One GIN provides memory requests from *clusters* to the GSM. The other GIN provides data and responses from GSM back to *clusters*. In the Cedar prototype built at CSRD, an Alliant FX/8 system was used as a cluster (See Fig. 2). Some hardware components in FX/8, such as the crossbar switch and the interface between the shared cache and the crossbar switch, were modified to accommodate additional ports for the *global network interface* (GNI). GNI provides pathways from a *cluster* to GIN. Each GNI board also has a software-controlled *vector prefetching unit* (VPU) that is very similar to the DMA in later IBM's Cell Broadband Engine.



Cedar Multiprocessor. Fig. 1 Cedar machine organization



Cedar Multiprocessor. Fig. 2 Cedar cluster

Cedar Cluster: In each Alliant system, there are eight processors, called *computational elements* (CEs). Those eight CEs are connected to a four-way interleaved shared cache through an 8×8 crossbar switch, and four ports to GNI that provide access to GIN and GSM (see Fig. 2). On the other side of the shared cache is a high-speed shared bus that is connected to multiple cluster memory modules and interactive processors (IPs). IPs handle input/output and network functions.

Each CE is a pipelined implementation of Motorola 68020 instruction set architecture, one of the most popular high-performance microprocessors in 1980s. The 68020 ISA was augmented with vector instructions. The vector unit includes both 64-bit floating-point and 32-bit integer operations. It also has eight 32-word vector registers. Each vector instruction could have one memory- and one register-operand to balance the use of registers and the requirement of memory bandwidth. The clock cycle time was 170 ns, but each CE has a peak performance of 11.8 Mflops for a multiply-and-add 64-bit floating-point vector instruction. It was a very high-performance microprocessor at that time, and was implemented on one printed-circuit board.

There is a *concurrency control bus* (CCB) that connects all eight CEs to provide synchronization and

coordination of all CEs. Concurrency control instructions include fast *fork*, *join*, and *fetch-and-increment* type of synchronization operations. For example, a single *concurrent start* instruction broadcast on CCB will spread the iterations and initiate the execution of a concurrent loop among all eight CEs simultaneously. Loop iterations are *self-scheduled* among CEs by fetch-and-incrementing a loop counter shared by all CEs. CCB also supports a *cascade synchronization* that enforces an ordered execution among CEs in the same cluster for a concurrent loop that requires a sequential ordering for a particular portion of the loop execution.

Memory Hierarchy: The 4 GB physical memory address space of Cedar is divided into two equal halves between the cluster memory and the GSM. There are 64 MB of GSM and 64 MB of cluster memory in each cluster on the Cedar prototype. It also supports a virtual memory system with a 4 KB page size. GSM could be directly addressed and shared by all clusters, but *cluster memory* is only addressable by the CEs within each cluster. Data coherence among multiple copies of data in different cluster memories is maintained explicitly through software by either programmer or the compiler. The peak GSM bandwidth is 768 MB/s and 24 MB/s per CE. The GSM is double-word interleaved, and aligned among all global memory modules. There is a *synchronization processor* in each GSM module that could execute each atomic Cedar synchronization instruction issued from a CE and staged at GNI.

Data Prefetching: It was observed early in the Cedar design phase that scientific application programs have very poor cache locality due to their vector operations. To compensate for the long access latency to GSM and to overcome the limitation of two outstanding memory requests per CE in the Alliant microarchitectural design, vector data prefetching is needed. The large GIN bandwidth is ideal for supporting such data prefetching.

To avoid major changes to Alliant's instruction set architecture (ISA) by introducing additional data prefetching instructions into its ISA, and also to avoid major alterations to CE's control and data paths of including a *data prefetching unit* (PFU) inside its CE, it was decided to build the PFU on the GIN board. Prefetched data is stored in a 512-word (each word is 8-byte) prefetch buffer inside PFU to avoid polluting the shared data cache and to allow data reuse. A CE will stage a prefetch operation at PFU by providing it with

the length, the stride, and the mask of the vector data to be prefetched. The prefetching operation could then be *fired* by providing the physical address of the first word of the vector. Prefetching operation could be overlapped with other computations or cluster memory operations.

When a page boundary is crossed during a data prefetching operation, the PFU will be suspended until the CE provides the starting address of the new page. In the absence of page crossing, PFU will prefetch 512 words without pausing into its prefetch buffer. Due to hardware limitation, only one prefetch buffer is implemented in each PFU. The prefetch buffer will thus be invalidated with another prefetch operation. Prefetched data could return from GSM out of order because of potential network and memory contentions. A *presence* bit per data word in the prefetch buffer allows the CE to both access the data without waiting for the completion of the prefetch instruction, and to access the prefetch data in the same order as requested. It was later proved from experiments that PFU is extremely useful in improving memory performance for scientific applications.

Data prefetching was later incorporated extensively into high-performance microprocessors that prefetch data from main memory into the last-level cache memory. Sophisticated compiler techniques that could insert prefetching instructions into user programs at compiler time or runtime have been developed and used extensively. Cache performance has shown significant improvement with the help of data prefetching.

Global Interconnection Network (GIN): The Cedar network is a multi-stage shuffle-exchange network as shown in Fig. 1. It is constructed with 8×8 crossbar switches with 64-bit wide data paths and some control signals for network flow control. As there are only 32-processors on the Cedar prototype, there are only two stages needed for each direction of GIN between clusters and GSM, i.e., it only need two clock cycles to go from GNI to one of the 32 GSM modules if there is no network contention. Hence, it provides a very low latency and high-bandwidth communication path to GSM from a CE. There is one GIN in each direction between clusters and GSM as mentioned above. To cut down possible signal noises and to maintain high reliability, all data signals between stages are implemented using differential signals. GIN is packet-switched and self-routed. Routing is based on a tag-controlled scheme

proposed in [11] for multi-stage shuffle-exchange networks. There is a unique path between each pair of GIN input and output ports. A two-word queue is provided at each input and output port of the 8×8 crossbar switch. Flow control between network stages is implemented to prevent buffer overflow and maintain low network contention. The total network bandwidth is 768 MB/s, and 24 MB/s each network port to match the bandwidth of GSM and CE. Hence, the requests and data flow from a CE to a GSM module and back to the CE. It forms a circular pipeline with balanced bandwidth that could support high-performance vector operations.

Memory-Based Synchronization: Given packet-switched multi-stage interconnection networks in Cedar, instead of a shared bus, it is very difficult to implement *atomic (indivisible)* synchronization operations such as a *test-and-set* or a *fetch-and-add* operation efficiently on the system interconnect. *Lock* and *unlock* operations are two low-level synchronization operations that will require multiple passes through GINs and GSM to implement a higher-level synchronization such as *fetch-and-add*, a very frequent synchronization operation in parallel applications that could be used to implement *barrier synchronizations* and *self-scheduling* for parallel loops.

Cedar implements a sophisticated set of synchronization operations [14]. They are very useful in supporting parallel loop execution that requires frequent fine-grained data synchronization between loop iterations that have cross-iteration data dependences, so called *doacross* loops [13]. Cedar synchronization instructions are basically *test-and-operate* operations, where *test* is any relational operation on 32-bit data (e.g., \leq) and *operate* could be a *read*, *write*, *add*, *subtract*, or *logical* operation on 32-bit data. These Cedar synchronization operations are also staged and controlled in a GNI board by each CE. There is also a *synchronization unit* at each GSM module that executes these atomic operations at GSM. It is a very efficient and effective way of implementing atomic global synchronization operations right at GSM.

Hardware Performance Monitoring System: It was determined at the Cedar design time that performance tuning and monitoring on a large-scale multiprocessor requires extensive support starting at the lowest hardware level. Important and critical system signals in all major system components that include GIN

and GSM must be made observable. To minimize the amount of hardware changes needed on Alliant clusters, it was decided to use external monitoring hardware to collect time-stamped event traces and histograms of various hardware signals. This allows the hardware performance monitoring system to evolve over time without having to make major hardware changes on the Cedar prototype. The Cedar hardware event tracers can each collect 1M events and the histogrammers have 64K 32-bit counters. These hardware tracers could be cascaded to capture more events. The triggering and stopping signals or events could come from special library calls in application programs, or some hardware signals such as a *GSM request* to indicate a shared cache miss. Software tools are built to support starting and stopping of tracers, off-loading data from the tracers and counters for extensive performance analysis.

Xylem Operating System

Xylem operating system [5] provides support for parallel execution of application programs on Cedar. It is based on the notion that a parallel user program is a flow graph of executable nodes. New system calls are added to allow Unix processes to create and control multiple tasks. It basically links the four separate operating systems in Alliant clusters into one Cedar operating system. Xylem provides virtual memory, scheduling and file system services for Cedar.

The Xylem scheduler schedules those tasks instead of the Unix scheduler. A Xylem task corresponds to one or more executions of each node in the program flowgraph. Support of multiprocessing in Xylem basically includes *create_task*, *delete_task*, *start_task*, *end_task*, and *wait_task*. The reason for separating *create_task* and *start_task* is because task creation is a very expensive operation, while starting a task is a relatively faster operation. Separating these two operations allow more efficient management of tasks, same for separating *delete_task* and *end_task* operations. For example, helper tasks could be created in the beginning of a process then later started when parallel loops are being executed. These helper tasks could be put back to the idle state without being deleted and recreated later.

There is no parent-child relationship between the *creator task* and the *created task* though. Either task could wait for, start or delete the other task. A task could delete itself if it is the last task in the process. *End_task*

marks the task idle and stops execution. *Wait_task* (*tasknum*) blocks the execution of the calling task until the task specified by *tasknum* enters an idle state. A task enters an idle state when it calls *end_task*. Hence, the waiting task will be unblocked when the task it is waiting for (identified by *tasknum*) ends execution.

Memory management of Xylem is based on a paging system. It has the notion of global memory pages and cluster memory pages, and provides kernels to allow a task to control its own memory environment or that of any other task in the process. They include kernels to allocate and de-allocate pages for any task in the process; change the attributes of pages; make copies and share pages with any other task in the process; and unmap pages from any task in the process. The attributes of a page include execute/no-execute, read-write/no-read-write, local/global/global-cached, and shared/private-copy/private-new/private.

Programming Cedar

A parallel program can be written using Cedar Fortran, a parallel dialect of the Fortran language, and Fortran 77 on Cedar. Cedar Fortran supports all key features of the Cedar system described above. Those Cedar features include its memory hierarchy, the data prefetching capability from GSM, the Cedar synchronization operations, and the concurrency control features on CCB. These features are supported through a language extension to Fortran 77. Programs written in Fortran 77 could be restructured by a *parallelizing restructurer*, and translated into parallel programs in Cedar Fortran. They are then fed into a backend compiler, mostly an enhanced and modified Alliant compiler, to produce Cedar executables. The *parallelizing restructurer* was based on the *KAP restructurer* [10], a product of *Kuck and Associates (KAI)*.

Cedar Fortran has many features common to the ANSI Technical Committee X3H5 standard for parallel Fortran whose basis was PCF Fortran developed by the Parallel Computing Forum (PCF). The main features encompass parallel loops, vector operations that include vector reduction operations and a WHERE statement that could mask vector assignment as in Fortran-90, declaration of *visibility* (or *accessibility*) of data, and post/wait synchronization.

Several variations of parallel loops that take into account the loop structure and the Cedar organization

are included in Cedar Fortran. There are basically two types of parallel loops: *doall* loops and *doacross* loops. A *doacross* loop is an *ordered* parallel loop whose loop iterations start sequentially as in its original sequential order. It uses *cascade synchronization* on CCB to enforce such sequential order on parts of its loop body (see Fig. 4). A *doall* loop is an *unordered* parallel loop that enforces no order among the iterations of its loop. However, a *barrier synchronization* is needed at the end of the loop.

The syntactic form of all variations of these two parallel loops is shown in Fig. 3.

There are three variations of the parallel loops: *cluster* loops, *spread* loops, and *cross-cluster* loops, denoted with prefixes C, S, X, respectively in the concurrent loop syntax shown in Fig. 3. CDOALL and CDOACROSS loops require *all* CEs in a *cluster* to join in the execution of the parallel loops. SDOALL and SDOACROSS loops cause a single CE from each cluster to join the execution of the parallel loops. It is not necessarily the best way to execute a parallel loop, but if each loop iteration has a large working set that could fill the cluster memory, such a scheduling will be very effective. Another common situation is to have a CDOALL loop nested inside an SDOALL loop. It could engage all CEs in a cluster. An XDOALL loop will require *all* CEs in *all* clusters to execute the loop body.

Local declaration of data variables in a CDOALL and XDOALL will be *visible* only to a single CE, while visible

```
{C/S/X}{DOALL/DOACROSS} index = start, end [,increment]
[local declarations of data variables]
[Preamble/Loop]
Loop Body
[Endloop/Postamble] (only SDO or XDO)

END {C/S/X}{DOALL/DOACROSS}
```

Cedar Multiprocessor. Fig. 3 Concurrent loop syntax

```
CDOACROSS j = 1, m
A(j) = B(j) + C(j)
call wait (1,1)
D(j) = E(j) + D(j-1)
call advance (1)

END DOACROSS
```

Cedar Multiprocessor. Fig. 4 An example DOACROSS loop with cascade synchronization

to *all* CEs in a *single cluster* if in an SDOALL. The statements in the *preamble* of a loop are executed *only once* by each CE when it first joins the execution of the loop and prior to the execution of loop body. The statements in the *postamble* of a loop are executed *only once* after all CEs complete the execution of the loop. Postamble is only available in SDOALL and XDOALL.

By default, data declared outside of a loop in a Cedar Fortran program are *visible* to *all* CEs in a *single* cluster. However, it provides statements to explicitly declare variables outside of a loop to be visible to *all* CEs in *all* clusters (see Fig. 5).

The GLOBAL and PROCESS COMMON statements in Fig. 5 declare that the data are visible to all CEs in all clusters. A single copy of the data exists in global memory. All CEs in all clusters could access the data, but it is the programmer's responsibility to maintain coherence if multiple copies of the data are kept in separate cluster memories. The CLUSTER and COMMON statements declare that the data are visible to all CEs inside a *single* cluster. A separate copy of the data is kept in the cluster memory of each cluster that participates in the execution of the program.

Implementation of Cedar Fortran on Cedar: All parallel loops in Cedar Fortran are self-scheduled. Iterations in a CDOALL or CDOACROSS loop are dispatched by CCB inside each cluster. CCB also provides *cascade synchronization* for a CDOACROSS loop through *wait* and *advance* calls in the loop as shown in Fig. 4. The execution of SDOALL and XDOALL are supported by the Cedar Fortran runtime library. The library starts a requested number of *helper tasks* by calling Xylem kernels in the beginning of the program execution. They remain idle until a SDOALL or XDOALL starts. The helper tasks begin to compete for loop iterations using self-scheduling.

Subroutine-level tasking is also supported in the Cedar Fortran runtime library. It allows a new thread of execution to be started for running a subroutine. In the

```
GLOBAL var [,var]
CLUSTER var [,var]
PROCESS COMMON /name/ var [,var]
COMMON /name/ var [,var]
```

Cedar Multiprocessor. Fig. 5 Variable declaration statements in Cedar Fortran

mean time, the main thread of execution continues following the subroutine call. The thread execution ends when the subroutine returns. The new thread of execution could be through one of the idle helper tasks or through the creation of a new task.

It also supports vector prefetching by generating prefetch instructions before a vector register load instruction. It could reduce the latency and overhead of loading vector data from GSM. The Cedar synchronization instructions are used primarily in the runtime library. They have been proven to be useful in controlling loop self-scheduling. They are also available to a Fortran programmer through runtime library routines.

Parallelizing Restructurer: There was a huge volume of research work on parallelizing scientific application programs written in Fortran before Cedar was built [14]. Hence, there were many sophisticated program analysis and parallelization techniques available through a very advanced parallelizing restructurer based on the *KAP restructurer* for Cedar [2, 3]. The parallelizing restructurer could convert a sequential program written in FORTRAN 77 into a parallel program in Cedar Fortran that takes advantage of all unique features in Cedar. Through this conversion and restructuring process, not only loop-level parallelism is exposed and extracted, but

also data locality is enhanced through advance techniques such as strip-mining, data globalization, and privatization [2, 4].

Performance measurements have been done extensively on Cedar [6, 7, 9]. Given the complexity of Cedar architecture, parallelizing restructurer, OS, and the programs themselves, it is very difficult to isolate the individual effects at all levels that influence the final performance of each program. The performance results shown in Fig. 6 are the most thorough measurements presented in [9]. A suite of scientific application benchmark programs called Perfect Benchmark [1] were used to measure Cedar performance. It was a collection of Fortran programs that span a wide spectrum of scientific and engineering applications from quantum chromodynamics (QCD) to analog circuit simulation (SPICE).

The table in Fig. 6 lists the performance improvement over the serial execution time of each individual program. The second column shows the performance improvement using KAP-based parallelizing restructurer. The results show that despite the most advance parallelizing restructurer at that time, the performance improvement overall is still quite limited. Hence, each benchmark program was analyzed and parallelized

Program	Complied by Kap/Cedar time (Improvement)	Auto, transforms time (Improvement)	W/o Cedar Synchronization time (% slowdown)	W/o prefetch time (% slowdown)	MFLOPS (YMP-8/Cedar)
ADM	689 (1.2)	73 (10.8)	81 (11%)	83 (2%)	6.9 (3.4)
ARC2D	218 (13.5)	141 (20.8)	141 (0%)	157 (11%)	13.1 (34.2)
BDNA	502 (1.9)	111 (8.7)	118 (6%)	122 (3%)	8.2 (18.4)
DYFESM	167 (3.9)	60 (11.0)	67 (12%)	100 (49%)	9.2 (6.5)
FLO52	100 (9.0)	63 (14.3)	64 (1%)	79 (23%)	8.7 (37.8)
MDG	3200 (1.3)	182 (22.7)	202 (11%)	202 (0%)	18.9 (1.1)
MG3D ^a	7929 (1.5)	348 (35.2)	346 (0%)	350 (1%)	31.7 (3.6)
OCEAN	2158 (1.4)	148 (19.8)	174 (18%)	187 (7%)	11.2 (7.4)
QCD	369 (1.1)	239 (1.8)	239 (0%)	246 (3%)	1.1 (11.8)
SPEC77	973 (2.4)	156 (15.2)	156 (0%)	165 (6%)	11.9 (4.8)
SPICE	95.1 (1.02)	NA	NA	NA	0.5 (11.4)
TRACK	126 (1.1)	26 (5.3)	28 (8%)	28 (0%)	3.1 (2.7)
TRFD	273 (3.2)	21 (41.1)	21 (0%)	21 (0%)	20.5 (2.8)

^aThis version of MG3D includes the elimination of file I/O.

Cedar Multiprocessor. Fig. 6 Cedar performance improvement for Perfect Benchmarks [9]

manually. The techniques are limited to those that could be implemented in an *automated* parallelizing restructurer. The third column under an *automatable transformation* shows the performance improvement that could be achieved if those programs are restructured by a more intelligent parallelizing restructurer.

It is quite clear that there was still a lot of potential in improving the performance of parallelizing restructurer because *manually* parallelized programs show substantial improvement in overall performance. However, through another decade of research in more advanced techniques of parallelizing restructurer since the publication of [2–4, 13], the consensus seems to be pointing to a direction that programmers must be given more tools and control to parallelize and write their own parallel code instead of relying totally on a parallelizing restructurer to convert a sequential version of their code automatically into a parallel form, and expect to have a performance improvement equals to that of the parallel code implemented by the programmers themselves.

The fourth column in the table of Fig. 6 shows the performance improvement using Cedar synchronization instructions. The fifth column shows the impact of vector prefetching on overall performance. The improvement from vector prefetching was not as significant as those obtained in later studies in other literatures because the Cedar backend compiler was not using sophisticated algorithms to place those prefetching instructions, and the number of prefetching buffers is too small.

Bibliography

1. Berry M et al (1989) The perfect club benchmarks: effective performance evaluation of supercomputers. *Int J Supercomput Appl* 3(3):5–40
2. Eigenmann R et al (1991) Restructuring Fortran Programs for Cedar. In: Proceedings of ICPP'91, vol 1, pp 57–66
3. Eigenmann R et al (1992) The Cedar Fortran Project. CSRD Report No. 1262, University of Illinois at Urbana-Champaign
4. Eigenmann R, Hoeflinger J, Li Z, Padua DA (1991) Experience in the automatic parallelization of four perfect-benchmark programs. In: Proceedings for the fourth workshop on languages and compilers for parallel computing, Santa Clara, CA, pp 65–83, August 1991
5. Emrath P et al (1991) The xylem operating system. In: Proceedings of ICPP'91, vol 1, pp 67–70
6. Gallivan K et al (1991) Preliminary performance analysis of the cedar multiprocessor memory system. In: Proceedings of 1991 ICPP, vol 1, pp 71–75
7. Gallivan KA, Plemmons RJ, Sameh AH (1990) Parallel algorithms for dense linear algebra computations. *SIAM Rev* 32(1):54–135
8. Guzzi MD, Padua DA, Hoeflinger JP, Lawrie DH (1990) Cedar Fortran and other vector and parallel Fortran dialects. *J Supercomput* 4(1):37–62
9. Kuck D et al (1993) The cedar system and an initial performance study. In: Proceedings of international symposium on computer architecture, San Diego, CA, pp 213–223
10. Kuck & Associates, Inc (1988) *KAP User's Guide*. Champaign Illinois
11. Lawrie DH (1975) Access and alignment of data in an array processor. *IEEE Trans Comput C-24(12)*:1145–1155, Dec 1975
12. Midkiff S, Padua DA (1987) Compiler algorithms for synchronization. *IEEE Trans C-36(12)*:1485–1495
13. Padua DA, Wolfe MJ (1986) Advanced compiler optimizations for supercomputers. *Commun ACM* 29(12):1184–1201
14. Zhu CQ, Yew PC (1987) A scheme to enforce data dependence on large on large multiprocessor system. *IEEE Trans Softw Eng SE-13(6)*:726–739, June 1987

CELL

► [Cell Broadband Engine Processor](#)

Cell Broadband Engine Processor

H. PETER HOFSTEE

IBM Austin Research Laboratory, Austin, TX, USA

Synonyms

[CELL](#); [Cell processor](#); [Cell/B.E.](#)

Definition

The Cell Broadband Engine is a processor that conforms to the Cell Broadband Engine Architecture (CBEA). The CBEA is a heterogeneous architecture defined jointly by Sony, Toshiba, and IBM that extends the Power architecture with “Memory flow control” and “Synergistic processor units.”

CBEA compliant processors are used in a variety of systems, most notably the PlayStation 3 (now PS3) from Sony Computer Entertainment, the IBM QS20, QS21, and QS22 server blades, Regza-Cell Televisions from Toshiba, single processor PCI-express accelerator boards, rackmount servers, and a variety of custom systems including the Roadrunner supercomputer at Los

Alamos National Laboratory that was the first to achieve petaflop level performance on the Linpack benchmark and was ranked as the world's #1 supercomputer from June 2008 to November 2009.

Discussion

Note: In what follows a system or chip that has multiple differing cores sharing memory is referred to as heterogeneous and a system that contains multiple differing computational elements but that does not provide shared memory between these elements is referred to as hybrid.

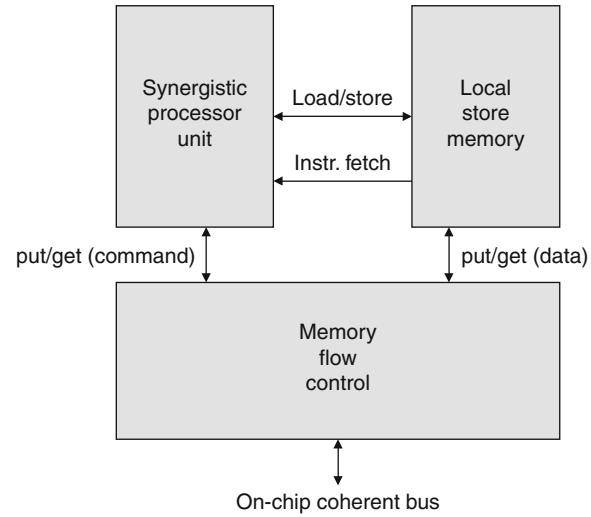
Cell Broadband Engine

The Cell Broadband Engine Architecture (CBEA) defines a heterogeneous architecture for shared-memory multi-core processors and systems. Heterogeneity allows a higher degree of efficiency and/or improved performance compared to conventional homogeneous shared-memory multi-core processors by allowing cores to gain efficiency through specialization.

The CBEA extends the Power architecture and processors that conform to the CBEA architecture are also fully Power architecture compliant. Besides the Power cores CBEA adds a new type of core: the Synergistic Processor Element (SPE). The SPE derives its efficiency and performance from the following key attributes:

- A per-SPE local store for code and data
- Asynchronous transfers between the local store and global shared memory
- A single-mode architecture
- A large register file
- A SIMD-only instruction set architecture
- Instructions to improve or avoid branches
- Mechanisms for fast communication and synchronization

Whereas a (traditional) CISC processor defines instructions that transform main memory locations directly and RISC processors transform only data in registers and therefore must first load operands into registers, the SPEs stage code and data from main memory to the local store, and the SPEs RISC core called the Synergistic Processor Unit (SPU) (Fig. 1). The SPU operates on local store the way a conventional RISC processor operates on memory, i.e., by loading data from the local store into registers before it is transformed. Similarly,



Cell Broadband Engine Processor. Fig. 1 Internal organization of the SPE

results are produced in registers, and must be staged through the local store on its way to main memory. The motivation for this organization is that while a large enough register file allows a sufficient number of operations to be simultaneously executing to effectively hide latencies to a store closely coupled to the core, a much larger store or buffer is required to effectively hide the latencies to main memory that, taking multi-core arbitration into account, can approach a thousand cycles. In all the current implementations of the SPE the local store is 256 kB which is an order of magnitude smaller than the size of a typical per-core on-chip cache for similarly performing processors. SPEs can be effective with an on-chip store that is this small because data can be packed as it is transferred to the local store and, because the SPE is organized to maximally utilize the available memory bandwidth, data in the local store is allowed to be replaced at a higher rate than is typical for a hardware cache.

In order to optimize main memory bandwidth, a sufficient number of main memory accesses (*put* and *get*) must be simultaneously executable and data access must be nonspeculative. The latter is achieved by having software issue *put* and *get* commands rather than having hardware cache pre-fetch or speculation responsible for bringing multiple sets of data on to the chip simultaneously. To allow maximum flexibility in how *put* and *get* commands are executed, without adding hardware

complexity, *put* and *get* semantics defines these operations as asynchronous to the execution of the SPU. The unit responsible for executing these commands is the Memory Flow Control unit (MFC). The MFC supports three mechanisms to issue commands.

Any unit in the system with the appropriate memory access privileges can issue an MFC command by writing to or reading from memory-mapped command registers.

The SPU can issue MFC commands by reading or writing to a set of channels that provide a direct interface to the MFC for its associated SPU.

Finally, *put-list* and *get-list* commands instruct the MFC to execute a list of *put* and *get* commands from the local store. This can be particularly effective if a substantial amount of noncontiguous data needs to be gathered into the local store or distributed back to main memory.

put and *get* commands are issued with a tag that allows groups of these commands to be associated in a tag group. Synchronization between the MFC and the SPU or the rest of the system is achieved by checking on the completion status of a tag group or set of tag groups. The SPU can avoid busy waiting by checking on the completion status with a blocking MFC channel read operation. Put and get commands adhere to the Power addressing model for main memory and specify effective addresses for main (shared) memory that are translated to real addresses according to the page and segment tables maintained by the operating system. While the local store, like the register file, is considered private, access to shared memory follows the normal Power architecture coherence rules.

The design goals of supporting both a relatively large local store and high single-thread performance imply that bringing data from the local store to a register file is a multi-cycle operation (six cycles in current SPE implementations). In order to support efficient execution for programs that randomly access data in the local store a typical loop may be unrolled four to eight times. Supporting this without creating a lot of register spills requires a large register file. Thus the SPU was architected to provide a 128-entry general-purpose register file. Further efficiency is gained by using a single register file for all data types including bit, byte, half-word and word unsigned integers, and word and double-word floating point. All of these data types are SIMD with a width that equates to 128 bits. The unified register file

allows for a rich set of operations to be encoded in a 32-bit instruction, including some performance critical operations that specify three sources and an independent target including select (conditional assignment) and multiply-add.

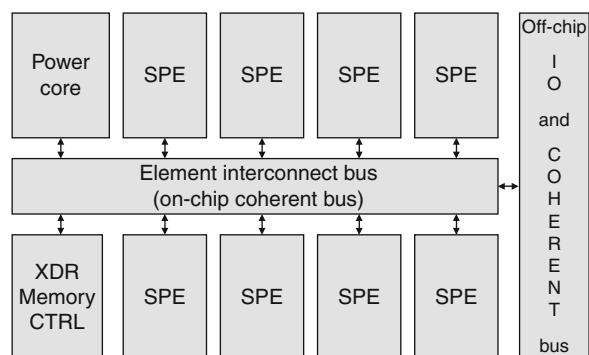
The abovementioned select instruction can quite often be used to design branchless routines. A second architectural mechanism that is provided to limit branch penalties is a branch hint instruction that provides advance notice that an upcoming branch is predicted taken and also specifies its target so that the code can be pre-fetched by the hardware and the branch penalty avoided.

The Cell Broadband Engine and PowerXCell8i processors combine eight SPEs, a Power core and high-bandwidth memory controllers, and a configurable off-chip coherence fabric onto a single chip. On-chip the cores and controllers are interconnected with a high-bandwidth coherent fabric. While physically organized as a set of ring buses for data transfers, the intent of this interconnect fabric is to allow all but those programs most finely tuned for performance to ignore its bandwidth or connectivity limitations and only consider bandwidth to memory and I/O, and bandwidth in and out of each unit (Fig. 2).

Cell B.E.-Based Systems

Cell Broadband Engine processors have been used in a wide variety of systems. Each system is reviewed briefly.

1. Sony PlayStation 3. This is perhaps the best-known use of the Cell B.E. processor. In PlayStation 3



Cell Broadband Engine Processor. Fig. 2 Organization of the Cell Broadband Engine and PowerXCell8i



the Cell B.E. processor is configured with a high-bandwidth I/O interface to connect to the RSX graphics processor. A second I/O interface connects the Cell B.E. to a SouthBridge which provides the connectivity to optical BluRay, HDD, and other storage and provides network connectivity. In the PlayStation 3 application, seven of the eight synergistic processors are used. This benefits the manufacturing efficiency of the processor.

2. Numerous PlayStation 3-based clusters. These range from virtual grids of PlayStations such as the grid used for the “Folding at Home” application that first delivered Petascale performance, to numerous clusters of PlayStations running the Linux operating system that are connected with Ethernet switches used for applications from astronomy to cryptography.
3. IBM QS20, QS21, and QS22 server blades. In these blades two Cell processors are used connected with a high-bandwidth coherent interface. Each of the Cell processors is also connected to a bridge chip that provides an interface to PCI-express, Ethernet, as well as other interfaces. The QS22 uses a version of the 65nm Cell processor with added double-precision floating-point capability that also supports larger (DDR2) memory capacities.
4. Cell-based PCI-express cards (multiple vendors). These PCI-express cards have a single Cell B.E. or PowerXCell8i processor and PCI-express bridge. The cards are intended for use as accelerators in workstations.
5. The “Roadrunner” supercomputer at Los Alamos. This supercomputer consists of a cluster of more than 3,000 Dual Dual-Core Opteron-based IBM server blades clustered together with an InfiniBand network. Each Opteron blade is PCI-express connected to two QS22 server blades, i.e., one PowerXCell8i processor per Opteron Core on each blade. This supercomputer, installed at Los Alamos in 2008, was the first to deliver a sustained Petaflop on the Linpack benchmark used to rank supercomputers (top500.org). Roadrunner was nearly three times more efficient than the next supercomputer to reach a sustained Petaflop.
6. The QPACE supercomputer developed by a European University consortium in collaboration with IBM leverages the PowerXCell8i processor in combination with an FPGA that combines the I/O

bridge and network switching functions. This configuration allows the system to achieve the very low communication latencies that are critical to Quantum Chromo Dynamics calculations. QPACE is a watercooled system, and the efficiency of water-cooling in combination with the efficiency of the PowerXCell8i processor made this system top the green500 list (green500.org) in November 2009 (Figs. 3–5).

In addition to the systems discussed above, the Cell B.E. and PowerXCell8i processors have been used in Televisions (Toshiba Regza-Cell), dual-Cell-processor 1U servers, experimental blade servers that combine FPGAs and Cell processors, and a variety of custom systems used as subsystems in medical systems and in aerospace and defense applications.

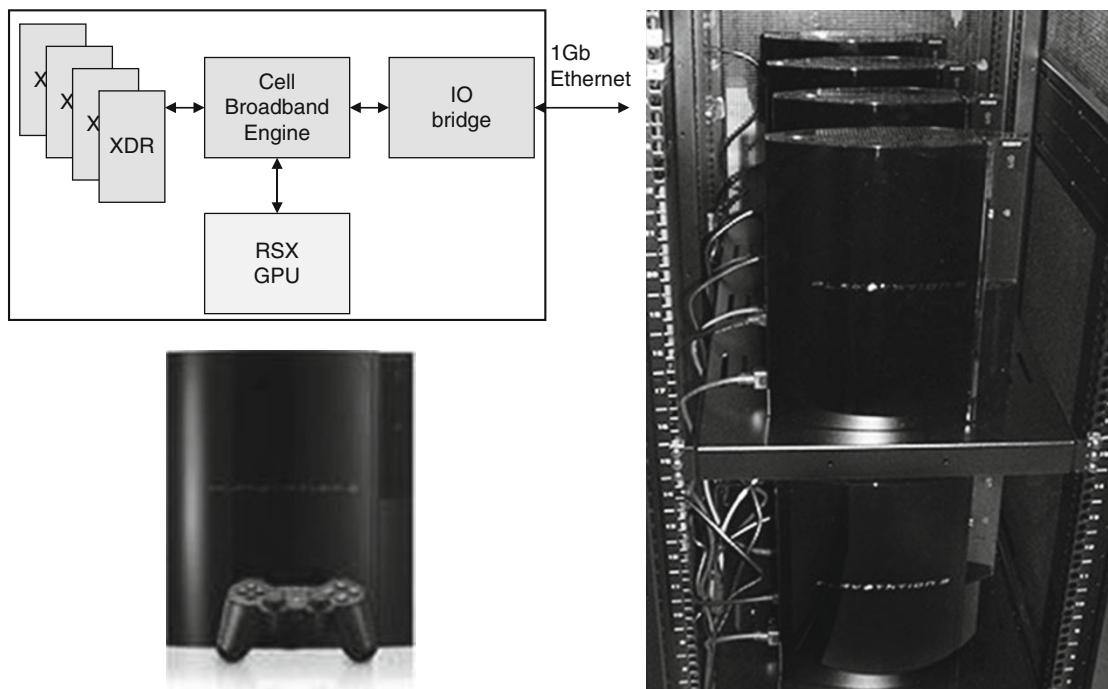
Programming Cell

The CBEA reflects the view that aspects of programs that are critical to their performance should be brought under software control for best performance and efficiency. Critical to application performance and efficiency are:

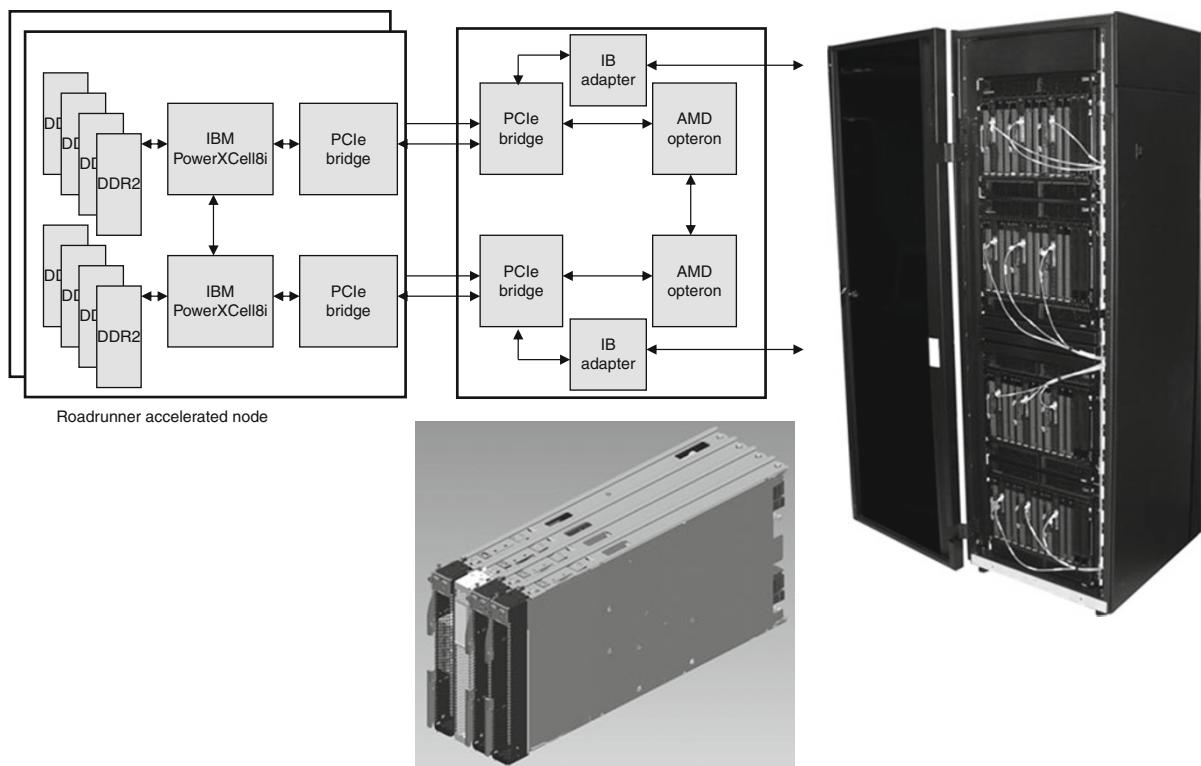
- Thread concurrency (i.e., ability to run parts of the code simultaneously on shared data)
- Data-level concurrency (i.e., ability to apply operations to multiple data simultaneously)
- Data locality and predictability (i.e., ability to predict what data will be needed next)
- Control flow predictability (i.e., ability to predict what code will be needed next)

The threading model for the Cell processor is similar to that of a conventional multi-core processor in that main memory is coherently shared between SPEs and Power cores.

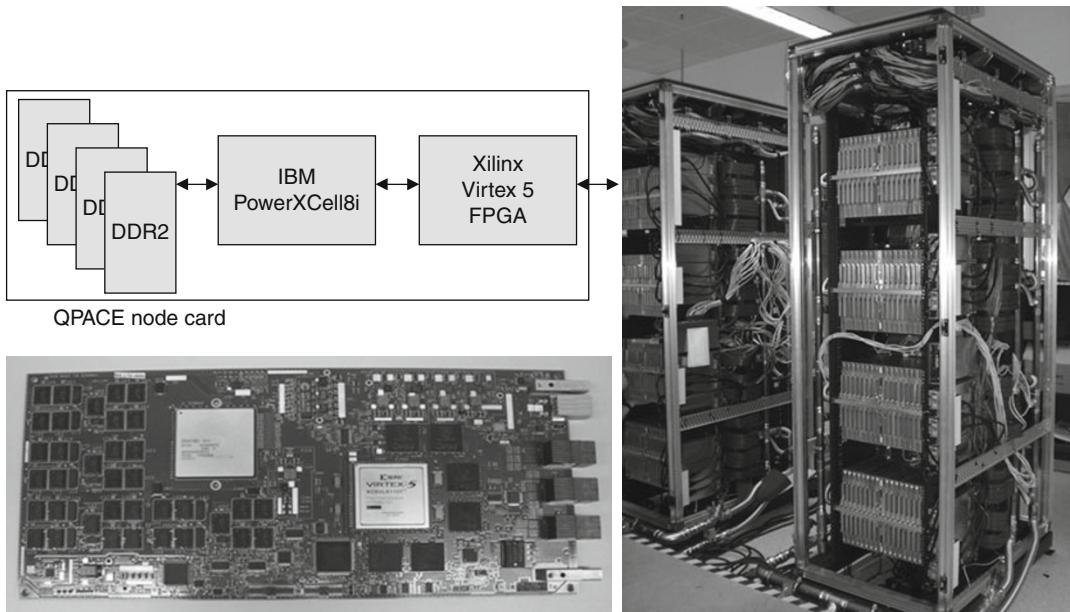
The effective addresses used by the application threads to reference shared memory are translated on the SPEs in the same way, and based on the same segment and page tables that govern translation on the Power cores. Threads can obtain locks in a consistent manner on the Power cores and on the SPEs. To this end the SPE supports atomic “get line and reserve” and “store line conditional” commands that mirror the Power core’s atomic load word and reserve and store word conditional operations. An important practical



Cell Broadband Engine Processor. Fig. 3 PlayStation 3 (configuration and system/cluster picture)



Cell Broadband Engine Processor. Fig. 4 Roadrunner (system configuration and picture)



Cell Broadband Engine Processor. Fig. 5 QPACE (card and system configuration and picture)

difference between a thread that runs on the Power core and one that runs on an SPE is that the context on the SPE is large and includes the 128 general-purpose registers, the 256 kB local store, and the state of the MFC. Therefore, unless an ABI is used that supports cooperative multitasking (i.e., switching threads only when there is minimal state in the SPE), doing a thread switch is an expensive operation. SPE threads therefore are preferentially run to completion and when a thread requires an operating system service it is generally preferable to service this with code on another processor than to interrupt and context switch the SPE. In the Linux operating system for Cell, SPE threads are therefore first created as Power threads that initialize and start the SPEs and then remain available to service operating system functions on behalf of the SPE threads they started. Note that while a context switch on an SPE is expensive, once re-initialized, the local store is back in the same state where the process was interrupted, unlike a cache which typically suffers a significant number of misses right after a thread switch. Because the local store is much larger than a typical register file and part of the state of the SPE, a (preemptive) context switch on an SPE is quite expensive. It is therefore best to think of the SPE as a single-mode or batch-mode resource. If the time to gather the inputs for a computation into the

local store leaves the SPE idle for a substantial amount of time, then double-buffering of tasks within the same user process can be an effective method to improve SPE utilization.

The SPU provides a SIMD instruction set that is similar to the instruction sets of other media-enhanced processors and therefore data-level concurrency is handled in much the same way. Languages such as OpenCL provide language support for vector data types allowing portable code to be constructed that leverages the SIMD operations. Vectorizing compilers for Cell provide an additional path toward leveraging the performance provided by the SIMD units while retaining source code portability. The use of standard libraries, the implementation of which uses the SIMD instructions provided by Cell directly, provides a third path toward leveraging the SIMD capabilities of Cell and other processors without sacrificing portability. While compilers for the Cell B.E. adhere to the language standards and thus also accept scalar data types, performance on scalar applications can suffer performance penalties for aligning the data in the SIMD registers.

The handling of data locality predictability and the use of the local store is the most distinguishing characteristic of the Cell Broadband Engine. It is possible to use the local store as a software-managed cache for code

and data with runtime support and thus remove the burden of dealing with locality from both the programmer and the compiler. While this provides a path toward code portability of standard multithreaded languages, doing so generally results in poor performance due to the overheads associated with software tag checks on loads and stores (the penalties associated with issuing the MFC commands on a cache miss are insignificant in comparison to the memory latency penalties incurred on a miss). On certain types of codes, data pre-fetching commands generated by the compiler can be effective. If the code is deeply vectorized then the local store can be treated by the compiler as essentially a large vector register file and if the vectors are sufficiently long this can lead to efficient compiler-generated pre-fetching and gathering of data. Streaming languages, where a set of kernels is applied to data that is streamed from and back to global memory can also be efficiently supported. Also, if the code is written in a functional or task-oriented style which allows operands of a piece of work to be identified prior to execution then again compiler-generated pre-fetching of data can be highly effective. Finally, languages that explicitly express data locality and/or privacy can be effectively compiled to leverage the local store. It is not uncommon for compilers or applications to employ multiple techniques, e.g., the use of a software data cache for hard to prefetch data in combination with block pre-fetching of vector data.

Because the local store stores code as well as data, software must also take care of bringing code into the local store. Unlike the software data cache, a software instruction cache generally operates efficiently and explicitly dealing with pre-fetching code to the local store can be considered a second-order optimization step for most applications. That said, there are cases, such as hard-real-time applications, where explicit control over code locality is beneficial. As noted earlier, the Cell B.E. provides branch hint instructions that allow compilers to leverage information about preferred branch directions.

Cell Processor Performance

Cell Broadband Engine processors occupy a middle ground between CPUs and GPUs. On properly structured applications, an SPU is often an order of magnitude more efficient than a high-end CPU. This can be seen by comparing the performance of the Cell B.E.

or PowerXCell8i to that of dual core processors that require a similar amount of power and chip area in the same semiconductor technology. For an application to benefit from the Cell B.E. architecture it is most important that it be possible to structure the application such that the majority of the data can be pre-fetched into the local store prior to program execution. The SIMD-width on the Cell B.E. is similar to that of CPUs of its generation and thus the degree of data parallelism is not a big distinguishing factor between Cell B.E. and CPUs. GPUs are optimized for streaming applications and with a higher degree of data parallelism can be more efficient than Cell B.E. on those applications.

Related Entries

- [NVIDIA GPU](#)
- [IBM Power Architecture](#)

Bibliographic Notes and Further Reading

An overview of Cell Broadband Engine is provided in [1] with a more detailed look at aspects of the architecture in [2] and [3]. In [4] more detail is provided on the implementation aspects of the microprocessor. Reference [5] goes into more detail on the design and programming philosophy of the Cell B.E., whereas references [6] and [7] provide insight into compiler design for the Cell B.E. Reference [8] provides an overview of the security architecture of the Cell B.E. References [9] and [10] provide an introduction to performance attributes of the Cell B.E.

Bibliography

1. Kahle JA, Day MN, Hofstee HP, Johns CR, Maeurer, TR, Shippy D (2005) Introduction to the cell multiprocessor. *IBM J Res Dev* 49(4/5):589–604
2. Johns CR, Brokenshire DA (2007) Introduction to the Cell Broadband Engine architecture. *IBM J Res Dev* 51(5):503–520
3. Gschwind M, Hofstee HP, Flachs B, Hopkins M, Watanabe Y, Yamazaki T (2006) Synergistic processing in cell's multicore architecture. *IEEE Micro* 26(2):10–24
4. Flachs B, Asano S, Dhong SH, Hofstee HP, Gervais G, Kim R, Le T, Liu P, Leenstra J, Liberty JS, Michael B, Oh H-J, Mueller SM, Takahashi O, Hirairi K, Kawasumi A, Murakami H, Noro H, Onishi S, Pille J, Silberman J, Yong S, Hatakeyama A, Watanabe Y, Yano N, Brokenshire DA, Peyravian M, To V, Iwata E (2007) Microarchitecture and implementation of the synergistic processor in 65-nm and 90-nm SOI. *IBM J Res Dev* 51(5):529–544
5. Keckler SW, Olokuton K, Hofstee HP (eds) (2009) Multicore processors and systems. Springer, New York

6. Eichenberger AE, O'Brien K, O'Brien KM, Wu P, Chen T, Oden PH, Prener DA, Sheperd JC, So B, Sura Z, Wang A, Zhang T, Zhao P, Gschwind M, Achambault R, Gao Y, Koo R (2006) Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture. *IBM Syst J* 45(1):59–84
7. Perez JM, Bellens P, Badia RM, Labarta J (2007) CellSs: making it easier to program the Cell Broadband Engine processor. *IBM J Res Dev* 51(5):593–604
8. Shimizu K, Hofstee HP, Liberty JS (2007) Cell Broadband Engine processor vault security architecture, *IBM J Res Dev* 51(5):521–528
9. Chen T, Raghavan R, Dale JN, Iwata E (2007) Cell Broadband Engine architecture and its first implementation – a performance view. *IBM J Res Dev* 51(5):559–572
10. Williams S, Shalf J, Oliker L, Husbands P, Kamil S, Yelick K (2006) The potential of the cell processor for scientific computing. In: Proceedings of the third conference on computing frontiers, Ischia, pp 9–20

Cell Processor

► [Cell Broadband Engine Processor](#)

Cell/B.E.

► [Cell Broadband Engine Processor](#)

Cellular Automata

MATTHEW SOTTILE
Galois, Inc., Portland, OR, USA

Definition

A cellular automaton is a computational system defined as a collection of cells that change state in parallel based on a transition rule applied to each cell and a finite number of their neighbors. A cellular automaton can be treated as a graph in which vertices correspond to cells and edges connect adjacent cells to define their neighborhood. Cellular automata are well suited to parallel implementation using a Single Program, Multiple Data (SPMD) model of computation. They have historically been used to study a variety of problems in the biological, physical, and computing sciences.

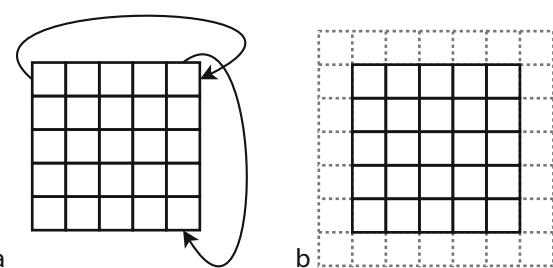
Discussion

Definition of Cellular Automata

Cellular automata (CA) are a class of highly parallel computational systems based on a transition function applied to elements of a grid of cells. They were first introduced in the 1940s by John von Neumann as an effort to model self-reproducing biological systems in a framework based on mathematical logic. The system is evolved in time by applying this transition function to all cells in parallel to generate the state of the system at time t based on the state from time $t-1$. Three properties are common amongst different CA algorithms:

1. A grid of cells is defined such that for each cell there exists a finite neighborhood of cells that influence its state change. This neighborhood is frequently defined to be other cells that are spatially adjacent. The topology of the grid of cells determines the neighbors of each cell.
2. A fixed set of state values are possible for each cell. These can be as simple as boolean true/false values, all the way up to real numbers. Most common examples are restricted to either booleans or a small set of integer values.
3. A state transition rule that evolves the state of a cell to a new state based on its current value and that of its neighbors.

Cells are positioned at the nodes of a regular grid, commonly based on rectangular, triangular, or hexagonal cells. The grid of cells may be finite or unbounded in size. When finite grids are employed, the model must account for the boundaries of the grid (Fig. 1). The most common approaches taken are to either impose a toroidal or cylindrical topology to the space in which all or some edges wrap around, or adopt a fixed value for all



Cellular Automata. Fig. 1 Common boundary types.
(a) Toroidal, (b) planar with fixed values off grid

off-grid cells to represent an appropriate boundary condition. Finite grids are easiest to model computationally, as unbounded grids will require a potentially large and endlessly growing amount of memory to represent all cells containing important state values.

1D Grids

In a simple 1D CA, the grid is represented as an array of cells. The neighborhood of a cell is the set of cells that are spatially adjacent within some finite radius. For example, in Fig. 2 a single cell is highlighted with a neighborhood of radius 1, which includes one cell on each side.

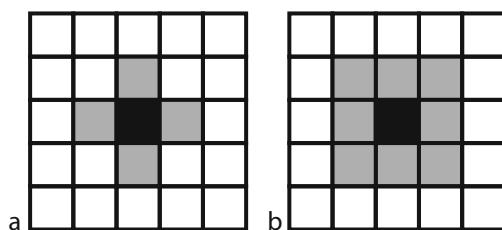
2D Grids

In a 2D CA, there are different ways in which a neighborhood can be defined. The most common are the von Neumann and Moore neighborhoods as shown in Fig. 3. For each cell, the eight neighbor cells are considered due to either sharing a common face or corner. The von Neumann neighborhood includes only those cells that share a face, while the Moore neighborhood also includes those sharing a corner.

When considering 2D systems, the transition rule computation bears a strong similarity to stencil-based computations common in programs that solve systems of partial differential equations. For example, a simple stencil computation involves averaging all values in the neighborhood of each point in a rectangular grid and replacing the value of the point at the center with the result. In the example of Conway's game of life introduced later in this entry, there is a similar computation for the transition rule, in which the number of cells that



Cellular Automata. Fig. 2 1D neighborhood of radius 1



Cellular Automata. Fig. 3 2D neighborhoods. (a) von Neumann (b) Moore

are in the “on” state in the neighborhood of a cell is used to determine the state of the central cell. In both cases there is a neighborhood of cells where a reduction operation (such as +) is applied to their state to compute a single value from which the new state of the central cell is computed.

Common Cellular Automata

1D Boolean Automata

The simplest cellular automata are one-dimensional boolean systems. In these, the grid is an array of boolean values. The smallest neighborhood is that in which a cell is influenced only by its two adjacent neighbors. The state transition rule for a cell c_i is defined as

$$c_i^{t+1} = R(c_i^t, c_{i-1}^t, c_{i+1}^t).$$

Wolfram Rule Scheme

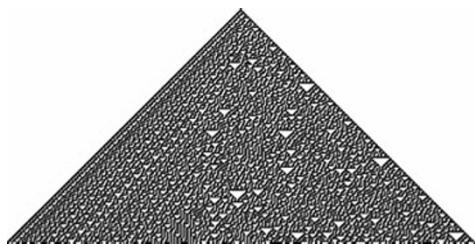
Wolfram [12] defined a concise scheme for naming 1D cellular automata that is based on a binary encoding of the output value for a rule over a family of possible inputs. The most common scheme is applied to systems in which a cell is updated based only on the values of itself and its two nearest neighbors. For any cell there are only eight possible inputs to the transition rule R : 000, 001, 010, 011, 100, 101, 110, and 111. For each of these inputs, R produces a single bit. Therefore, if each 3-bit input is assigned a position in an 8-bit number and set that bit to either 0 or 1 based on the output of R for the corresponding 3-bit input state, an 8-bit number can be created that uniquely identifies the rule R .

For example, a common example rule is number 30, which in binary is written as 00011110. This means that the input states 000, 101, 110, and 111 map to 0, and the inputs 001, 010, 011, and 100 map to 1. It is common to see these rules represented visually as in Fig. 4. The result of this rule is shown in Fig. 5.

This numbering scheme can be extended to 1D cellular automata with larger neighborhoods. For example, a CA with a neighborhood of five cells (two on each side of the central cell) would have $2^5 = 32$ possible input states; so each rule could be summarized with a single



Cellular Automata. Fig. 4 Rule 30 transition rule. Black is 1, white is 0



Cellular Automata. Fig. 5 A sequence of time steps for the 1D rule 30 automaton, starting from the top row with a single cell turned on



Cellular Automata. Fig. 6 A three color 1D automaton representing rule 1635 with initial conditions 101

32-bit number. The 1D scheme can also be extended to include cells that have more than two possible states. Extensions to these larger state spaces yield example rules that result in very complex dynamics as they evolve. For example, rule 1635 in Fig. 6 shows the evolution of the system starting from an initial condition of 101 centered on the first row.

2D Boolean Automata

In 1970, Martin Gardner introduced a cellular automaton invented by John Conway [8] on 2D grids with boolean-valued cells that is known as the *game of life*. For each cell, the game of life transition rule considers all cells in the 8-cell Moore neighborhood. Cells that contain a true value are considered to be “alive,” while those that are false are “dead.” The rule is easily stated as follows, where c_t is the current state of the cell at time step t , and N is the number of cells in its neighborhood that are alive:

- If c_t is alive:
 - If $N < 2$ then c_{t+1} is set to dead.
 - If $N > 3$ then c_{t+1} is set to dead.
 - If $N = 2$ or $N = 3$ then c_{t+1} is set to alive.
- If c_t is dead and $N = 3$, c_{t+1} is set to alive.

This rule yields very interesting dynamic behavior as it evolves. Many different types of structures have been discovered, ranging from those that are static and stable, those that repeat through a fixed sequence of states, and persistent structures that move and produce new structures. Figure 7a shows a typical configuration of the space after a number of time steps from a random initial state. A number of interesting structures that appear commonly are also shown, such as the self-propagating glider (Fig. 7b), oscillating blinker (Fig. 7d), and a configuration that rapidly stabilizes and ceases to change (Fig. 7c).

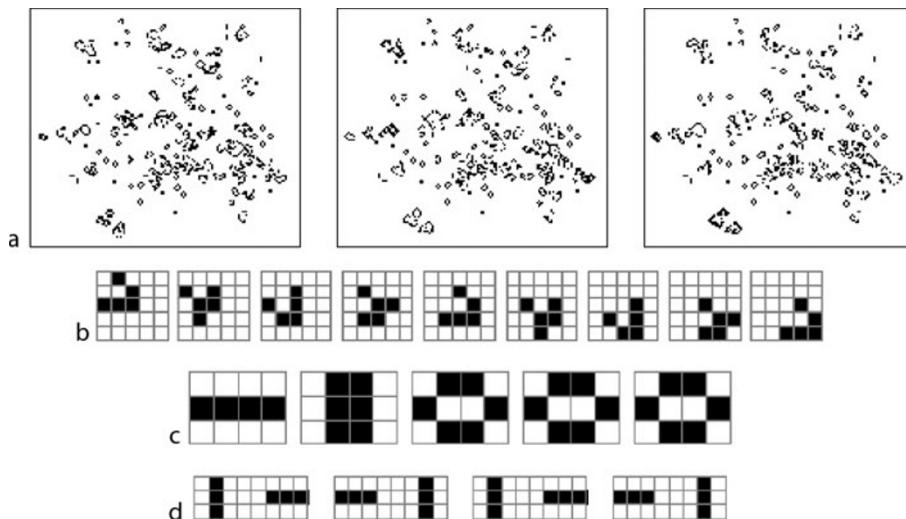
Lattice Gas Automata

Cellular automata have a history of being employed for modeling physical systems in the context of statistical mechanics, where the macroscopic behavior of the system is dictated primarily by microscopic interactions with a small spatial extent. The Ising model used in solid state physics is an example of non-automata systems that share this characteristic of local dynamics leading to a global behavior that mimics real physical systems.

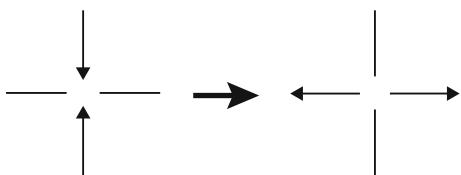
In the 1970s, cellular automata became a topic of interest in the physics community for modeling fluid systems based on the observation that macroscopic behavior of fluids is determined by the microscopic behavior of a large ensemble of fluid particles. Instead of considering cells as containing generic boolean values, one could encode more information in each cell to represent entities such as interacting particles. Two important foundational cellular automata systems are described here that laid the basis for later models of modern relevance such as the Lattice Boltzmann Method (LBM). The suitability of CA-based algorithms to parallelization is a primary reason for current interest in LBM methods for physical simulation.

HPP Lattice Gas

One of the earliest systems that gained attention was the HPP automaton of Hardy, Pomeau and de Pazzis [9]. In the simple single fluid version of this system, each cell contains a 4-bit value. Each bit in this value corresponds to a vector relating the cell to its immediately adjacent neighbors (up, down, left, and right). A bit being on corresponds to a particle coming from the neighbor into the cell, and a bit being off corresponds to the absence of a particle arriving from the neighbor.



Cellular Automata. Fig. 7 Plots showing behavior observed while evolving the game of life CA. (a) Game of life over three time steps; (b) two full periods of the glider; (c) feature that stabilizes after three steps; (d) two features that repeat every two steps



Cellular Automata. Fig. 8 A sample HPP collision rule

The transition rule for the HPP automaton was constructed to represent conservation of momentum in a physical system. For example, if a cell had a particle arriving from above and from the left, but none from the right or from below, then after a time step the cell should produce a particle that leaves to the right and to the bottom. Similarly, if a particle arrives from the left and from the right, but from neither the top or bottom, then two particles should exit from the top and bottom. In both of these cases, the momentum of the system does not change. This simple rule models the physical properties of a system based on simple collision rules. This is illustrated in Fig. 8 for a collision of two particles entering a cell facing each other. If the cell encoded the entry state as the binary value 1010 where the bits correspond in order to North, East, South, and West, then the exit state for this diagram would be defined as 0101.

The rules for the system are simple to generate by taking advantage of rotation invariant properties of the system – the conservation laws underlying a collision

rule do not change if they are transformed by a simple rotation. For example, in the cell illustrated here, rotation of the state by 90° corresponds to a circular shift of the input and output bit encodings by one. As such, a small set of rules need to be explicitly defined and the remainder can be computed by applying rotations. The same shortcut can be applied to generating the table of transition rules for more sophisticated lattice gas systems due to the rotational invariance of the underlying physics of the system.

From a computational perspective, this type of rule set was appealing for digital implementation because it could easily be encoded purely based on boolean operators. This differed from traditional methods for studying fluid systems in which systems of partial differential equations needed to be solved using floating point arithmetic.

FHP Rules

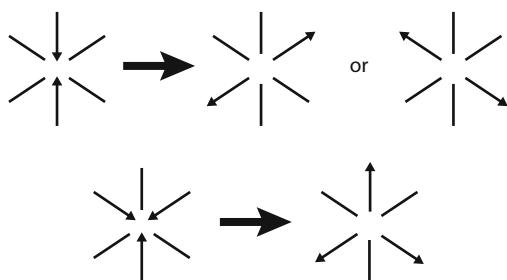
The HPP system, while intuitively appealing, lacks properties that are necessary for modeling realistic physical systems – for example, the HPP system does not exhibit isotropic behavior. The details related to the physical basis of these algorithms are beyond the scope of this entry, but are discussed at length by Doolen [5], Wolf-Gladrow [4], and Rivet [10]. A CA inspired by the early HPP automaton was created by Frisch, Hasslacher,

and Pomeau [7], leading to the family of FHP lattice gas methods.

The primary advance of these new methods made over prior lattice gas techniques was that it was possible to derive a rigorous relationship between the FHP cellular automaton and more common models for fluid flow based on Navier–Stokes methods requiring the solution of systems of partial differential equations. Refinements of the FHP model, and subsequent models based on it, are largely focused on improving the correspondence of the CA-based algorithms to the physical systems that they model and better match accepted traditional numerical methods.

In the 2D FHP automaton, instead of a neighborhood based on four neighbors, the system is defined using a neighborhood of six cells arranged in a hexagonal grid with triangular cells. This change in the underlying topology of the grid of cells was critical to improving the physical accuracy of the system while maintaining a computationally simple CA-based model. The exact same logic is applied in constructing the transition rules in this case as for the HPP case – the rules must obey conservation laws. The first instance of the automaton was based on this simple extension of HPP, in which the rules for transitions were intended to model collisions in which the state represented only particles arriving at a cell. Two example rules are shown in Fig. 9, one that has two possible outcomes (each with equal probability), and one that is deterministic. In both cases, the starting state indicates the particles entering a cell, and the exit state(s) show the particles leaving the cell after colliding.

Later revisions of the model added a bit to the state of each cell corresponding to a “rest particle” – a particle that, absent any additional particle arriving from



Cellular Automata. Fig. 9 Two examples of FHP collision rules

any neighbor, remained in the same position. Multiple variants of the FHP-based rule set appeared in the literature during the 1980s and 1990s. These introduced richer rule sets with more collision rules, with the effect of producing models that corresponded to different viscosity coefficients [4]. Further developments added the ability to model multiple fluids interacting, additional forces influencing the flow, and so on.

Modeling realistic fluid systems required models to expand to three dimensions. To produce a model with correct isotropy, d'Humières, Lallemand and Frisch [3] employed a four-dimensional structure known as the *face centered hyper-cubic*, or FCHC, lattice. The three-dimensional lattice necessary for building a lattice gas is based on a dimension reducing procedure that projects FCHC into three-dimensional space. Given this embedded lattice, a set of collision rules that obey the appropriate physical conservation laws can be defined much like those for FHP in two dimensions.

Lattice gas methods were successful in demonstrating that, with careful selection of transition rules, realistic fluid behavior could be modeled with cellular automata. The lattice Boltzmann method (LBM) is a derivative of these early CA-based fluid models that remains in use for physical modeling problems today. In the LBM, a similar grid of cells is employed, but the advancement of their state is based on continuous valued function evaluation instead of a boolean state machine.

Self-organized Criticality: Forest Fire Model

In 1992, a model was proposed by Drossel and Schwabl [6] to model forest fire behavior that built upon a previous model proposed by Bak, Chen, and Tang [1]. This model is one of a number of similar systems that are used to study questions in statistical physics, such as phase transitions and their critical points. A related system is the sandpile model, in which a model is constructed of a growing pile of sand that periodically experiences avalanches of different sizes.

As a cellular automaton, the forest fire model is interesting because it is an automaton in which the rules for updating a cell are probabilistic. The lattice gas models above include rules that are also probabilistic, but unlike the forest fire system, the lattice gas probabilities are fixed to ensure accurate correspondence with the physical system being modeled. Forest fire parameters are intended to be changed to study the response

of the overall system to their variation. Two probability parameters are required: p and f , both of which are real numbers between 0 and 1. The update rule for a cell is defined as:

- A cell that is burning becomes empty (burned out).
- A cell will start burning if one or more neighbor cells are burning.
- A cell will ignite with probability f regardless of how many neighbors are burning.
- A cell that is empty turns into a non-burning cell with probability p .

The third and fourth rules are the probabilistic parts of the update algorithm. The third rule states that cells can spontaneously combust without requiring burning neighbors to ignite them. The fourth rule states that a burned region will eventually grow back as a healthy, non-burning cell. All four rules correspond to an intuitive notion for how a real forest functions. Trees can be ignited by their neighbors; events can cause trees to start burning in a non-burning region (such as by lightning or humans), and over time, burned regions will eventually grow back.

This model differs from basic cellular automata due to the requirement that for each cell update there may be a required sampling of a pseudorandom number source to determine whether or not spontaneous ignition or new tree growth occurs. This has implications for parallel implementation because it requires the ability to generate pseudorandom numbers in a parallel setting.

Universality in Cellular Automata

Cellular automata are not only useful for modeling physical phenomena. They have also been used to study computability theory. It has been shown that cellular automata exist that exhibit the property of computational universality. This is a concept that arises in computability theory and is based on the notion of Turing-completeness. Simply stated, a system that exhibits Turing-completeness is able to perform any calculation by following a simple set of rules on input data. In essence, given a carefully constructed input, the execution of the automaton will perform a computation (such as arithmetic) that has been “programmed into” the input data. This may be accomplished by finding an encoding of an existing universal computing system

within the system which is to be shown as capable of universal computation.

For example, the simple one-dimensional Rule 110 automaton was shown by Cook [2] to be capable of universal computation. He accomplished this by identifying structures known as “spaceships” that are self-perpetuating and could be configured to interact. These interactions could be used then to encode an existing computational system in the evolving state of the Rule 110 system. The input data and program to execute would then be encoded in a linear form to serve as the initial state of the automaton. A similar approach was taken to show that the game of life could also emulate any Turing machine or other universal computing system. Using self-perpetuating structures, their generators, and structures that react with them, one can construct traditional logic gates and connect them together to form structures equivalent to a traditional digital computer.

As Cook points out, a consequence of this is that it is formally undecidable to predict certain behaviors, such as reaching periodic states or a specific configuration of bits. The property of universality does not imply that encoding of a system like a digital computer inside a CA would be at all efficient – the CA implementation would be very slow relative to a real digital computer.

Parallel Implementation

Cellular automata fit well with parallel implementations due to the high degree of parallelism present in the update rules that define them. Each cell is updated in parallel with all others, and the evolution of the system proceeds by repeatedly applying the rule to update the entire set of cells.

The primary consideration for parallel implementation of a cellular automaton is the dependencies imposed by the update rule. Parallel implementation of any sequential algorithm often starts with an analysis of the dependencies within the program, both in terms of control and data. In a CA implementation, there exists a data dependency between time steps due to the need for state data from time step t to generate the state for time $t + 1$.

For this discussion, consider the simple 1D boolean cellular automaton. To determine the value of the i th cell at step t of the evolution of the system, the update rule requires the value of the cell itself and its neighbors ($i - 1$

and $i + 1$) at step $t - 1$. This dependency has two effects that determine how the algorithm can be implemented in parallel.

1. Given that the previous step $t - 1$ is not changed at all and can be treated as read-only, all cell values for step t can be updated in parallel.
2. An in-place update of the cells is not correct, as any given cell i from step $t - 1$ is in the dependency set of multiple cells. If an in-place update occurred, then it is possible that the value from step $t - 1$ would be destructively overwritten and lost, disrupting the update for any other cell that requires its step $t - 1$ value. A simple double-buffering scheme can be used to overcome this issue for parallel implementations.

In more sophisticated systems such as those discussed earlier, the number of dependencies that each cell has between time steps grows with the size of its neighborhood. The game of life rule states that each cell requires the state of nine cells to advance – the state of the cell itself and that of its eight neighbors. The 3D FCHC lattice gas has 24 channels per node to influence the transition rule leading to a large number of dependencies per cell.

Hardware Suitability

Historically cellular automata were of interest on massively parallel processing (MPP) systems in which a very large number of simple processing elements were available. Each processing element was assigned a small region of the set of cells (often a spatially contiguous patch), and iterated over the elements that it was responsible for to apply the update rule. A limited amount of synchronization would be required to ensure that each processing element worked on the same update step.

In the 1990s, there was a research effort at MIT to build the CAM8 [11], a parallel architecture designed specifically for executing cellular automata. Machines specialized for CA models could achieve performance comparable to conventional parallel systems of the era, including the Thinking Machines CM-2 and Cray X-MP. The notable architectural features of the CAM8 were the processing nodes based on DRAM for storing cell state data, SRAM for storing a lookup table holding the transition rules for a CA, and a mesh-based

interconnection network connecting processing nodes together for exchanging data. The machine could be programmed for different CA rules, and was demonstrated on examples from fluid dynamics (lattice gases), statistical physics (diffusion limited aggregation), image processing, and large-scale logic simulation.

Another approach that has been taken to hardware implementation of CAs is the use of Field Programmable Gate Arrays (FPGAs) to encode the boolean transition rule logic directly in hardware. More recently, hardware present in accelerator-based systems such as General Purpose Graphics Processing Units (GPGPU) presents a similar feature set as the traditional MPPs. These devices present the programmer with the ability to execute a large number of small parallel threads that operate on large volumes of data. Each thread in a GPU is very similar to the basic processing elements from a traditional MPP. The appeal of these modern accelerators is that they can achieve performance comparable to traditional supercomputers on small, specialized programs like cellular automata that are based on a single small computation run in parallel on a large data set.

References

1. Bak P, Chen K, Tang C (1990) A forest fire model and some thoughts on turbulence. *Phys Lett A* 147:297–300
2. Cook M (2004) Universality in elementary cellular automata. *Complex Syst* 15(1):1–40
3. d'Humieres D, Lallemand P, Frisch U (1986) Lattice gas models for 3-D hydrodynamics. *Europhys Lett* 2:291–297
4. Wolf-Gladrow DA (2000) Lattice-gas cellular automata and lattice Boltzmann models: an introduction, volume 1725 of Lecture Notes in Mathematics. Springer, Berlin
5. Doolen GD (ed) (1991) Lattice gas methods: theory, application, and Hardware. MIT Press, Cambridge, MA
6. Drossel B, Schwabl F (1992) Self-organized critical forest-fire model. *Phys Rev Lett* 69(11):1629–1632
7. Frisch U, Hasslacher B, Pomeau Y (1986) Lattice-gas automata for the Navier-Stokes equation. *Phys Rev Lett* 56(14):1505–1508
8. Gardner M (1970) The fantastic combinations of John Conway's new solitaire game "life". *Sci Am* 223:120–123
9. Hardy J, Pomeau Y, de Pazzis O (1973) Time evolution of a two-dimensional model system. *J Math Phys* 14(12):1746–1759
10. Rivet J-P, Boon JP (2001) Lattice gas hydrodynamics, volume 11 of Cambridge Nonlinear Science Series. Cambridge University Press, Cambridge
11. Toffoli T, Margolus N (1991) Programmable matter: concepts and realization. *Phys D* 47:263–272
12. Wolfram S (1983) Statistical mechanics of cellular automata. *Rev Mod Phys* 55:601–644

Chaco

BRUCE HENDRICKSON
Sandia National Laboratories, Albuquerque, NM, USA

Definition

Chaco was the first modern graph partitioning code developed for parallel computing applications. Although developed in the early 1990s, the code continues to be widely used today.

Discussion

Chaco is a software package that implements a variety of graph partitioning techniques to support parallel applications. Graph partitioning is a widely used abstraction for dividing work amongst the processors of a parallel machine in such a way that each processor has about the same amount of work to do, but the amount of interprocessor communication is kept small. Chaco is a serial code, intended to be used as a preprocessing step to set up a parallel application. Chaco takes in a graph which describes the data dependencies in a computation and outputs a description of how the computation should be partitioned amongst the processors of a parallel machine.

Chaco was developed by Bruce Hendrickson and Rob Leland at Sandia National Laboratories. It provides implementations of a variety of graph partitioning algorithms. Chaco provided an advance over the prior state of the art in a number of areas [4].

- Although multilevel partitioning was simultaneously co-invented by several groups [2, 3, 5], the implementation in Chaco led to the embrace of this approach as the best balance between speed and quality for many practical problems. In parallel computing, this remains the dominant approach to partitioning problems.
- All the algorithms in Chaco support graphs with weights on both edges and vertices.
- Chaco provides a suite of partitioning algorithms including spectral, geometric and multilevel approaches.
- Chaco supports the coupling of global methods (e.g., spectral or geometric partitioning) with local

refinement provided by an implementation of the Fiduccia-Mattheyses (FM) implementation.

- Chaco provides a robust yet efficient algorithm for computing Laplacian eigenvectors which can be used for spectral partitioning or for other algorithms.
- Chaco supports generalized spectral, combinatorial, and multilevel algorithms that partition into more than two parts at each level of recursion.
- Chaco has several approaches that consider the topology of the target parallel computer during the partitioning process.

Related Entries

- ▶ [Load Balancing, Distributed Memory](#)
- ▶ [Graph Partitioning](#)
- ▶ [Hypergraph Partitioning](#)
- ▶ [METIS and ParMETIS](#)

Bibliographic Notes and Further Reading

Chaco is available for download under an open source license [1]. The Chaco Users Guide [4] has much more detailed information about the capabilities of the code. Subsequent partitioning codes like METIS, Jostle, PATOH, and Scotch have adopted and further refined many of the ideas first prototyped in Chaco.

Acknowledgment

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the US Department of Energy under contract DE-AC04-94AL85000.

Bibliography

1. Chaco: Software for partitioning graphs, <http://www.sandia.gov/~bahendr/chaco.html>
2. Bui T, Jones C (1993) A heuristic for reducing fill in sparse matrix factorization. In: Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing, SIAM, Portland, OR, pp 445–452
3. Cong J, Smith ML (1993) A parallel bottom-up clustering algorithm with applications to circuit partitioning in VLSI design. In: Proceedings 30th Annual ACM/IEEE International Design Automation Conference, DAC '93, ACM, Dallas, TX, pp 755–760
4. Hendrickson B, Leland R (1994) The Chaco user's guide, version 2.0. Technical Report SAND94-2692, Sandia National Laboratories, Albuquerque, NM, October 1994

5. Hendrickson B, Leland R (1995) A multilevel algorithm for partitioning graphs. In: Proceedings of the Supercomputing '95. ACM, December 1995. Previous version published as Sandia Technical Report SAND 93-1301

Chapel (Cray Inc. HPCS Language)

BRADFORD L. CHAMBERLAIN
Cray Inc., Seattle, WA, USA

Synonyms

Cascade high productivity language

Definition

Chapel is a parallel programming language that emerged from Cray Inc.'s participation in the High Productivity Computing Systems (HPCS) program sponsored by the Defense Advanced Research Projects Agency (DARPA). The name Chapel derives from the phrase "Cascade High Productivity Language," where "Cascade" is the project name for the Cray HPCS effort. The HPCS program was launched with the goal of raising user productivity on large-scale parallel systems by a factor of ten. Chapel was designed to help with this challenge by vastly improving the programmability of parallel architectures while matching or beating the performance, portability, and robustness of previous parallel programming models.

Discussion

History

The Chapel language got its start in 2002 during the first phase of Cray Inc.'s participation in the DARPA HPCS program. While exploring candidate system design concepts to improve user productivity, the technical leaders of the Cray Cascade project decided that one component of their software solution would be to pursue the development of an innovative parallel programming language. Cray first reported their interest in pursuing a new language to the HPCS mission partners in January 2003. The language was named Chapel later that year, stemming loosely from the phrase "Cascade High Productivity Language." The early phases of Chapel development were a joint effort between Cray Inc. and their

academic partners at Caltech/JPL (Jet Propulsion Laboratory).

The second phase of the HPCS program, from summer 2003 through 2006, saw a great deal of work in the specification and early implementation of Chapel. Chapel's initial design was spearheaded by David Callahan, Brad Chamberlain, and Hans Zima. This group published an early description of their design in a paper entitled "The Cascade High Productivity Language" [1]. Implementation of a Chapel compiler began in the winter of 2003, initially led by Brad Chamberlain and John Plevyak (who also played an important role in the language's early design). In late 2004, a rough draft of the language specification was completed. Around this same time, Steve Deitz joined the implementation effort, who would go on to become one of Chapel's most influential long-term contributors. This group established the primary themes and concepts that set the overall direction for the Chapel language. The motivation and vision of Chapel during this time was captured in an article entitled "Parallel Programmability and the Chapel Language" [4], which provided a wish list of sorts for productive parallel languages and evaluated how well or poorly existing languages and Chapel met the criteria.

By the summer of 2006, the HPCS program was transitioning into its third phase and the Chapel team's composition had changed dramatically, as several of the original members moved on to other pursuits and several new contributors joined the project. Of the original core team, only Brad Chamberlain and Steve Deitz remained and they would go on to lead the design and implementation of Chapel for the majority of phase III of HPCS (ongoing at the time of this writing).

The transition to phase III also marked a point when the implementation began to gain significant traction due to some important changes that were made to the language and compiler design based on experience gained during phase II. In the spring of 2006, the first multi-threaded task-parallel programs began running on a single node. By the summer of 2007, the first task-parallel programs were running across multiple nodes with distributed memory. And by the fall of 2008, the first multi-node data-parallel programs were being demonstrated.

During this time period, releases of Chapel also began taking place, approximately every 6 months. The



very first release was made available in December 2006 on a request-only basis. The first release to the general public occurred in November 2008. And in April 2009, the Chapel project more officially became an open-source project by migrating the hosting of its code repository to SourceForge.

Believing that a language cannot thrive and become adopted if it is controlled by a single company, the Chapel team has often stated its belief that as Chapel grows in its capabilities and popularity, it should gradually be turned over to the broader community as an open, consortium-driven language. Over time, an increasing number of external collaborations have been established with members of academia, computing centers, and industry, representing a step in this direction.

At the time of this writing, Chapel remains an active and evolving project. The team's current emphasis is on expanding Chapel's support for user-defined distributions, improving performance of key idioms, supporting users, and seeking out strategic collaborations.

Influences

Rather than extending an existing language, Chapel was designed from first principles. It was decided that starting from scratch was important to avoid inheriting features from previous languages that were not well suited to large-scale parallel programming. Examples include pointer/array equivalence in C and common blocks in Fortran. Moreover, Chapel's design team believed that the challenging part of learning any language is learning its semantics, not its syntax. To that end, embedding new semantics in an established syntax can often cause more confusion than benefit. To this end, Chapel was designed from a blank slate. That said, Chapel's design does contain many concepts and influences from previous languages, most notably ZPL, High Performance Fortran (HPF), and the Tera/Cray MTA extensions to C and Fortran, reflecting the backgrounds of the original design team. Chapel utilizes a partitioned global namespace for convenience and scalability, similar to traditional PGAS languages like UPC, Co-Array Fortran, and Titanium; yet it departs from those languages in other ways, most notably by supporting more dynamic models of execution and parallelism. Other notable influences include CLU, ML, NESL, Java, C#, C/C++, Fortran, Modula, and Ada.

Themes

Chapel's design is typically described as being motivated by five major themes: (1) support for general parallel programming, (2) support for global-view abstractions, (3) a multiresolution language design, (4) support for programmer control over locality and affinity, and (5) a narrowing of the gap between mainstream and HPC programming models. This section provides an overview of these themes.

General Parallel Programming

Chapel's desire to support general programming stems from the observation that while programs and parallel architectures both typically contain many types of parallelism at several levels, programmers typically need to use a mix of distinct programming models to express all levels/types of software parallelism and to target all available varieties of hardware parallelism. In contrast, Chapel aims to support multiple levels of hardware and software parallelism using a unified set of concepts for expressing parallelism and locality. To this end, Chapel programs support parallelism at the function, loop, statement, and expression levels. Chapel language concepts support data parallelism, task parallelism, concurrent programming, and the ability to compose these different styles within a single program naturally. Chapel programs can be executed on desktop multicore computers, commodity clusters, and large-scale systems developed by Cray Inc. or other vendors.

Global-View Abstractions

Chapel is described as supporting global-view abstractions for data and for control flow. In the tradition of ZPL and High Performance Fortran, Chapel supports the ability to declare and operate on large arrays in a holistic manner even though they may be implemented by storing their elements within the distributed memories of many distinct nodes. Chapel's designers felt that many of the most significant challenges to parallel programmability stem from the typical requirement that programmers write codes in a cooperating executable or Single Program, Multiple Data (SPMD) programming model. Such models require the user to manually manage many tedious details including data ownership, local-to-global index transformations, communication, and synchronization. This overhead often clutters a program's text, obscuring its intent and making the code



difficult to maintain and modify. In contrast, languages that support a global view of data such as ZPL, HPF, and Chapel shift this burden away from the typical user and onto the compiler, runtime libraries, and data distribution authors. The result is a user code that is cleaner and easier to understand, arguably without a significant impact on performance.

Chapel departs from the single-threaded logical execution models of ZPL and HPF by also providing a global view of control flow. Chapel's authors define this as a programming model in which a program's entry point is executed by a single logical task, and then additional parallelism is introduced over the course of the program through explicit language constructs such as parallel loops and the creation of new tasks. Supporting a global view for control flow and data structures makes parallel programming more like traditional programming by removing the requirement that users must write programs that are complicated by details related to running multiple copies of the program in concert as in the SPMD model.

A Multiresolution Design

Another departure from ZPL and HPF is that those languages provide high-level data-parallel abstractions without providing a means of abandoning those abstractions in order to program closer to the machine in a more explicit manner. Chapel's design team felt it was important for users to have such control in order to program as close to the machine as their algorithm requires, whether for reasons of performance or expressiveness. To this end, Chapel's features are designed in a layered manner so that when high-level abstractions like its global-view arrays are inappropriate, the programmer can drop down to lower-level features and control things more explicitly. As an example of this, Chapel's global-view arrays and data-parallel features are implemented in terms of its lower-level task-parallel and locality features for creating distinct tasks and mapping them to a machine's processors. The result is that users can write different parts of their program using different levels of abstraction as appropriate for that phase of the computation.

Locality and Affinity

The placement of data and tasks on large-scale machines is crucial for performance and scalability due to the

latencies incurred by communicating with other processors over a network. For this reason, performance-minded programmers typically need to control where data is stored on a large-scale system and where the tasks accessing that data will execute relative to the data. As multicore processors grow in the number and variety of compute resources, such control over locality is likely to become increasingly important for desktop programming as well. To this end, Chapel provides concepts that permit programmers to reason about the compute resources that they are targeting and to indicate where data and tasks should be located relative to those compute resources.

Narrowing the Gap Between Mainstream and HPC Languages

Chapel's designers believe there to be a wide gap between programming languages that are being used in education and mainstream computing such as Java, C#, Matlab, Perl, and Python and those being used by the High Performance Computing community: Fortran, C, and C++ in combination with MPI and OpenMP (and in some circles, Co-Array Fortran and UPC – Unified Parallel C). It was believed that this gap should be bridged in order to take advantage of productivity improvements in modern language design while also being able to better utilize the skills of the emerging workforce. The challenge was to design a language that would not alienate traditional HPC programmers who were perhaps most comfortable in Fortran or C. An example of such a design decision was to have Chapel support object-oriented programming since it is a staple of most modern languages and programmers, yet to make the use of objects optional so that Fortran and C programmers would not need to change their way of thinking about program and data structure design. Another example was to make Chapel an imperative, block-structured language, since the languages that have been most broadly adopted by both the mainstream and HPC communities have been imperative rather than functional or declarative in nature.

Language Features

Data Parallelism

Chapel's highest-level concepts are those relating to data parallelism. The central concept for data-parallel programming in Chapel is the domain, which is a first-class



language concept for representing an index set, potentially distributed between multiple processors. Domains are an extension of the region concept in ZPL. They are used in Chapel to represent iteration spaces and to declare arrays. A domain's indices can be Cartesian tuples of integers, representing dense or sparse index sets on a regular grid. They may also be arbitrary values, providing the capability for storing arbitrary sets or key/value mappings. Chapel also supports the notion of an unstructured domain in which the index values are anonymous, providing support for irregular, pointer-based data structures.

The following code declares three Chapel domains:

```
const D: domain(2) = [1..n, 1..n] ,
  DDiag: sparse subdomain(D)
    = [i in 1..n] (i,i),
  Employees: domain(string)
    = readNamesFromFile(infile);
```

In these declarations, *D* represents a regular 2-dimensional $n \times n$ index set; *DDiag* represents the sparse subset of indices from *D* describing its main diagonal; and *Employees* is a set of strings representing the names of a group of employees.

Chapel arrays are defined in terms of domains and represent a mapping from the domain's indices to a set of variables. The following declarations declare a pair of arrays for each of the domains above:

```
var A, B: [D] real,
  X, Y: [DDiag] complex,
  Age, SSN: [Employees] int;
```

The first declaration defines two arrays *A* and *B* over domain *D*, creating two $n \times n$ arrays of real floating point values. The second creates sparse arrays *X* and *Y* that store complex values along the main diagonal of *D*. The third declaration creates two arrays of integers representing the employees' ages and social security numbers.

Chapel users can express data parallelism using forall loops over domains or arrays. As an example, the following loops express parallel iterations over the indices and elements of some of the previously declared domains and arrays.

```
forall a in A do
  a += 1.0;

forall (i,j) in D do
  A(i,j) = B(i,j) + 1.0i * X(i,j);
```

```
forall person in Employees do
  if (Age(person) < 18) then
    SSN = 0;
```

Scalar functions and operators can be promoted in Chapel by calling them with array arguments. Such promotions also result in data-parallel execution, equivalent to the forall loops above. For example, each of the following whole-array statements will be computed in parallel in an element-wise manner:

```
A = B + 1.0i * X;
B = sin(A);
Age += 1;
```

In addition to the use cases described above, domains are used in Chapel to perform set intersections, to slice arrays and refer to subarrays, to perform tensor or elementwise iterations, and to dynamically resize arrays. Chapel also supports reduction and scan operators (including user-defined variations) for efficiently computing common collective operations in parallel. In summary, domains provide a very rich support for parallel operations on a rich set of potentially distributed data aggregates.

Locales

Chapel's primary concept for referring to machine resources is called the locale. A locale in Chapel is an abstract type, which represents a unit of the target architecture that can be used for reasoning about locality. Locales support the ability to execute tasks and to store variables, but the specific definition of the locale for a given architecture is defined by a Chapel compiler. In practice, an SMP node or multicore processor is often defined to be the locale for a system composed of commodity processors.

Chapel programmers specify the number of locales that they wish to use on the executable's command line. The Chapel program requests the appropriate resources from the target architecture and then spawns the user's code onto those resources for execution. Within the Chapel program's source text, the set of execution locales can be referred to symbolically using a built-in array of locale values named *Locales*. Like any other Chapel array, *Locales* can be sliced, indexed, or reshaped to organize the locales in any manner that suits the program. For example, the following statements create customized views of the locale set. The first divides the locales into two disjoint sets while the



second reshapes the locales into a 2-dimensional virtual grid of nodes:

```
// given: const Locales:
[0..#numLocales] locale;

const localeSetA = Locales[0..#localesInSetA],
    localeSetB = Locales[localesInSetA..];
const compGrid
    = Locales.reshape[1..2,numLocales/2];
```

Distributions

As mentioned previously, a domain's indices may be distributed between the computational resources on which a Chapel program is running. This is done by specifying a domain map as part of the domain's declaration, which defines a mapping from the domain's index set to a target locale set. Since domains are used to define iteration spaces and arrays, this also implies a distribution of the computations and data structures that are expressed in terms of that domain. Chapel's domain maps are more than simply a mapping of indices to locales, however. They also define how each locale should store its local domain indices and array elements, as well as how operations such as iteration, random access, slicing, and communication are defined on the domains and arrays. To this end, Chapel domain maps can be thought of as recipes for implementing parallel, distributed data aggregates in Chapel. The domain map's functional interface is targeted by the Chapel compiler as it rewrites a user's global-view array operations down to the per-node computations that implement the overall algorithm.

A well-formed domain map does not affect the semantics of a Chapel program, only its implementation and performance. In this way, Chapel programmers can tune their program's implementation simply by changing the domain declarations, leaving the bulk of the computation and looping untouched. For example, the previous declaration of domain D could be changed as follows to specify that it should be distributed using an instance of the Block distribution:

```
const D: domain(2) dmapped MyBlock
    = [1..n, 1..n];
```

However, none of the loops or operations written previously on D , A , or B would have to change.

Advanced users can write their own domain maps in Chapel and thereby create their own implementations

of parallel, distributed data structures. While Chapel provides a standard library of domain maps, a major research goal of the language is to implement these standard distributions using the same mechanism that an end-user would rather than by embedding semantic knowledge of the distributions into the compiler and runtime as ZPL and HPF did.

Task Parallelism

As mentioned previously, Chapel's data-parallel features are implemented in terms of its lower-level task-parallel features. The most basic task-parallel construct in Chapel is the begin statement, which creates a new task while allowing the original task to continue executing. For example, the following code starts a new task to compute an FFT while the original task goes on to compute a Jacobi iteration:

```
begin FFT(A);
Jacobi(B);
```

Inter-task coordination in Chapel is expressed in a data-centric way using special variables called synchronization variables. In addition to storing a traditional data value, these variables also maintain a logical full/empty state. Reads to synchronization variables block until the variable is full and leave the variable empty. Conversely, writes block until the variable is empty and leave it full. Variations on these default semantics are provided via method calls on the variable. This leads to the very natural expression of inter-task coordination. Consider, for example, the elegance of the following bounded buffer producer/consumer pattern:

```
var buffer: [0..#buffsize] sync int;

begin { // producer
    for i in 0..n do
        buffer[i%buffsize] = ...;
}
{ // consumer
    for j in 0..n do
        ...buffer[j%buffsize]...;
}
```

Because each element in the bounded buffer is declared as a synchronized integer variable, the consumer will not read an element from the buffer until the producer has written to it, marking its state as full. Similarly, the producer will not overwrite values in the buffer until the consumer's reads have reset the state to empty.



In addition to these basic task creation and synchronization primitives, Chapel supports additional constructs to create and synchronize tasks in structured ways that support common task-parallel patterns.

Locality Control

While domain maps provide a high-level way of mapping iteration spaces and arrays to the target architecture, Chapel also provides a low-level mechanism for controlling locality called the on-clause. Any Chapel statement may be prefixed by an on-clause that indicates the locale on which that statement should execute. On-clauses can take an expression of locale type as their argument, which specifies that the statement should be executed on the specified locale. They may also take any other variable expression, in which case the statement will execute on the locale that stores that variable. For example, the following modification to an earlier example will execute the FFT on Locale #1 while executing the Jacobi iteration on the locale that owns element i,j of B:

```
on Locales[1] do begin FFT(A);
on B(i,j) do Jacobi(B);
```

Base Language

Chapel's base language was designed to support productive parallel programming, the ability to achieve high performance, and the features deemed necessary for supporting user-defined distributions effectively. In addition to a fairly standard set of types, operators, expressions, and statements, the base language supports the following features:

- A rich compile-time language: Chapel supports the ability to define functions that are evaluated at compile time, including functions that return types. Users may also indicate conditionals that should be folded at compile time as well as loops that should be statically unrolled.
- Static-type inference: Chapel supports a static-type inference scheme in which specifications can be omitted in most declaration settings, causing the compiler to infer the types from context. For example, a variable declaration may omit the variable's type as long as it is initialized, in which case the compiler infers its type from the initializing expression. This supports exploratory programming as in scripting languages without the runtime overhead of

dynamic typing. It also supports generic programming and code reuse.

- Iterators: Chapel's iterators are functions that yield multiple values over their lifetime (as in CLU or Ruby) rather than returning a single time. These iterators can be used to control serial and parallel loops.
- Configuration variables: Chapel's configuration variables are symbols whose default values can be overridden on the command line of the compiler or executable using argument parsing that is implemented automatically by the compiler.
- Tuples: These support the ability to group values together in a lightweight manner, for example, to return multiple values from a function or to represent multidimensional array indices using a single variable.

Future Directions

At the time of this writing, the Chapel language is still evolving based on user feedback, code studies, and the implementation effort. Some notable areas where additional work is expected in the future include:

- Support for heterogeneity: Heterogeneous systems are becoming increasingly common, especially those with heterogeneous processor types such as traditional CPUs paired with accelerators such as graphics processing units (GPUs). To support such systems, it is anticipated that Chapel's locale concept will need to be refined to expose architectural substructures in abstract and/or concrete terms.
- Transactional memory: Chapel has plans to support an atomic block for expressing transactional computations against memory, yet software transactional memory (STM) is an active research area in general, and becomes even trickier in the distributed memory context of Chapel programs.
- Exceptions: One of Chapel's most notable omissions is support for exception- and/or error-handling mechanisms to deal with software failures in a robust way, and perhaps also to be resilient to hardware failures. This is an area where the original Chapel team felt unqualified to make a reasonable proposal and intended to fill in that lack over time.

In addition to the above areas, future design and implementation work is expected to take place in the areas of task teams, dynamic load balancing, garbage



collection, parallel I/O, language interoperability, and tool support.

Related Entries

- ▶ [Fortress \(Sun HPCS Language\)](#)
- ▶ [HPF \(High Performance Fortran\)](#)
- ▶ [NESL](#)
- ▶ [PGAS \(Partitioned Global Address Space\) Languages](#)
- ▶ [Tera MTA](#)
- ▶ [ZPL](#)

Bibliographic Notes and Further Reading

The two main academic papers providing an overview of Chapel's approach are also two of the earliest: "The Cascade High Productivity Language" [1] and "Parallel Programmability and the Chapel Language" [4]. While the language has continued to evolve since these papers were published, the overall motivations and concepts are still very accurate. The first paper is interesting in that it provides an early look at the original team's design while the latter remains a good overview of the language's motivations and concepts. For the most accurate description of the language at any given time, the reader is referred to the Chapel Language Specification. This is an evolving document that is updated as the language and its implementation improve. At the time of this writing, the current version is 0.796 [9].

Other important early works describing specific language concepts include "An Approach to Data Distributions in Chapel" by Roxana Diaconescu and Hans Zima [13]. While the approach to user-defined distributions eventually taken by the Chapel team differs markedly from the concepts described in this paper [6], it remains an important look into the early design being pursued by the Caltech/JPL team. Another concept-related paper is "Global-view Abstractions for User-Defined Reductions and Scans" by Deitz, Callahan, Chamberlain, and Snyder, which explored concepts for user-defined reductions and scans in Chapel [12].

In the trade press, a good overview of Chapel in Q&A form entitled "Closing the Gap with the Chapel Language" was published in HPCWire in 2008 [14]. Another Q&A-based document is a position paper entitled "Multiresolution Languages for Portable yet Efficient Parallel Programming," which espouses the

use of multiresolution language design in efforts like Chapel [3].

For programmers interested in learning how to use Chapel, there is perhaps no better resource than the release itself, which is available as an open-source download from SourceForge [7]. The release is made available under the Berkeley Software Distribution (BSD) license and contains a portable implementation of the Chapel compiler along with documentation and example codes. Another resource is a tutorial document that walks through some of the HPC Challenge benchmarks, explaining how they can be coded in Chapel. While the language has evolved since the tutorial was last updated, it remains reasonably accurate and provides a gentle introduction to the language [5]. The Chapel team also presents tutorials to the community fairly often, and slide decks from these tutorials are archived at the Chapel Web site [8].

Many of the resources above, as well as other useful resources such as presentations and collaboration ideas, can be found at the Chapel project Web site hosted at Cray [8].

To read about Chapel's chief influences, the best resources are probably dissertations from the ZPL team [2, 11], the High Performance Fortran Handbook [15], and the Cray XMT Programming Environment User's Guide [10] (the Cray XMT is the current incarnation of the Tera/Cray MTA).

Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

Bibliography

1. Callahan D, Chamberlain B, Zima H (April 2004) The Cascade high productivity language. 9th International workshop on high-level parallel programming models and supportive environments, pp 52–60, Santa Fe, NM
2. Chamberlain BL (November 2001) The design and implementation of a region-based parallel language. PhD thesis, University of Washington
3. Chamberlain BL (October 2007) Multiresolution languages for portable yet efficient parallel programming. <http://chapel.cray.com/papers/DARPA-RFI-Chapel-web.pdf>. Accessed 4 May 2011

4. Chamberlain BL, Callahan D, Zima HP (August 2007) Parallel programmability and the Chapel language. *Int J High Perform Comput Appl* 21(3):291–312
5. Chamberlain BL, Deitz SJ, Hribar MB, Wong WA (November 2008) Chapel tutorial using global HPCC benchmarks: STREAM Triad, Random Access, and FFT (revision 1.6). <http://chapel.cray.com/hpcc/hpccTutorial-1.6.pdf>. Accessed 4 May 2011
6. Chamberlain BL, Deitz SJ, Iten D, Choi S-E (2010) User-defined distributions and layouts in Chapel: Philosophy and framework. In: Hot-PAR '10: Proceedings of the 2nd USENIX workshop on hot topics, June 2010
7. Chapel development site at SourceForge. <http://sourceforge.net/projects/chapel>. Accessed 4 May 2011
8. Chapel project website. <http://chapel.cray.com>. Accessed 4 May 2011
9. Cray Inc., Seattle, WA. Chapel Language Specification (version 0.796), October 2009. <http://chapel.cray.com/papers.html>. Accessed 4 May 2011
10. Cray Inc. Cray XMT Programming Environment User's Guide, March 2009 (see <http://docs.cray.com>). Accessed 4 May 2011
11. Deitz SJ (2005) High-Level Programming Language Abstractions for Advanced and Dynamic Parallel Computations. PhD thesis, University of Washington
12. Deitz SJ, Callahan D, Chamberlain BL, Synder L (March 2006) Global-view abstractions for user-defined reductions and scans. In: PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on principles and practice of parallel programming, pp 40–47. ACM Press, New York
13. Diaconescu R, Zima HP (August 2007) An approach to data distributions in Chapel. *Intl J High Perform Comput Appl* 21(3):313–335
14. Feldman M, Chamberlain BL (2008) Closing the parallelism gap with the Chapel language. HPCWire, November 2008. http://www.hpcwire.com/hpcwire/2008-11-19/closing_the_parallelism_gap_with_the_chapel_language.html. Accessed 4 May 2011
15. Koelbel CH, Loveman DB, Schreiber RS, Steele Jr GL, Zosel ME (September 1996) The High Performance Fortran handbook. Scientific and engineering computation. MIT Press, Cambridge, MA

Charm++

LAXMIKANT V. KALÉ
University of Illinois at Urbana-Champaign, Urbana,
IL, USA

Definition

Charm++ is a C++-based parallel programming system that implements a *message-driven migratable objects* programming model, supported by an adaptive runtime system.

Discussion

Charm++ [1] is a parallel programming system developed at the University of Illinois at Urbana-Champaign. It is based on a message-driven migratable objects programming model, and consists of a C++-based parallel notation, an adaptive runtime system (RTS) that automates resource management, a collection of debugging and performance analysis tools, and an associated family of higher level languages. It has been used to program several highly scalable parallel applications.

Motivation and Design Philosophy

One of the main motivations behind Charm++ is the desire to create an optimal division of labor between the programmer and the system: that is, to design a programming system so that the programmers do what they can do best, while leaving to the “system” what it can automate best. It was observed that deciding what to do in parallel is relatively easy for the application developer to specify; conversely, it has been very difficult for a compiler (for example) to automatically parallelize a given sequential program. On the other hand, automating resource management – which subcomputation to carry out on what processor and which data to store on a particular processor – is something that the system may be able to do better than a human programmer, especially as the complexity of the resource management task increases. Another motivation is to emphasize the importance of data locality in the language, so that the programmer is made aware of the cost of non-local data references.

Programming Model in Abstract

In Charm++, computation is specified in terms of collections of objects that interact via asynchronous method invocations. Each object is called a *chare*. Chares are assigned to processors by an adaptive runtime system, with an optional override by the programmer. A chare is a special kind of C++ object. Its behavior is specified by a C++ class that is “special” only in the sense that it must have at least one method designated as an “entry” method. Designating a method as an entry method signifies that it can be invoked from a remote processor. The signatures of the entry methods (i.e., the type and structure of its parameters) are specified in a separate interface file, to allow the system to generate code for packing (i.e., serializing) and unpacking the parameters into messages. Other than the existence of

the interface files, a Charm++ program is written in a manner very similar to standard C++ programs, and thus will feel very familiar to C++ programmers.

The chares communicate via *asynchronous* method invocations. Such a method invocation does not return any value to the caller, and the caller continues with its own execution. Of course, the called chare may choose to send a value back by invoking an entry method upon the caller object. Each chare has a globally valid ID (its *proxy*), which can be passed around via method invocations. Note that the programmer refers to only the target chare by its global ID, and not by the processor on which it resides. Thus, in the baseline Charm++ model, the processor is not a part of the ontology of the programmer.

Chares can also create other chares. The creation of new chares is also asynchronous in that the caller does not wait until the new object is created. Programmers typically do not specify the processor on which the new chare is to be created; the system makes this decision at runtime. The number of chares may vary over time, and is typically much larger than the number of processors.

Message-Driven Scheduler

At any given time, there may be several pending method invocations for the chares on a processor. Therefore, the Charm++ runtime system employs a *user-level* scheduler (Fig. 1) on each processor. The scheduler is user-level in the sense that the operating system is not aware of it. Normal Charm++ methods are non-preemptive: once a method begins execution, it returns control to the scheduler only after it has completed execution. The scheduler works with a queue of pending entry

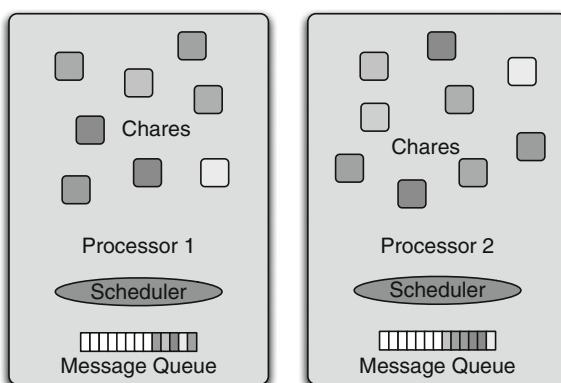
method invocations. Note that this queue may include asynchronous method invocations for chares located on this processor, as well as “seeds” for the creation of new chares. These seeds can be thought of as invocations of the constructor entry method. The scheduler repeatedly selects a message (i.e., a pending method invocation) from the queue, identifies the object targeted, creating an object if necessary, unpacks the parameters from the message if necessary, and then invokes the specified method with the parameters. Only when the method returns does it select the next message and repeats the process.

Chare-arrays and Iterative Computations

The model described so far, with its support for dynamic creation of work, is well-suited for expressing divide-and-conquer as well as divide-and-divide computations. The latter occur in state-space search. Charm++ (and its C-based precursor, Charm, and Chare Kernel [2]) were used in the late 1980s for implementing parallel Prolog [3] as well as several combinatorial search applications [4].

In principle, singleton chares could also be used to create the arbitrary networks of objects that are required to decompose data in Science and Engineering applications. For example, one can organize chares in a two-dimensional mesh network, and through some additional message passing, ensure that each chare knows the ID of its four neighboring chares. However, this method of creating a network of chares is quite cumbersome, as it requires extensive bookkeeping on the part of the programmer. Instead, Charm++ supports indexed collections of chares, called *chare-arrays*. A Charm++ computation may include multiple chare-arrays. Each chare-array is a collection of chares of the same type. Each chare is identified by an index that is unique within its collection. Thus, an individual chare belonging to a chare-array is completely identified by the ID of the chare-array and its own index within it. Common index structures include dense as well as sparse multidimensional arrays, but arbitrary indices such as strings or bit vectors are also possible. Elements in a chare-array may be created all at once, or can be inserted one at a time.

Method invocations can be broadcast to an entire chare-array, or a section of it. Reductions over chare-arrays are also supported, where each chare in a chare-array contributes a value, and all submitted values are combined via a commutative-associative operation. In



Charm++. Fig. 1 Message-driven scheduler

many other programming models, a reduction is a collective operation which blocks all callers. In contrast, reductions in Charm++ are non-blocking, that is, asynchronous. A `contribute` call simply deposits the value created by the calling chare into the system and returns to its caller. At some later point after all the values have been combined, the system delivers them to a user-specified callback. The callback, for example, could be a broadcast to an entry method of the same chare-array.

Charm++ does not allow generic global variables, but it does allow “specifically shared variables.” The simplest of these are read-only variables, which are initialized in the main chare’s constructor, and are treated as constants for the remainder of the program. The runtime system (RTS) ensures that a copy of each read-only variable is available on all physical processors.

Benefits of Message-driven Execution

The message-driven execution model confers several performance and/or productivity benefits.

Automatic and Adaptive Overlap of Computation and Communication

Since objects are scheduled based on the availability of messages, no single object can occupy the processor while waiting for some remote data. Instead, objects that have asynchronous method invocations (messages) waiting for them in the scheduler’s queue are allowed to execute. This leads to a natural overlap of communication and computation, without any extra work from the programmer. For example, a chare may send a message to a remote chare and wait for another message from it before continuing. The ensuing communication time, which would otherwise be an idle period, is naturally and automatically filled in (i.e., overlapped) by the scheduler with useful computation, that is, processing of another message from the scheduler’s queue for another chare.

Concurrent Composition

The ability to compose in parallel two individually parallel modules is referred to as concurrent composition. Consider two modules P and Q that are both ready to execute and have no direct dependencies among them. With other programming models (e.g., MPI) that

associate work directly with processors, one typically has two options. One may divide the set of processors so that a subset is executing one module (P) while the remaining processors execute the other (Q). Alternatively, one may sequentialize the modules, executing P first, followed by Q, on all processors. Neither alternative is efficient. Allowing the two modules to interleave the execution on all processors is often beneficial, but is hard to express even with wild-card receives, and it breaks abstraction boundaries between the modules in any case. With message driven execution, such interleaving happens naturally, allowing idle time in one module to be overlapped with computation in the other. Coupled with the ability of the adaptive runtime system to migrate communicating objects closer to each other, this adds up to strong support for concurrent composition, and thereby for increased modularity.

Prefetching Data and Code

Since the message driven scheduler can examine its queue, it knows what the next several objects scheduled to execute are and what methods they will be executing. This information can be used to asynchronously prefetch data for those objects, while the system executes the current object. This idea can be used by the Charm++ runtime system for increasing efficiency in various contexts, including on accelerators such as the Cell processor, for out-of-core execution and for prefetching data from DRAM to cache.

Capabilities of the Adaptive Runtime System Based on Migrability of Chares

Other capabilities of the runtime system arise from the ability to migrate chares across processors and the ability to place newly created chares on processors of its choice.

Supporting Task Parallelism with Seed Balancers

When a program calls for the creation of a singleton chare, the RTS simply creates a seed for it. This seed includes the constructor arguments and class information needed to create a new chare. Typically, these seeds are initially stored on the same processor where they are created, but they may be passed from processor to processor under the control of a runtime component called

the *seed balancer*. Different seed balancer strategies are provided by the RTS. For example, in one strategy, each processor monitors its neighbors' queues in addition to its own, and balances seeds between them as it sees fit. In another strategy, a processor that becomes idle requests work from a random donor – a work stealing strategy [5, 6]. Charm++ also includes strategies that balance priorities and workloads simultaneously, in order to give precedence to high-priority work over the entire system of processors.

Migration-based Load Balancers

Elements of chare-arrays can be migrated across processors, either explicitly by the programmer or by the runtime system. The Charm++ RTS leverages this capability to provide a suite of dynamic load balancing strategies. One class of such strategies is based on the *principle of persistence*, which is the empirical observation that in most science and engineering applications expressed in terms of their natural objects, computational loads and communication patterns tend to persist over time, even for dynamically evolving applications. Thus, the recent past is a reasonable predictor of the near future. Since the runtime system mediates communication and schedules computations, it can automatically instrument its execution so as to measure computational loads and communication patterns accurately. Load balancing strategies can use these measurements, or alternatively, any other mechanisms for predicting such patterns. Multiple load balancing strategies are available to choose from. The choice may depend on the machine context and applications, although one can always use the default strategy provided. Programmers can write their own strategy, either to specialize it to the specific needs of the application or in the hope of doing better than the provided strategies.

Dynamically Altering the Sets of Processors Used

A Charm++ program can be asked to change the set of processors it is using at runtime, without requiring any effort by the programmer. This can be useful to increasing utilization of a cluster running multiple jobs that arrive at unpredictable times. The RTS accomplishes this by migrating objects and adjusting its runtime data structures, such as spanning trees used in its collective

operations. Thus, a $1,024 \times 1,024 \times 1,024$ cube of data partitioned into $16 \times 16 \times 16$ array of chares, each holding $64 \times 64 \times 64$ data subcube, can be shrunk from 256 cores to (say) 247 cores without significantly losing efficiency. Of course, some cores will house 17 objects, instead of the 16 objects they did earlier.

Fault Tolerance

Charm++ provides multiple levels of support for fault tolerance, including alternative competing strategies. At a basic level, it supports automated application-level checkpointing by leveraging its ability to migrate objects. With this, it is possible to create a checkpoint of the program without requiring extra user code. More interestingly, it is also possible to use a checkpoint created on P processors to restart the computation on a different number of processors than P.

On appropriate machines and with job schedulers that permit it, Charm++ can also automatically detect and recover from faults. This requires that the job scheduler not kill a job if one of its nodes were to fail. At the time of this writing, these schemes are available on workstation clusters. The most basic strategy uses the checkpoint created on disk, as described above, to effect recovery. A second strategy avoids using disks for checkpointing, instead creating two checkpoints of each chare in the memory of two processors. It is suitable for those applications whose memory footprint at the point of checkpointing is relatively small compared with the available memory. Fortunately, many applications such as molecular dynamics and computational astronomy fall into this category. When it can be used, it is very fast, often accomplishing a checkpoint in less than a second and recovery in a few seconds. However, both strategies described above send *all* processors back to their checkpoints even when just one out of a million processors has failed. This wastes all the computation performed by processors that did not fail. As the number of processors increases and, consequently, the MTBF decreases, this will become an untenable recovery strategy. A third experimental strategy in Charm++ sends only the failed processor(s) to their checkpoints by using a message-logging scheme. It also leverages the over-decomposition and migratability of Charm++ objects to parallelize the restart process. That is, the

objects from failed processors are reincarnated on multiple other processors, where they re-execute, in parallel, from their checkpoints using the logged messages. Charm++ also provides a fourth *proactive* strategy to handle situations where a future fault can be predicted, say based on heat sensors, or estimates of increasing (corrected) cache errors. The runtime simply migrates objects away from such a processor and readjusts its runtime data structures.

Associated Tools

Several tools have been created to support the development and tuning of Charm++ applications. *LiveViz* allows one to inject messages into a running program and display attributes and images from a running simulation. *Projections* supports performance analysis and visualization, including live visualization, parallel on-line analysis, and log-based post-mortem analysis. *CharmDebug* is a parallel debugger that understands Charm++ constructs, and provides online access to runtime data structures. It also supports a sophisticated record-replay scheme and provisional message delivery for dealing with nondeterministic bugs. The communication required by these tools is integrated in the runtime system, leveraging the message-driven scheduler. No separate monitoring processes are necessary.

Code Example

[Figures 2](#) and [3](#) show fragments from a simple Charm++ example program to give a flavor of the programming model. The program is a simple Lennard-Jones molecular dynamics code. The computation is decomposed into a one-dimensional array of LJ objects, each holding a subset of atoms. The interface file ([Fig. 2](#)) describes

the main Charm-level entities, their types and signatures. The program has a read-only integer called numChares that holds the size of the LJ chare-array. In this particular program, the main chare is called Main and has only one method, namely, its constructor. The LJ class is declared as constituting a one-dimensional array of chares. It has two entry methods in addition to its constructor. Note the others [n] notation used to specify a parameter that is an array of size n, where n itself is another integer parameter. This allows the system to generate code to serialize the parameters into a message, when necessary. CkReductionMsg is a system-defined type which is used as a target of entry methods used in reductions.

Some important fragments from the C++ file that define the program itself are shown in [Fig. 3](#). Sequential code not important for understanding the program is omitted. The classes Main and LJ inherit from classes generated by a translator based on the interface file. The program execution consists of a number of time steps. In each time step, each processor sends its particles on a round-trip to visit all other chares. Whenever a packet of particles visits a chare via the passOn method, the chare calculates forces on each of the visiting (others) particles due to each of its own particles. Thus, when the particles return home after the round-trip, they have accumulated forces due to all the other particles in the system. A sequential call (integrate) then adds local forces to the accumulated forces and calculates new velocities and positions for each owned particle. At this point, to make sure the time step is truly finished for all chares, the program uses an “asynchronous reduction” via the contribute call. To emphasize the asynchronous nature of this call, the example makes

```
mainmodule ljdyn {
    readonly int numChares;
    ...
    mainchare Main {
        entry Main(CkArgMsg *m);
    };

    array [1D] LJ {
        entry LJ(void);
        entry void passOn(int home, int n, Particle others[n]);
        entry void startNextStep(CkReductionMsg *m);
    };
}
```

Charm++. Fig. 2 A simple molecular dynamics program: interface file

```

/*readonly*/ int numChares;

classMain : public CBase_Main {
    Main(CkArgMsg*m) {
        //Process command-line arguments
        ...
        numChares = atoi(m->argv[1]);
        ...
        CProxy_LJ arr = CProxy_LJ::ckNew(numChares);
    }
};

class LJ : public CBase_LJ {
    int timeStep, numParticles, next;
    Particle * myParticles;
    ...
    LJ() {
        ...
        myParticles = new Particle[numParticles];
        ... // initialize particle data
        next = (thisIndex + 1) % numChares;
        timeStep = 0;
        startNextStep((CkReductionMsg * )NULL);
    }

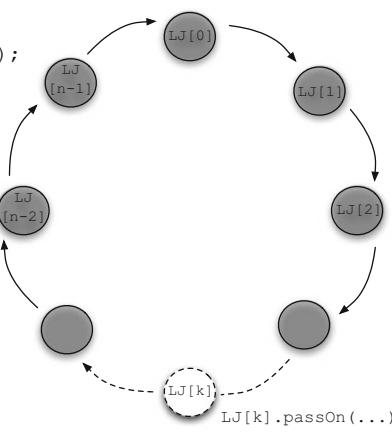
    void startNextStep(CkReductionMsg* m) {
        if (++timeStep > MAXSTEPS) {
            if (thisIndex == 0) { ckout << "Done\n" << endl; CkExit(); }
        } else
            thisProxy[next].passOn(thisIndex, numParticles, myParticles);
    }

    void passOn(int homeIndex, int n, Particle* others) {
        if (thisIndex != homeIndex) {
            interact(n, others); //add forces on "others" due to my particles
            thisProxy[next].passOn(homeIndex,n, others);
        } else { // particles are home, with accumulated forces
            CkCallback cb(CkIndex_LJ::startNextStep(NULL), thisProxy);
            contribute(cb); // asynchronous barrier
            integrate( n, others); // add forces and update positions
        }
    }

    void interact(int n, Particle* others){
        /* add forces on "others" due to my particles */
    }

    void integrate(int n, Particle* others){
        /*... apply forces, update positions... */
    }
};

```



Charm++.Fig. 3 A simple molecular dynamics program: fragments from the C++ file

the call before `integrate`. The `contribute` call simply deposits the contribution into the system and continues on to `integrate`. The `contribute` call specifies that after all the array elements of `LJ` have contributed, a callback will be made. In this case, the callback is

a broadcast to all the members of the chare-array at the entry-method `startNextStep`. Inherited variable `thisProxy` is a proxy to the entire chare-array. Similarly, `thisIndex` refers to the index of the calling chare in the chare-array to which it belongs.

Language Extensions and Features

The baseline programming model described so far is adequate to express all parallel interaction structures. However, for programming convenience, increased productivity and/or efficiency, Charm++ supports a few additional features. For example, individual entry methods can be marked as “threaded.” This results in the creation of a user-level, lightweight thread whenever the entry method is invoked. Unlike normal entry methods, which always complete their execution and return control to the scheduler before other entry methods are executed, threaded entry methods can block their execution; of course they do so without blocking the processor they are running on. In particular, they can wait for a “future,” wait until another entry method unblocks them, or make blocking method invocations. An entry method can be tagged as blocking (actually called a “sync” method), and such a method is capable of returning values, unlike normal methods that are asynchronous and therefore have a return type of `void`.

Often, threaded entry methods are used to describe the life cycle of a chare. Another notation within Charm++, called “structured dagger,” accomplishes the same effect without the need for a separate stack and associated memory for each user level thread and the, admittedly small, overhead associated with thread context switches. However, it requires that all dependencies on remote data be expressed in this notation within the text of an entry-method. In contrast, the thread of control may block waiting for remote data within functions called from a threaded entry method.

Charm++ as described so far does not bring in the notion of a “processor” in the programming model. However, some low-level constructs that refer to processors are also provided to programmers and especially to library writers. For example, when a chare is created, one can optionally specify which processor to create it on. Similarly, when a chare-array is created, one can specify its initial mapping to processors. One can create specialized chare-arrays, called *groups*, that have exactly one member on each processor, which are useful for implementing services such as load balancers.

Languages in the Charm Family

AMPI or Adaptive MPI is an implementation of the message passing interface standard on top of the

Charm++ runtime system. Each MPI process is implemented as a user level thread that is embedded inside a Charm++ object, as a threaded entry method. These objects can be migrated across processors, as is usual for Charm++ objects, thus bringing benefits of the Charm++ adaptive runtime system, such as dynamic load balancing and fault tolerance, to traditional MPI programs. Since there may be multiple MPI “processes” on each core, commensurate with the overdecomposition strategy of Charm++ applications, the MPI programs need to be modified in a mechanical, systematic fashion to avoid conflict among the global variables. Adaptive MPI provides tools for automating this process to some extent. As a result, a standard Adaptive MPI program is also a legal MPI program, but the converse is true only if the use of global variables has been handled via such modifications. In addition, Adaptive MPI provides primitives such as asynchronous collectives, which are not part of the MPI 2 standard. An asynchronous reduction, for example, carries out the communication associated with a reduction in the background while the main program continues on with its computation. A blocking call is then used to fetch the result of the reduction.

Two recent languages in the Charm++ family are multiphase shared arrays (MSA) and Charisma. These are part of the Charm++ strategy of creating a toolbox consisting of incomplete languages that capture some interaction modes elegantly and frameworks that capture the needs of specific domains or data structures, both backed up by complete languages such as Charm++ and AMPI. The compositionality afforded by message-driven execution ensures that modules written using multiple paradigms can be efficiently composed in a larger application.

MSA is designed to support disciplined use of shared address space. The computation consists of collections of threads and multiple user-defined data arrays, each partitioned into user-defined pages. Both entities are implemented as migratable objects (i.e., chares) available to the Charm++ runtime system. The threads can access the data in the arrays, but each array is in only one of a restrictive set of access modes at a time. *Read-only*, *exclusive-write*, and *accumulate* are examples of the access modes supported by MSA. At designated synchronization points,

a program may change the access modes of one or more arrays.

Charisma, another language implemented on top of Charm++, is designed to support computations that exhibit a static data flow pattern among a set of Charm++ objects. Such a pattern, where the flow of messages remains the same from iteration to iteration, even though the content and the length of messages may change, is extremely common in science and engineering applications. For such applications, Charisma provides a convenient syntax that captures the flow of values and control across multiple collections of objects clearly. In addition, Charisma provides a clean separation of sequential and parallel code that is convenient for collaborative application development involving parallel programmers and domain specialists.

Frameworks atop Charm++

In addition to its use as a language for implementing applications directly, Charm++ is seen as backend for higher level frameworks and languages described above. Its utility in this context arises because of the interoperability and runtime features it provides, which one can leverage to put together a new domain-specific framework relatively quickly. An example of such a framework is ParFUM, which is aimed at unstructured mesh applications. ParFUM allows developers of sequential codes based on such meshes to retarget them to parallel machines, with relatively few changes. It automates several commonly needed functions including the need to exchange boundary nodes (or boundary layers, in general) with neighboring objects. Once a code is ported to ParFUM, it can automatically benefit from other Charm++ features such as load balancing and fault tolerance.

Applications

Some of the highly scalable applications developed using Charm++ are in extensive use by scientists on national supercomputers. These include NAMD (for biomolecular simulations), OpenAtom (for electronic structure simulations), and ChaNGa (for astrophysical N-body simulations).

Origin and History

The Chare Kernel, a precursor of the Charm++ system, arose from the work on parallel Prolog at the University of Illinois in Urbana-Champaign in the late 1980s. The implementation mechanism required a collection of computational entities, one for each active clause (i.e., its activation record) of the underlying logic program. Each of these entities typically received multiple responses from its children in the proof tree, and they needed to create new nodes in the proof tree which had to fire new “tasks” for each of its active clauses. The implementation led to a message-driven scheduler and dynamic creation of the seeds of work. These entities were called *chares*, borrowing the term used by an earlier parallel functional-languages project, RediFlow. Chare Kernel essentially separated this implementation mechanism from its parallel Prolog context, into a C-based parallel programming paradigm of its own. Charm had similarities (in particular, its message-driven execution) with the earlier research on reworking of the Hewitt’s Actor framework by Agha and Yonezawa [7, 8]. However, its intellectual progenitors were in parallel logic and functional languages. With the increase in popularity of C++, Charm++ became the version of Charm for C++, which was a natural fit for its object-based abstraction. As many researchers in parallel logic programming shifted attention to scientific computations, indexed collections of migratable chares were developed in Charm++ to simplify addressing chares in mid-1990s. In the late 1990s, Adaptive MPI was developed in the context of applications being developed at the Center for Simulation of Advanced Rockets at Illinois. Charm++ continues to be developed and maintained from the University of Illinois, and its applications are in regular use at many supercomputers around the world.

Availability and Usage

Charm++ runs on most parallel machines available at the time this entry was written, including multicore desktops, clusters, and large-scale proprietary supercomputers, running Linux, Windows, and other operating systems. Charm++, its associated software tools and libraries can be downloaded in source code and binary forms from <http://charm.cs.illinois.edu> under a license that allows free use for noncommercial purposes.

Related Entries

- [Actors](#)
- [NAMD \(NAnoscale Molecular Dynamics\)](#)
- [Combinatorial Search](#)

Bibliographic Notes and Further Reading

One of the earliest papers on the Chare Kernel, the precursor of Charm++ was published in 1989 [9], followed by a more detailed description of the model and its load balancers [2, 10]. This work arose out of earlier work on parallel Prolog [3]. The C++ based version was described in an OOPSLA paper in 1993 [1] and was expanded upon in a book on parallel C++ in [11]. Early work on quantifying benefits of the programming model is summarized in a later paper [12].

Some of the early applications using Charm++ were in symbolic computing and, specifically, in parallel combinatorial search [4]. A scalable framework for supporting migratable arrays of chares is described in a paper [13], which is also useful for understanding the programming model. An early paper [14] describes support for migrating Chares for dynamic load balancing. Recent papers describe an overview of Charm++ [15] and its applications [16].

Bibliography

1. Kale LV, Krishnan S (1993) CHARM++: a portable concurrent object oriented system based on C++. In: Paepcke A (ed) Proceedings of OOPSLA'93, ACM, New York, September 1993, pp 91–108
2. Shu WW, Kale LV (1990) Chare Kernel – a runtime support system for parallel computations. *J Parallel Distrib Comput* 11:198–211
3. Kale LV (1987) Parallel execution of logic programs: the REDUCE-OR process model. In: Proceedings of the fourth international conference on logic programming, Melbourne, May 1987, pp 616–632
4. Kale LV, Ramkumar B, Salelore V, Sinha AB (1993) Prioritization in parallel symbolic computing. In: Ito T, Halstead R (eds) Lecture notes in computer science, vol 748. Springer, pp 12–41
5. Lin Y-J, Kumar V (1991) And-parallel execution of logic programs on a sharedmemory multiprocessor. *J Logic Program* 10(1/2/3&4):155–178
6. Frigo M, Leiserson CE, Randall KH (1998) The implementation of the Cilk-5 multithreaded language. In: ACM SIGPLAN '98 conference on programming language design and implementation (PLDI), Montreal, June 1998. vol 33 of ACM Sigplan Notices, pp 212–223

7. Agha G (1986) Actors: a model of concurrent computation in distributed systems. MIT, Cambridge
8. Yonezawa A, Briot J-P, Shibayama E (1986) Object-oriented concurrent programming in ABCL/I. *ACM SIGPLAN Notices*, Proceedings OOPSLA '86, Nov 1986, 21(11):258–268
9. Kale LV, Shu W (1989) The Chare Kernel base language: preliminary performance results. In: Proceedings of the 1989 international conference on parallel processing, St. Charles, August 1989, pp 118–121
10. Kale LV (1990) The Chare Kernel parallel programming language and system. In: Proceedings of the international conference on parallel processing, August 1990, vol II, pp 17–25
11. Kale LV, Krishnan S (1996) Charm++: parallel programming with message-driven objects. In: Wilson GV, Lu P (eds) Parallel programming using C++. MIT, Cambridge, pp 175–213
12. Gursoy A, Kale LV (2003) Performance and modularity benefits of message-driven execution. *J Parallel Distrib Comput* 64:461–480
13. Lawlor OS, Kale LV (2003) Supporting dynamic parallel object arrays. *Concurr Comput Pract Exp* 15:371–393
14. Brunner RK, Kale LV (2000) Handling application-induced load imbalance using parallel objects. In: Parallel and distributed computing for symbolic and irregular applications. World Scientific, Singapore, pp 167–181
15. Kale Lv, Zheng G (2009) Charm++ and AMPI: adaptive runtime strategies via migratable objects. In: Parashar M (ed) Advanced computational infrastructures for parallel and distributed applications. Wiley-Interscience, Hoboken, pp 265–282
16. Kale LV, Bohm E, Mendes CL, Wilmarth T, Zheng G (2008) Programming petascale applications with Charm++ and AMPI. In: Bader B (ed) Petascale computing: algorithms and applications. Chapman & Hall, CRC, Boca Raton, pp 421–441

Checkpoint/Restart

- [Checkpointing](#)

Checkpointing

MARTIN SCHULZ

Lawrence Livermore National Laboratory, Livermore, CA, USA

Synonyms

[Checkpoint-recovery](#); [Checkpoint/Restart](#)

Definition

In the most general sense, Checkpointing refers to the ability to store the state of a computation in a way that

allows it to be continued at a later time without changing the computation's behavior. The preserved state is called the *Checkpoint* and the continuation is typically referred to as a *Restart*.

Checkpointing is most typically used to provide fault tolerance to applications. In this case, the state of the entire application is periodically saved to some kind of stable storage, e.g., disk, and can be retrieved in case the original application crashes due to a failure in the underlying system. The application is then restarted (or recovered) from the checkpoint that was created last and continued from that point on, thereby minimizing the time lost due to the failure.

Discussion

Checkpointing is a mechanism to store the state of a computation so that it can be retrieved at a later point in time and continued. The process of writing the computation's state is referred to as *Checkpointing*, the data written as the *Checkpoint*, and the continuation of the application as *Restart* or *Recovery*. The execution sequence between two checkpoints is referred to as a *Checkpointing Epoch* or just *Epoch*.

As discussed in section “► [Checkpointing Types](#)”, Checkpointing can be accomplished either at system level, transparently to the application (section “► [System-Level Checkpointing](#)”), or at application level, integrated into an application (section “► [Application-Level Checkpointing](#)”). While the first type is easier to apply for the end user, the latter one is typically more efficient.

While checkpointing is useful for any kind of computation, it plays a special role for parallel applications (section “► [Parallel Checkpointing](#)”), especially in the area of High-Performance Computing. With rising numbers of processors in each system, the overall system availability is decreasing, making reliable fault tolerance mechanisms, like checkpointing, essential. However, in order to apply checkpointing to parallel applications, the checkpointing software needs to be able to create globally consistent checkpoints across the entire application, which can be achieved using either coordinated (section “► [Coordinated Checkpointing](#)”) or uncoordinated (section “► [Uncoordinated Checkpointing](#)”) checkpointing protocols.

Independent of the type and the underlying system, checkpointing systems always require some kind

of storage to which the checkpoint can be saved, which is discussed in section “Checkpoint Storage Considerations”.

Checkpointing is most commonly associated with fault tolerance: It is used to periodically store the state of an application to some kind of stable storage, such that, after a hardware or operating system failure, an application can continue its execution from the last checkpoint, rather than having to start from scratch. The following entry will concentrate on this usage scenario, but will also discuss some alternate scenarios in section “Alternate Usage Scenarios”.

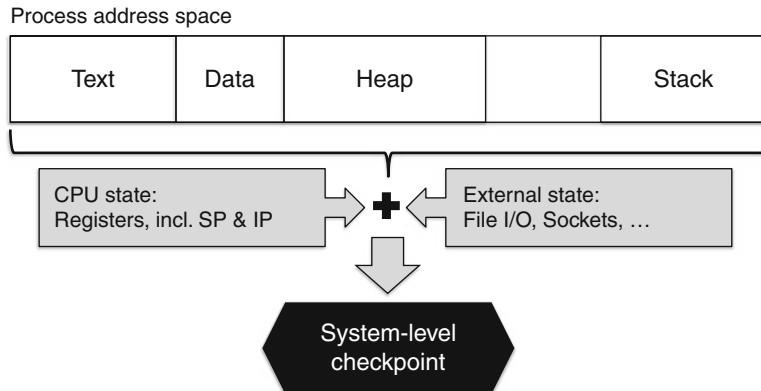
Checkpointing Types

Checkpointing can be implemented either at the system level, i.e., by the operating system or the system environment, or within the application itself.

System-Level Checkpointing

In system level checkpointing, the state of a computation is saved by an external entity, typically without the application's knowledge or support. Consequently, the complete process information has to be included in the checkpoint, as illustrated in Fig. 1. This includes not only the complete memory footprint including data segments, the heap, and all stacks, but also register and CPU state as well as open file and other resources. On restart, the complete memory footprint is restored, all file resources are made available to the process again, and then the register set is restored to its original state, including the program counter, allowing the application to continue at the same point where it had been interrupted for the checkpoint.

System-level checkpoint solutions can either be implemented inside the kernel as a kernel module or service, or at the user level. The former has the advantage that the checkpointer has full access to the target process as well as its resources. User-level checkpointers have to find other ways to gather this information, e.g., by intercepting all system calls. On the flip side, user-level schemes are typically more portable and easier to deploy, in particular in large-scale production environments with limited access for end users.



Checkpointing. Fig. 1 System-level checkpointing

Application-Level Checkpointing

The alternative to system-level checkpointing is to integrate the checkpointing capability into the actual application, which leads to application-level checkpointing solutions. In such systems, the application is augmented with the ability to write its own state into a checkpoint as well as to restart from it. While this requires explicit code inside the application and hence is no longer transparent, it gives the application the ability to decide when checkpoints should be taken (i.e., when it is a good time to write the state, e.g., when memory usage is low or no extra resources are used) and what should be contained in the checkpoint (Fig. 2). The latter enables applications to remove noncritical memory regions, e.g., temporary fields, from the checkpoint and hence reduce the checkpoint size.

To illustrate the latter point, let us consider a classical particle simulation in which the location and speed of a set of particles is updated at each iteration step based on the physical properties of the underlying system. The complete state of the computation can be represented by only the two arrays representing the coordinates and velocities of all particles in the simulation. If checkpoints are taken at iteration boundaries after the update of all arrays is complete, only these arrays need to be stored to continue the application at a later time. Any temporary array, e.g., used to compute forces between particles, as well as stack information does not need to be included. A system-level checkpoint, on the other hand, would not be able to determine which parts of the data are relevant and which are not and would have to store the entire memory segment.

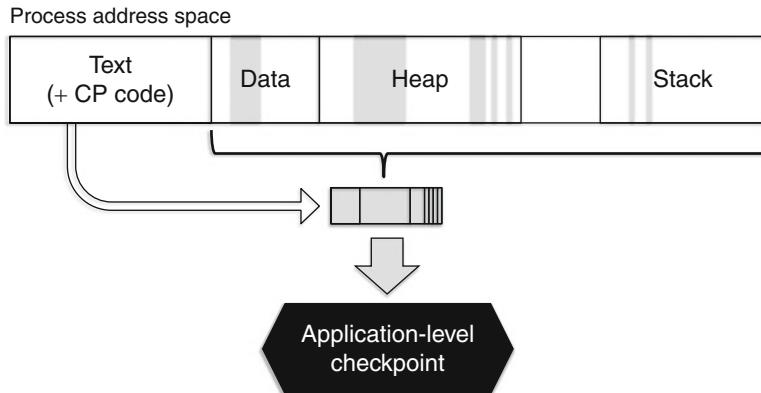
Application-level checkpointing is used in many high-performance computing applications, especially in simulation codes. They are part of the applications' base design and implemented by the programmer. Additionally, systems like SRS [1] provide toolboxes that allow users to implement application-level checkpoints in their codes on top of a simple and small API.

Trade-offs

Both approaches have distinct advantages and disadvantages. The key differences are summarized in Table 1. While system-level checkpoints provide full transparency to the user and require no special mechanism or consideration inside an application, this transparency is missing in application-level checkpointers. On the other hand, this transparency comes at the price of high implementation complexity for the checkpointing software. Not only must it be able to checkpoint the complete system state of an arbitrary process with arbitrary resources, but also has to do so at any time independent of the state the process is in.

In contrast to system-level checkpointers, application-level approaches can exploit application-specific information to optimize the checkpointing process. They are able to control the timing of the checkpoints and they can limit the data that is written to the checkpoint, which reduces the data that has to be saved and hence the size of the checkpoints.

Combining the advantages of both approaches by providing a transparent solution with the benefits of an application-level approach is still a topic of basic research. The main idea is, for each checkpoint inserted



Checkpointing. Fig. 2 Application-level checkpointing

Checkpointing. Table 1 Key differences between system- and application-level checkpointing

System-level checkpointing	Application-level checkpointing
Transparent	Integrated into the application
Implementation complexity high	Implementation complexity medium or low
System specific	Portable
Checkpoints taken at arbitrary times	Checkpoints taken at predefined locations
Full memory dump	Only save what is needed
Large checkpoint files	Checkpoint files only as large as needed

into the application (either by hand or by a compiler), to identify which variables need to be included in the checkpoint at that location, i.e., those variables that are in scope and that are actually used after the checkpoint location (and cannot be easily recomputed) [2, 3]. While this can eliminate the need to save some temporary arrays, current compiler analysis approaches are not powerful enough to achieve the same efficiency as manual approaches.

Parallel Checkpointing

Checkpointing is of special importance in parallel systems. The larger numbers of components used for a single computation naturally decreases the mean time between failures, making system faults more likely. Already today's largest systems consist of over 100,000

processing cores and systems with 1,000 to 10,000 cores are common in High-Performance Computing [4]; future architectures will have even more, as plans for machines with over a million cores have already been announced [5]. This scaling trend requires effective fault tolerance solutions for such parallel platforms and their applications.

However, checkpointing a parallel application cannot be implemented by simply replicating a sequential checkpointing mechanism to all tasks of a parallel application, since the individual checkpoints would not be coordinated and it would therefore not be possible to capture a complete snapshot of the application's state from which the computation can be restarted.

The solution of this problem depends on the type of system used. In the following, the entry discusses approaches for shared memory and message passing applications, the two dominant programming models for HPC (High-Performance Computing).

Checkpointing in Shared Memory Applications

Checkpointing for shared memory systems programmed using threads can be implemented very similar to a checkpointer for a sequential system. Since all state is shared, it is sufficient for a system-level checkpointer to suspend all threads before a checkpoint and then checkpoint the complete state of the process, including the separate call stacks and register files of all threads. No further coordination is necessary. On restart, the system has to recreate all threads that the original system had, including all resources like locks or semaphores held by the application before the

checkpoint, before loading the actual checkpoint into the context of all threads.

Application-level checkpointers typically use their knowledge about the code to insert checkpoints at locations in which only one thread is active and then can checkpoint the complete state as in a sequential program. Otherwise, a coordination between the threads and individual local checkpoints is necessary to ensure checkpoints with consistent global state (which is shared and accessible by all threads) and matching per thread call stacks (which are only accessible by the individual threads). An example for such a coordination protocol is given by Bronevetsky et al. [6], which ensures that all active threads are stopped at appropriate locations before taking a checkpoint.

Checkpointing in Message Passing Applications

The situation for message passing systems is more complicated since checkpoints from multiple processes, potentially running on different nodes, have to be taken. Therefore, a checkpointing mechanism needs to take into account which communication and interprocess dependencies exist before any checkpoint is actually committed to the system. In order to achieve this goal, an additional protocol is required that coordinates the checkpoints in one way or another.

[Figure 3](#) illustrates the two conflict scenarios that can occur without the proper coordination. All figures show the time lines of two processes P_0 and P_1 with time increasing from left to right and arrows indicating messages between the two processes. If local checkpoints are to be taken at arbitrary times and communication occurs between the individual instances of checkpoints, one of two scenarios can occur: A message is expected by a process after restart that will not be sent anymore because it was originally sent before the other sender's checkpoint ([Fig. 3a](#)); or a message should not be sent anymore after a restart since it is already been consumed by a receiver before its own checkpoint ([Fig. 3b](#)). The first type of scenario is typically referred to as a *late message*, while the second one is referred to as an *early message*.

Two main approaches exist to achieve proper checkpoint coordination between parallel processes to eliminate the conflicts caused by these two scenarios:

Coordinated Checkpointing protocols enforce a coordination at checkpoint time and hence avoid the problem before it occurs, while *Uncoordinated Checkpointing* protocols allow checkpoints at arbitrary points, but then use additional mechanisms to eliminate inconsistencies if they should occur.

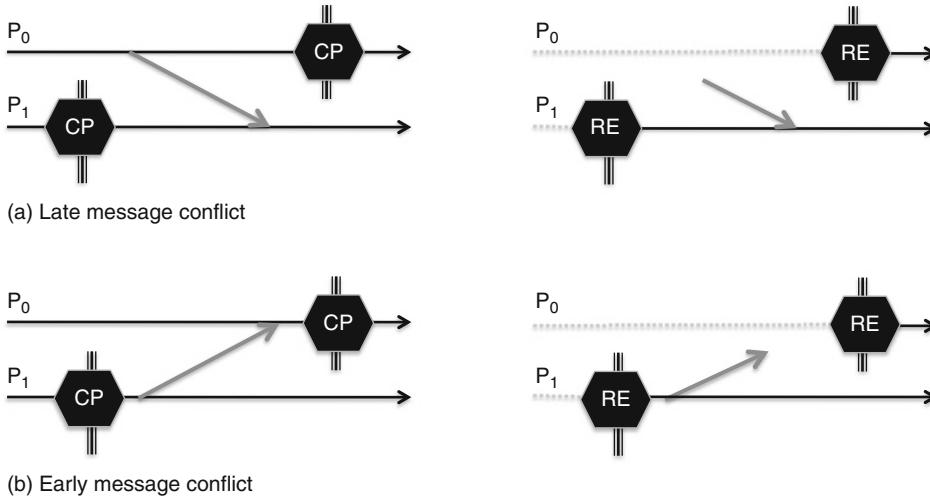
Coordinated Checkpointing

The first variant, coordinated checkpointing, controls the system in a way that all local checkpoints are taken at a point that avoids such inconsistencies. However, avoiding late messages in a general scheme without any application knowledge or global state is hard since a receiver can never know whether another process has already sent a message or not when it decides to take a checkpoint. Luckily, tracking and dealing with late message is straightforward: They can simply be buffered on the receiving task and stored at the receiver. During the restart, they are then replayed from the checkpoint instead of received from the network.

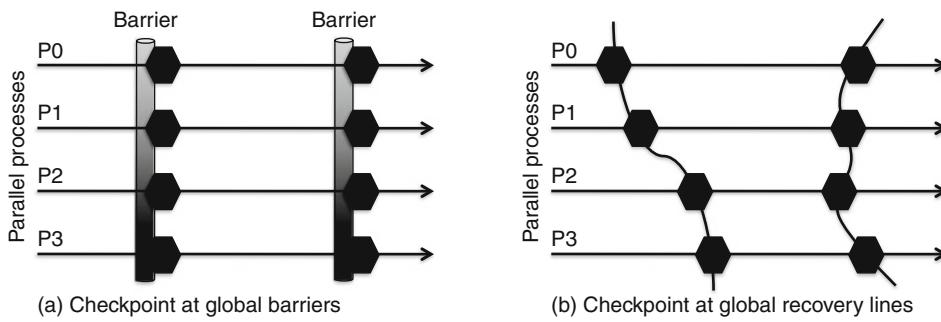
Early messages, on the other hand, are more difficult to handle. Not only do they require a global operation to avoid messages not being sent on restart, but they also require a deterministic execution after a restart to ensure that the same message content would have been sent in the early message, since the receiver has already consumed the message before the checkpoint and hence its contents is now part of the checkpointed state.

Due to these problems, most coordinated checkpointing schemes are designed in a way that prevents the occurrence of early messages. In the simplest case, as illustrated in [Fig. 4a](#), checkpoints are taking at global synchronization points, e.g., at barrier operations. This ensures that all tasks are at the same point in their executions and that no early messages can occur in the system. However, this solution is limited to codes that have natural points for global synchronization.

In the more general case, which is applicable to any irregular application, the system aims at forming a globally consistent cut through the program, which is not traversed by an early message ([Fig. 4b](#)). An algorithm for this has been introduced by Chandy and Lamport [7]. In this approach, the system forces a local checkpoint as soon as it receives a message from a process that has already started a checkpoint, but before the message is consumed. This prevents any process from receiving a message sent from a future checkpointing



Checkpointing. Fig. 3 Message conflicts caused by uncoordinated checkpoints (checkpoint shown left, restart shown right)



Checkpointing. Fig. 4 Consistency lines in coordinated checkpointing systems

epoch, since such messages would have triggered the automatic checkpoint.

The main restriction of this approach is that it dictates when local checkpoints have to be taken. This limits possible optimizations of the checkpoint location and reduces the flexibility when such a mechanism is intended to be used in an application-level scheme. Some research projects [8] have their proposed alternate solutions allowing early messages and adding the necessary information to remove their effects to the checkpoint.

Uncoordinated Checkpointing

A second approach to parallel checkpointing is the use of an uncoordinated checkpointing system. In such systems, local checkpoints are taken at arbitrary times, but then the system deploys additional mechanisms

to correct for inconsistent messages. The technique most used for this purpose is message logging. In this approach, all messages in an epoch are logged (together with the results of all nondeterministic operations and library calls) and stored together with the checkpoint. During restart, this complete message log is then used to recreate the complete global state of the application by replaying all messages and nondeterministic events that occurred between taking the individual local checkpoints.

In contrast to coordinated checkpointing, this approach requires less complexity during the initial application run since both the checkpoints and the message logging are purely local operation; no global coordination has to be executed. On the other hand, the restart of an application is significantly more complex since the shared state has to be recreated at that time. An

additional drawback is that for some applications, the message logs can be of significant size putting further pressure on the storage system.

Checkpoint Storage Considerations

Any checkpointing solution, independent of the design decisions discussed above, requires that each generated checkpoint has to be written to a storage location outside the current process, so that the computation can be restarted once the original process terminates. In the case of checkpointing for fault tolerance, this storage must also be able to survive the crash of the application or the underlying system.

Checkpoint Storage Locations

Therefore, the most common storage scenario is that checkpoints are written to disk, preferably to a remote server, since this allows the user to retrieve the checkpoint even if the initial system the application had been executed on becomes inaccessible or suffers a complete data loss. In case of a parallel checkpoint, the storage location has to receive checkpoints from all processes in the target application. While this is typically not a problem for small systems, it can lead to significant bottlenecks on systems with large numbers of nodes. In such cases, checkpoints are typically written to a parallel file system like lustre, GPFS, or PVFS, since those are designed to handle concurrent writes from a large number of processes.

Diskless Checkpointing

However, even with the use of a parallel file system, the I/O required for frequent checkpoints can be large and can, for growing numbers of nodes, start dominating the execution cost. Alternatively, checkpoints can also be stored in the memory of remote nodes, but within the cluster itself, which is referred to as *Diskless Checkpointing* and has, e.g., been demonstrated by Silvia et al. [9], Planck et al. [10], and Zheng et al. [11].

Such systems often distribute the state of a single node on multiple remote nodes using error correcting codes, such that the failure of one of the machines containing the checkpoint does not lead to the loss of the checkpoint. Diskless checkpointing eliminates the need for any I/O to a central storage facility outside the cluster and therefore reduces the time needed to store a

checkpoint. However, during the recovery, a more complex protocol is required to identify the remote storage locations of the last checkpoint and to reassemble it for a proper restart. Further, the application remains vulnerable to a failure of the entire cluster, e.g., caused by power loss. Some checkpoint solutions, like SCR [12], therefore provide mechanisms to combine in memory and on disk checkpoints by adaptively choosing the appropriate location based on the application's fault tolerance needs.

Incremental Checkpointing

A different aspect to optimizing storage are techniques to reduce the size of the checkpoints. Of particular interest for several research projects is thereby incremental checkpointing [13, 14]. In this approach, each checkpoint only stores the difference between itself and the previous checkpoint. In scenarios with frequent checkpoints or slowly evolving application, this has the potential to reduce the checkpoint storage requirements, and with that also the time it takes to store the checkpoint, significantly. On the downside, though, this approach requires the system to keep all previous checkpoints around, since these may be needed to recreate the most recent state.

Alternate Usage Scenarios

The usage scenario commonly associated with checkpointing is to provide transparent fault tolerance to the application that is being checkpointed. However, the idea of checkpointing, using the same techniques, can also be applied in different scenarios. Some of them are described in the following:

Time Sharing of Large-Scale Resources

In order to allow a fair scheduling for all users, large-scale compute centers often impose a limit on the total execution time of a single job. Large-scale applications, in particular simulation codes, however, often require significantly longer overall execution times than those dictated by these artificial limits.

In order to execute longer running jobs, they can take a checkpoint at the end of their time slot and then store this checkpoint on stable storage. When the application is granted access to a partition again, it can then read this checkpoint file and continue its execution.

This way several partition allocations, potentially even from the different machines, can be chained together.

Such checkpointing is typically implemented with the individual applications using an application-level approach. This reduces the amount of data that has to be stored and also allows for more flexibility during restart.

Migration

Similar to the above is the idea of process migration: A process is stored, removed from the system, and then either immediately or later restored, but in this case on a different node. [Figure 5](#) illustrates this further. The ability to migrate checkpoints is straightforward as long as the source and target architectures are identical, i.e., the checkpoint can be transferred and restarted without conversion.

In heterogeneous environments, however, this can be more complex. In this case, the checkpoint either needs to be stored in an architecture-independent format or the checkpoint needs to be converted from one architecture to the next. This has many implications, including the problem that heap address cannot be maintained anymore across checkpoint/restart sequences. Consequently, this option typically requires an application-level checkpointing approach or imposes limits on the applications, e.g., by only allowing a subset of a program language to be used.

Migration systems that use some kind of checkpointing exist at various granularities. For example, the Condor scheduling system [15, 16] applies it to complete jobs to achieve a better utilization of compute clusters for high-throughput computing, while systems like Charm++ [17, 18] have the ability to apply it to fine grain

parallelism and use it to checkpoint individual threads or active objects in a large-scale system to achieve load balancing.

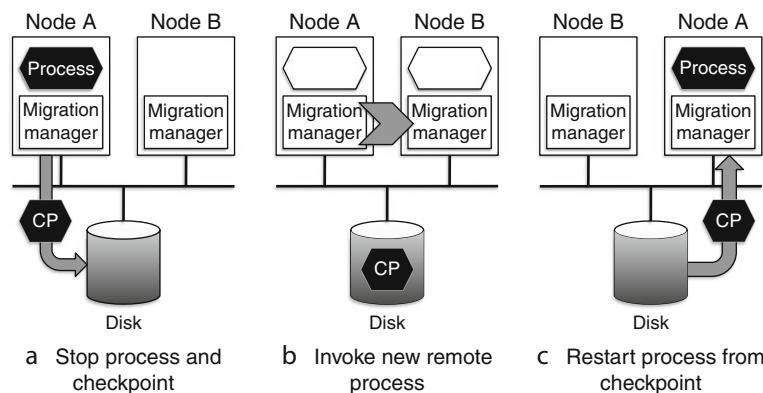
Virtual Machine Checkpointing

Virtual machines that allow the execution or emulation of an entire system on top of a (potentially different) host system have become commonplace. These systems typically allow the state of the virtual machine, which is often contained in a single file within the host system, to be saved such that the virtual machine can be interrupted and later continued. This functionality represents a system-level checkpointing, which is completely transparent. However, the target of the checkpointing is no longer a single process or computation, but rather the complete virtualized operating system image.

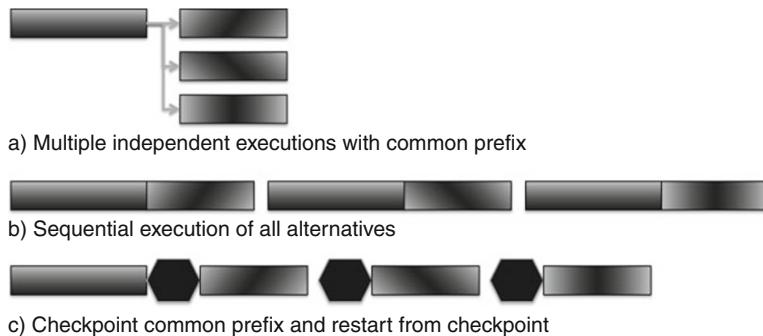
Exploration of Alternate Executions

Checkpointing systems can also improve the explore alternate execution sequences if those require a common prefix. Such scenarios can be found, e.g., in parameters studies of simulations (in which the common prefix is a warmup or initialization phase that should not be included in the actual parameter study) or in exhaustive search algorithms that require backtracking to intermediate execution points.

[Figure 6](#) illustrates this approach further: The top graphics ([Fig. 6a](#)) shows the conceptual execution with a common prefix and three alternate executions after the prefix. Without checkpointing, the complete run requires the execution of the application three times ([Fig. 6b](#)), repeating the common prefix. Using checkpointing it is possible, though, to execute the prefix



Checkpointing. [Fig. 5](#) Migrating a process using checkpointing



Checkpointing. Fig. 6 Investigating alternate executions

only once, checkpoint the state of the process, and then continue with the first execution. Once this is complete, the checkpoint can be restarted as many times as necessary to complete the remaining execution alternatives (Fig. 6c).

In contrast to traditional checkpointing, where checkpoints are written proactively and read only once, in this scenario, the application only writes a very limited set of checkpoints, but then reuses them several times.

Related Entries

- Fault Tolerance
- I/O

Bibliographic Notes and Further Reading

Elnozahy et al. provide an in-depth overview of checkpoint/restart implementations [19]. Besides that, checkpointing solutions have been implemented on several platforms, typically in the form of system-level checkpointing, in some cases, like on IBM's BlueGene line, even as part of the standard operating system stack. Also some platform-independent solutions are commercially available, one example being Librato's Availability Services (Avs) [20] (a user-level implementation). Additionally, several academic projects offer system-level checkpoint/restart solutions, e.g., LibCkpt [21] (user-level), BLCR [22] (kernel-level), CoCheck [23] (user-level for MPI and PVM applications), or Condor [15, 16] (user-level, integrated with a cluster scheduling system). Message logging-based approaches have received special attention in the past few years due to their scaling promises. Examples for the latter are

in extensions to MVAPICH-2 by Bouteiller et al. [24] or the Optimistic Message Logging approach by Wang et al. [25].

Bibliography

1. Vadhiyar S, Dongarra J (2003) SRS – a framework for developing malleable and migratable parallel software. *Parallel Process Lett* 13(2):291–312
2. Beck M, Plank JS, Kingsley G, Kingsley G (1994) Compiler-assisted checkpointing. In: Technical report CS-94-269, department of computer science, University of Tennessee, Knoxville, December 1994
3. Chung chi Jim Li, Stewart EM, Fuchs WK (1994) Compiler-assisted full checkpointing. *Pract Exper* 24(10):871–886
4. University of Mannheim, University of Tennessee, and NERSC/LBNL. TOP500 Supercomputing Sites. <http://www.top500.org/>
5. Lawrence Livermore National Laboratory. NNSA awards IBM contract to build next generation supercomputer, press release. https://publica_airs.llnl.gov/news/newsreleases/2009/NR-09-02-01.html. Accessed Feb 2009
6. Bronevetsky G, Pingali K, Stodghill P (2006) Experimental evaluation of application-level checkpointing for OpenMP programs. In: International conference on supercomputing (ICS), Queensland, June 2006
7. Chandy M, Lamport L (1985) Distributed snapshots: determining global states of distributed systems. *ACM Transact Comput Syst* 3(1):63–75
8. Schulz M, Bronevetsky G, Fernandes R, Marques D, Pingali K, Stodghill P (2004) Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for MPI programs. In: Proceedings of IEEE/ACM supercomputing '04, Washington, DC, November 2004
9. Silva LM, Silva JG (1998) An experimental study about diskless checkpointing. *EUROMICRO Conf* 1:10395
10. Plank JS, Li K, Puening MA (1998) Diskless checkpointing. *IEEE Trans Parallel Distrib Syst* 9(10):972–986
11. Zheng G, Shi L, Kale LV (2004) FTC-Charm++: an In-Memory checkpoint-based fault tolerant runtime for Charm++ and MPI.

- In: 2004 IEEE international conference on cluster computing, pp 93–103, San Diego, September 2004
12. Moody A, Bronevetsky G, Mohror K, de Supinski BR (2010) Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: Proceedings of IEEE/ACM supercomputing '10, New Orleans, LA, 2010
 13. Agarwal S, Garg R, Gupta MS, Moreira JE (2004) Adaptive incremental checkpointing for massively parallel systems. In: ICS '04: proceedings of the 18th annual international conference on supercomputing. ACM, New York, pp 277–286
 14. Sancho JC, Petrini F, Johnson G, Fernndez J, Frachtenberg E (2004) On the feasibility of incremental checkpointing for scientific computing. Parallel Distrib Process Symp Int 1:58b
 15. Litzkow JBM, Tannenbaum T, Livny M (1997) Checkpoint and migration of UNIX processes in the condor distributed processing system. In: Technical report 1346, University of Wisconsin, Madison, 1997
 16. Condor. <http://www.cs.wisc.edu/condor/manual>
 17. CHARM research group. <http://charm.cs.uiuc.edu/>
 18. Kale LV, Krishnan S (1993) CHARM++: a portable concurrent object oriented system based on C++. Parallel Process Lett 28(10):91–108
 19. Elnozahy M, Alvisi L, Wang YM, Johnson DB (1996) A survey of rollback-recovery protocols in message passing systems. In: Technical report CMU-CS-96-181, school of computer science, Carnegie Mellon University, Pittsburgh, October 1996
 20. Librato. Availability Services (AvS). <http://www.librato.com/products/availability.services>
 21. Plank JS, Beck M, Kingsley G, Li K (1994) Libckpt: transparent checkpointing under UNIX. In: Technical report UT-CS-94-242, Department of Computer Science, University of Tennessee, Princeton University
 22. Duell J The design and implementation of Berkeley lab's linux checkpoint/restart. <http://www.nersc.gov/research/FTG/checkpoint/reports.html>
 23. Stellner G (1996) CoCheck: checkpointing and process migration for MPI. In: Proceedings of the 10th international parallel processing symposium (IPPS '96), Honolulu, 1996
 24. Bouteiller A, Cappello F, Herault T, Krawezik G, Lemarnier P, Magniette F (2003) MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In: Proceedings of IEEE/ACM supercomputing '03, Phoenix, November 2003
 25. Wang YM, Fuchs WK (1992) Optimistic message logging for independent checkpointing in message-passing systems. In: Proceedings of the 11th symposium on reliable distributed systems, Houston, October 1992, pp 147–154

CHiP Architecture

► Blue CHiP

CHiP Computer

► Blue CHiP

Cholesky Factorization

► Dense Linear System Solvers
► Sparse Direct Methods

Cilk

CHARLES E. LEISERSON
Massachusetts Institute of Technology, Cambridge,
MA, USA

Synonyms

Cilk-1; Cilk-5; Cilk++; Cilk plus

Definition

Cilk (pronounced “silk”) is a linguistic and runtime technology for algorithmic multithreaded programming originally developed at MIT. The philosophy behind Cilk is that a programmer should concentrate on structuring her or his program to expose parallelism and exploit locality, leaving Cilk’s runtime system with the responsibility of scheduling the computation to run efficiently on a given platform. The Cilk runtime system takes care of details like load balancing, synchronization, and communication protocols. Cilk is algorithmic in that the runtime system guarantees efficient and predictable performance. Important milestones in Cilk technology include the original Cilk-1, which provided a provably efficient work-stealing runtime support but

Checkpoint-Recovery

► Checkpointing

little linguistic support; the later Cilk-5, which provided simple linguistic extensions for multithreading to ANSI C; the commercial Cilk++, which extended the Cilk model to C++ and introduced “reducer hyper-objects” as an efficient means for resolving races on nonlocal variables; and Intel Cilk Plus, which provided transparent interoperability with legacy C/C++ binary executables.

Discussion

Introduction

Cilk technology has developed and evolved over more than 15 years since its origin at MIT. Key releases of Cilk include Cilk-1 [11, 14, 53], Cilk-NOW [11, 18], Cilk-5 [38, 40, 76, 81], JCilk [29, 30, 59], Cilk++ [25, 50, 62], Intel Cilk Plus [51], and Cilk-M [60]. Section “► A Brief History of Cilk Technology” overviews the history of Cilk technology. Some of the Cilks were more runtime systems than full-blown parallel languages, but it is the simplicity of the more linguistically oriented Cilks – Cilk-5, JCilk, Cilk++, and Cilk Plus – which makes the technology compelling. This article will focus on Cilk++, whose linguistics are both full featured and simple.

Cilk++ is a *faithful* linguistic extension of the serial C++ programming language [80], which means that parallel code retains its serial semantics when run on one processor. The Cilk++ extensions to C++ consist of just three keywords, which can be understood from an example. Figure 1 shows a Cilk++ program adapted from http://en.wikibooks.org/wiki/Algorithm_implementation/Sorting/Quicksort, which implements the quicksort algorithm [26, Chap. 7]. Observe that the program would be an ordinary C++ program if the keywords `cilk_spawn` and `cilk_sync` were elided and `cilk_for` replaced by `for`. The program so modified is called the *serialization* of the Cilk++ program. (The term *serial elision* was used in earlier Cilks, because all the keywords could simply be elided.) One of the things that makes Cilk simple is that the serialization of a parallel code always provides a legal semantics for the parallel code that can execute on a single processor.

Parallel work is created when the keyword `cilk_spawn` precedes the invocation of a function.

The semantics of spawning differ from a C++ function (or method) call only in that the parent can continue to execute in parallel with the child, instead of waiting for the child to complete as is done in C++. The scheduler in the Cilk++ runtime system takes the responsibility of scheduling the spawned functions on the individual processor cores of the multicore computer.

A function cannot safely use the values returned by its children until it executes a `cilk_sync` statement. The `cilk_sync` statement is a local “barrier,” not a global one as, for example, is used in message-passing programming [82, 83]. In the quicksort example, a `cilk_sync` statement occurs on line 14 before the function returns to avoid the anomaly that would occur if the preceding calls to `qsort` were scheduled to run in parallel and did not complete before the return, thus leaving the vector to be sorted in an intermediate and inconsistent state.

In addition to explicit synchronization provided by the `cilk_sync` statement, every Cilk function syncs implicitly before it returns, thus ensuring that all of its children terminate before it does. Thus, for this example, the `cilk_sync` before the return is technically unnecessary.

Cilk++ provides faithful extensions of other C++ language features. It provides full support for C++ exceptions. When exceptions are thrown in parallel, the one that would occur first in a serial execution is the one that executes the `catch` block. Loops can be parallelized by simply replacing the keyword `for` with the keyword `cilk_for` keyword, which allows all iterations of the loop to operate in parallel. Within the main routine from Fig. 1, for example, the loop starting on line 26 fills the array in parallel with “random” numbers. In addition, Cilk++ includes a library for mutual-exclusion (mutex) locks. Locking tends to be used much less frequently than in other parallel environments, such as Pthreads [49], because all protocols for control synchronization are handled by the Cilk++ runtime system. Cilk++ also provides a powerful “hyperobject” library, which allows races on nonlocal variables to be mitigated without lock contention or restructuring of code. Cilk++ provides tool support in the form of the Cilscreen race detector, which guarantees to find race bugs in ostensibly deterministic code,

```

1 // Parallel quicksort
2 using namespace std;
3
4 #include <algorithm>
5 #include <iterator>
6 #include <functional>
7
8 template <typename T>
9 void qsort(T begin, T end) {
10     if (begin != end) {
11         T middle = partition(begin, end, bind2nd(less<typename
12             iterator_traits<T>::value_type>(), *begin));
13         cilk_spawn qsort(begin, middle);
14         qsort(max(begin + 1, middle), end);
15         cilk_sync;
16     }
17 }
18 // Simple test code:
19 #include <iostream>
20 #include <cmath>
21
22 int cilk_main() {
23     int n = 100;
24     double a[n];
25
26     cilk_for (int i = 0; i < n; ++i) {
27         a[i] = sin((double) i);
28     }
29
30     qsort(a, a + n);
31     copy(a, a + n, ostream_iterator<double>(cout, "\n"));
32
33     return 0;
34 }
```

Cilk. Fig. 1 Parallel quicksort implemented in Cilk++

and the Cilkview scalability analyzer, which extrapolates speedups based on the algorithmic complexity measures of “work” and “span.”

The remainder of this article is organized as follows. Section “A Brief History of Cilk Technology” overviews the evolution of Cilk technology. Section “The Dag Model for Multithreading” provides a brief tutorial on the theory of parallelism. Section “►Runtime System” describes the performance guarantees of Cilk++’s “work-stealing” scheduler, illustrates the Cilkview scalability analyzer, and overviews how the scheduler operates. section “►Race Detection” briefly describes the Cilkscreen race-detection tool, and section “►Reducer Hyperobjects” explains Cilk++’s “hyperobject” technology. Finally, section “►Conclusion” provides some concluding remarks.

A Brief History of Cilk Technology

This section overviews the development of the Cilk technology starting from its origin at the MIT Laboratory for Computer Science under the direction of Professor Charles E. Leiserson. The four MIT Ph.D. theses [11, 38, 53, 76] contain more detailed descriptions of the foundation and early history of Cilk.

The Origins of Cilk

The first implementation of Cilk arose from three separate projects at MIT in 1993. The first project was theoretical work [15, 16] on scheduling multi-threaded applications. The second was StarTech [52, 55, 56], a parallel chess program built to run on the Thinking Machines Corporation’s Connection Machine Model CM-5 Supercomputer [63]. The third project

was PCM/Threaded-C [45], a C-based package for scheduling continuation-passing-style threads on the CM-5.

In April 1994 the three projects were combined and christened Cilk. (The name Cilk is not an acronym, but an allusion to nice threads (silk) and to the C programming language (hence Cilk and not Silk). One of Leiserson’s graduate students commented that Cilk is a language of the C ilk. When his graduate students held an acronym contest for Cilk, the winner was “Charles’s Idiotic Linguistic Kludge.”) The team began by implementing a new parallel chess program. Unlike StarTech, which intertwined the search code and the scheduler, the new chess program implemented its search completely on top of a general-purpose runtime system that incorporated a provably efficient work-stealing scheduler. At the end of June, the *Socrates chess program was entered into the 1994 ACM International Chess Championship, where, running on a 512-node CM-5, it finished third. The “Cilk-1” system [14] itself was released in September 1994. A notable branch of the Cilk-1 codebase was Cilk-NOW [11, 18], which provided an adaptively parallel and fault-tolerant network-of-workstations implementation.

Simple Linguistics

The Cilk-2 release in May 1995 [68] featured full type-checking, supported all of ANSI C in its C-language subset, and offered call-return semantics for writing multithreaded procedures. Thus, instead of having to “wire together” continuations, the programmer could simply insert the `spawn` keyword before a function call and execute a `sync` statement to ensure that all spawned subroutines had completed (as described with `cilk_spawn` and `cilk_sync` in section “Introduction”).

The runtime system was made more portable by replacing the general continuation-passing mechanism with a continuation mechanism based on Duff’s device [84]. The new continuation mechanism greatly simplified the runtime system, which allowed the base release to support several architectures other than the CM-5.

Cilk-3, which was released in October 1995, featured an implementation of “dag-consistent” distributed shared memory [12, 13, 41, 53]. With this addition of shared memory, Cilk could be applied to solve a much wider class of applications. Dag-consistency is a weak but

nonetheless useful consistency model, and its relaxed semantics allows for an efficient, low-overhead software implementation.

Optimization and Enhancement

With the Cilk-4 release in June 1996, the authors of Cilk changed their primary development platform from the distributed-memory CM-5 to the shared-memory Sun Microsystems SPARC SMP. The compiler and runtime system were completely reimplemented, eliminating continuation-passing as the basis of the scheduler, and instead embedding scheduling decisions directly into the compiled code. Instead of stealing children, as in the earlier Cilk releases, Cilk-4 adopted the “lazy-task-creation” strategy [70] of stealing parent continuations. The overhead to spawn a parallel thread in Cilk-4 was typically less than three times the cost of an ordinary C procedure call, and so Cilk-4 programs “scaled down” to run on one processor with nearly the efficiency of analogous C programs.

Cilk-4 provided two new language features to support speculative parallelism, so that applications such as computer chess [28] could be more easily programmed. The keyword `inlet` specified that an internal function could be invoked by a returning child to execute code on its parent’s frame. The keyword `abort` abruptly terminated subcomputations that had already been spawned. These mechanisms allowed a programmer to speculate that a subcomputation would be worthwhile to execute in parallel, but abort it efficiently if the subcomputation turned out to be superfluous.

For Cilk-5 [40], which was released in March 1997, the runtime system was rewritten to be more flexible and portable. Cilk-5.0 could use operating system threads as well as processes to implement the individual Cilk “workers” that schedule Cilk threads. The Cilk-5.2 release included a debugging tool called the Nondeterminator [23, 24, 35, 79], which allowed Cilk programmers to localize data-race bugs in their code. With the Cilk-5.3 release, Cilk graduated from research prototype to real-world tool intended to be used by programmers who are not necessarily parallel-processing experts. Many improvements were made to the runtime system to make it faster, more portable, and more maintainable, but the basic language has remained largely the same.

A Cilk for Java

Up to this point, the Cilk technology was grounded in the C programming language. JCilk [30, 59], however, marked an excursion into Java [8], largely to explore how exception mechanisms should interoperate with the Cilk **spawn** and **sync** primitives. Specifically, JCilk defined semantics for exceptions that are consistent with the existing semantics of Java’s `try` and `catch` constructs, but which handle concurrency in spawned methods. JCilk extends Java’s exception semantics to allow exceptions to be passed from a spawned method to its parent in a natural way that obviates the need for Cilk-5’s `inlet` and `abort` constructs. This extension is “faithful” in that it obeys Java’s ordinary serial semantics when executed on a single processor. When executed in parallel, however, an exception thrown by a JCilk computation signals its sibling computations to abort, which yields a clean semantics in which only a single exception from the enclosing `try` block is handled.

The Multicore Era

During most of the evolution of Cilk technology, a multiprocessor with n processors typically cost more than n times the cost of a single processor. Moreover, since clock frequency was increasing at about 30% per year, software vendors preferred to wait for more performance than refactor their codebases to support parallel processing. Thus, parallel computing, with or without Cilk, was very much a niche business. Around 2003, however, the trend of ever-increasing clock frequency hit a brick wall. Although the density of integrated circuits continued to double about every 18 months, the heat dissipated by transistor switching reached its physical limit. Vendors of microprocessor chips responded by placing multiple processing cores on a single chip, ushering in the era of multicore computing.

In September 2006, responding to the multicore trend, MIT spun out the Cilk technology to Cilk Arts, Inc., a venture-funded start-up founded by technical leaders Charles E. Leiserson and Matteo Frigo, together with Stephen Lewin-Berlin and Duncan C. McCallum. Although Cilk Arts licensed the historical Cilk codebase from MIT, it developed an entirely new codebase for a C++ product aptly named Cilk++ [25, 62], which was released in December 2008 for the Windows Visual Studio and Linux/gcc compilers.

Cilk++ improved upon the original MIT Cilk in several ways. The linguistic distinction between Cilk functions and C/C++ functions was lessened, allowing C++ “call-backs” to Cilk code, as long as the C++ code was compiled with the Cilk++ compiler. (This distinction was later removed altogether by Intel Cilk Plus.) The **spawn** and **sync** keywords were renamed `cilk_spawn` and `cilk_sync` to avoid naming conflicts. Loops were parallelized by simply replacing the keyword `for` with the keyword `cilk_for` keyword, which allows all iterations of the loop to operate in parallel. Cilk++ provided full support for C++ exceptions. Cilk++ also introduced “reducer hyperobjects” (see section “Reducer Hyperobjects”), which allow races on nonlocal variables to be mitigated without lock contention or restructuring of code.

The Cilk++ toolkit included the Cilkscreen race-detection tool which guarantees to find race bugs in ostensibly deterministic code. It also included the Cilkview scalability analyzer [47], a software tool for profiling, estimating scalability, and benchmarking multithreaded Cilk++ applications.

Cilk Arts was sold to Intel Corporation in July 2009, which continued developing the technology. In September 2010, Intel released its ICC compiler with Intel Cilk Plus [51]. The product included Cilk support for C and C++, and the runtime system provided transparent integration with legacy binary executables.

Research at MIT

Research on Cilk technology has also continued at MIT and includes the following contributions:

- An adaptive scheduler for multiple Cilk jobs that uses parallelism feedback to minimize wasted processor cycles while guaranteeing fairness among the jobs [3]
- The design of a work-stealing runtime-system, called CWSTM, that supports transactional memory, where transactions can contain both nested parallelism and nested transactions [2]
- A library for performing parallel sparse matrix-vector and matrix-transpose-vector multiplication using a novel matrix layout called **compressed sparse blocks** [21]
- The Helper library, which supports “helper” locks in which a thread that cannot acquire a lock, rather

than blocking, helps to complete the parallel work in the critical region protected by the lock [4]

- The Rabbit library for efficiently executing task graphs with arbitrary acyclic dependencies [5]
- A fast breadth-first search algorithm and method for analyzing nondeterministic Cilk programs that incorporate reducer hyperobjects [64]
- The research prototype Cilk-M, which uses memory mapping to solve the “cactus-stack problem,” an interoperability problem with legacy binary executables [60] (see also section “►Runtime System”)

Recognition

Over the years, Cilk technology has influenced many other non-Cilk concurrency platforms, including Sun Microsystems’ Fortress [6], University of Texas’s Hood [17], Java’s Fork/Join Framework [58], Microsoft’s Task Parallel Library (TPL) [61] and PPL [66], Intel’s Threading Building Blocks (TBB) [77], and IBM’s X10 [22].

Cilk technology has garnered many awards, including the following:

- First Prize in the 1998 *International Conference on Functional Programming*’s Programming Contest for Cilk Pousse, where Cilk was cited as “the superior programming tool of choice for discriminating hackers” [1]
- First Prize in the 2006 HPC Challenge Class 2 (Productivity) competition, where Cilk was cited for “Best Overall Productivity” [57]
- The 2008 award by ACM SIGPLAN for Most Influential 1998 PLDI Paper for [40]
- The 2009 award by the ACM *Symposium on Parallelism in Algorithms and Architectures* for Best Paper for [39]

In addition, the Cilk-based chess programs StarTech, *Socrates, and Cilkchess have won numerous prizes in international computer-chess competitions.

The Dag Model for Multithreading

The Cilk++ runtime system contains a provably efficient work-stealing scheduler [16, 40], which scales application performance linearly with processor cores, as long as the application exhibits sufficient parallelism (and the processor architecture provides sufficient memory bandwidth). Thus, to obtain good performance, the programmer needs to know what it means for his or

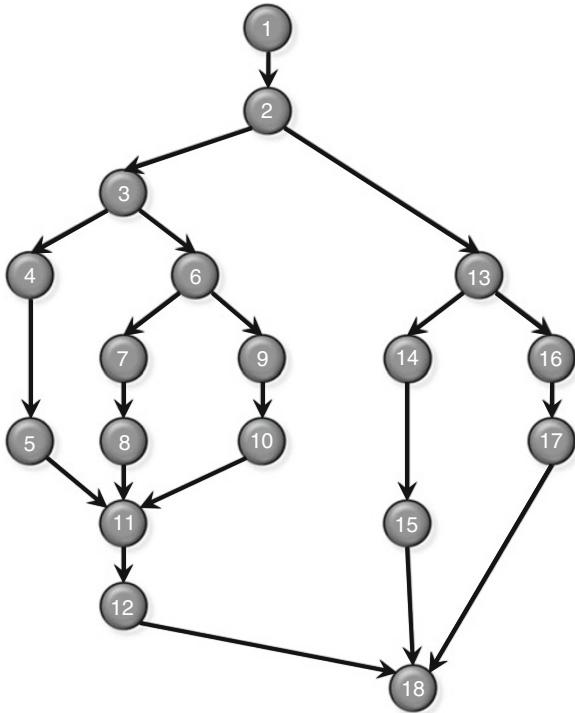
her application to exhibit sufficient parallelism. Before describing the Cilk++ runtime system, it is helpful to understand something about the theory of parallelism.

Many discussions of parallelism begin with Amdahl’s Law [7], originally proffered by Gene Amdahl in 1967. Amdahl made what amounts to the following observation. Suppose that 50% of a computation can be parallelized and 50% cannot. Then, even if the 50% that is parallel were run on an infinite number of processors, the total time is cut at most in half, leaving a speedup of at most 2. In general, if a fraction p of a computation can be run in parallel and the rest must run serially, Amdahl’s Law upper-bounds the speedup by $1/(1-p)$.

Although Amdahl’s Law provides some insight into parallelism, it does not *quantify* parallelism, and thus it does not provide a good understanding of what a concurrency platform such as Cilk++ should offer for multicore application performance. Fortunately, there is a simple theoretical model for parallel computing which provides a more general and precise quantification of parallelism that subsumes Amdahl’s Law. This “dag model of multithreading” [15, 16] provides a general and precise quantification of parallelism based on the theory developed by Graham [44]; Brent [19]; Eager, Zahorjan, and Lazowska [33]; and Blumofe and Leiserson [15, 16]. Tutorials on the dag model can be found in [26, Ch. 27] and [62].

The **dag model of multithreading** views the execution of a multithreaded program as a set of vertices called **strands** – sequences of serially executed instructions containing no parallel control – with graph edges indicating ordering dependencies between strands, as illustrated in Fig. 2. (The literature sometimes uses the term “Cilk thread” for “strand.”) A strand x **precedes** a strand y , denoted $x \prec y$, if x must complete before y can begin. If neither $x \prec y$ nor $y \prec x$, the strands are in **parallel**, denoted by $x \parallel y$. Figure 2, for example, has $1 \prec 2$, $6 \prec 8$, and $4 \parallel 9$. A strand can be as small as a single instruction, or it can represent a longer chain of serially executed instructions. A **maximal strand** is one that cannot be included in a longer strand. A maximal strand can be diced into a series of smaller strands in any manner that is convenient.

The dag model of multithreading can be interpreted in the context of the Cilk++ programming model. Normal serial execution of one strand after another creates a **serial edge** from the first strand to the next.



Cilk Fig. 2 A dag representation of a multithreaded execution. Each vertex is a strand. Edges represent ordering dependencies between instructions

A `cilk_spawn` of a function creates two dependency edges emanating from the instruction immediately before the `cilk_spawn`: the ***spawn edge*** goes to the strand containing the first instruction of the spawned function, and the ***continuation edge*** goes to the strand containing the first instruction after the spawned function. A `cilk_sync` creates a ***return edge*** from the strand containing the final instruction of each spawned function to the strand containing the instruction immediately after the `cilk_sync`. A `cilk_for` can be viewed as parallel divide-and-conquer recursion using `cilk_spawn` and `cilk_sync` over the iteration space.

The dag model admits two natural measures that allow parallelism to be defined precisely and provide important bounds on performance and speedup.

The Work Law

The first measure is ***work***, which is the total time spent in all the strands. Assuming for simplicity that it takes

unit time to execute a strand, the work for the example dag in Fig. 2 is 18.

A simple notation makes things more precise. Let T_P be the fastest possible execution time of the application on P processors. Since the work corresponds to the execution time on 1 processor, it is denoted by T_1 . One reason that work is an important measure is because it provides a lower bound on P -processor execution time:

$$T_P \geq T_1/P. \quad (1)$$

This ***Work Law*** holds, because in this simple theoretical model, each processor executes at most 1 instruction per unit time, and hence P processors can execute at most P instructions per unit time. Thus, with P processors, doing all the work requires at least T_1/P time.

One can interpret the Work Law Eq. (1) in terms of the ***speedup*** on P processors, which is just T_1/T_P . The speedup tells how much faster the application runs on P processors than on 1 processor. Rewriting the Work Law yields $T_1/T_P \leq P$, which is to say that the speedup on P processors can be at most P . If the application obtains speedup P (which is the best one can do in this simple theoretical model), the application exhibits ***linear speedup***. If the application obtains speedup greater than P (impossible in the model due to the Work Law, but possible in practice due to caching and other processor effects), the application exhibits ***superlinear speedup***.

The Span Law

The second measure is ***span***, which is the maximum time to execute along any path of dependencies in the dag. Assuming that it takes unit time to execute a strand, the span of the dag from Fig. 2 is 9, which corresponds to the path $1 \prec 2 \prec 3 \prec 6 \prec 7 \prec 8 \prec 11 \prec 12 \prec 18$. This path is sometimes called the ***critical path*** of the dag, and span is sometimes referred to in the literature as critical-path length. Since the span is the theoretically fastest time the dag could be executed on a computer with an infinite number of processors (assuming no overheads for communication, scheduling, etc.), it can be denoted by T_∞ . Like work, span also provides a bound on P -processor execution time:

$$T_P \geq T_\infty. \quad (2)$$

This ***Span Law*** arises for the simple reason that a finite number of processors cannot outperform an infinite number of processors, because the infinite-processor

machine could just ignore all but P of its processors and mimic a P -processor machine exactly.

Parallelism

The **parallelism** is defined as the ratio of work to span, or T_1/T_∞ . Parallelism can be viewed as the average amount of work along each step of the critical path. Moreover, perfect linear speedup cannot be obtained for any number of processors greater than the parallelism T_1/T_∞ . To see why, suppose that $P > T_1/T_\infty$, in which case the Span Law Eq. (2) implies that the speedup satisfies $T_1/T_P \leq T_1/T_\infty < P$. Since the speedup is strictly less than P , it cannot be perfect linear speedup. Another way to see that the parallelism bounds the speedup is to observe that, in the best case, the work is distributed evenly along the critical path, in which case the amount of work at each step is the parallelism. But, if the parallelism is less than P , there isn't enough work to keep P processors busy at every step.

As an example, the parallelism of the dag in Fig. 2 is $18/9 = 2$. Thus, executing it with more than two processors would be wasteful of cycles and yield diminishing returns, since the additional processors will be surely starved for work.

As a practical matter, many problems admit considerable parallelism. For example, matrix multiplication of 1000×1000 matrices is highly parallel, with a parallelism in the millions. Many problems on large irregular graphs, such as breadth-first search, generally exhibit parallelism on the order of thousands. Sparse-matrix algorithms can often exhibit parallelism in the hundreds.

Upper Bounds on Speedup

The Work and Span Laws engender two important upper bounds on speedup. The Work Law implies that the speedup on P processors can be at most P :

$$T_1/T_P \leq P. \quad (3)$$

The Span Law dictates that speedup cannot exceed parallelism:

$$T_1/T_P \leq T_1/T_\infty. \quad (4)$$

Runtime System

Although optimal multiprocessor scheduling is known to be NP-complete [42], Cilk++'s runtime system employs a “work-stealing” scheduler [16, 40] that

achieves provably tight bounds. An application with sufficient parallelism can rely on the Cilk++ runtime system to dynamically and automatically exploit an arbitrary number of available processor cores near optimally. Moreover, on a single core, typical Cilk++ programs run with negligible overhead (often 2% or less) when compared with its C++ serialization.

Performance Bounds

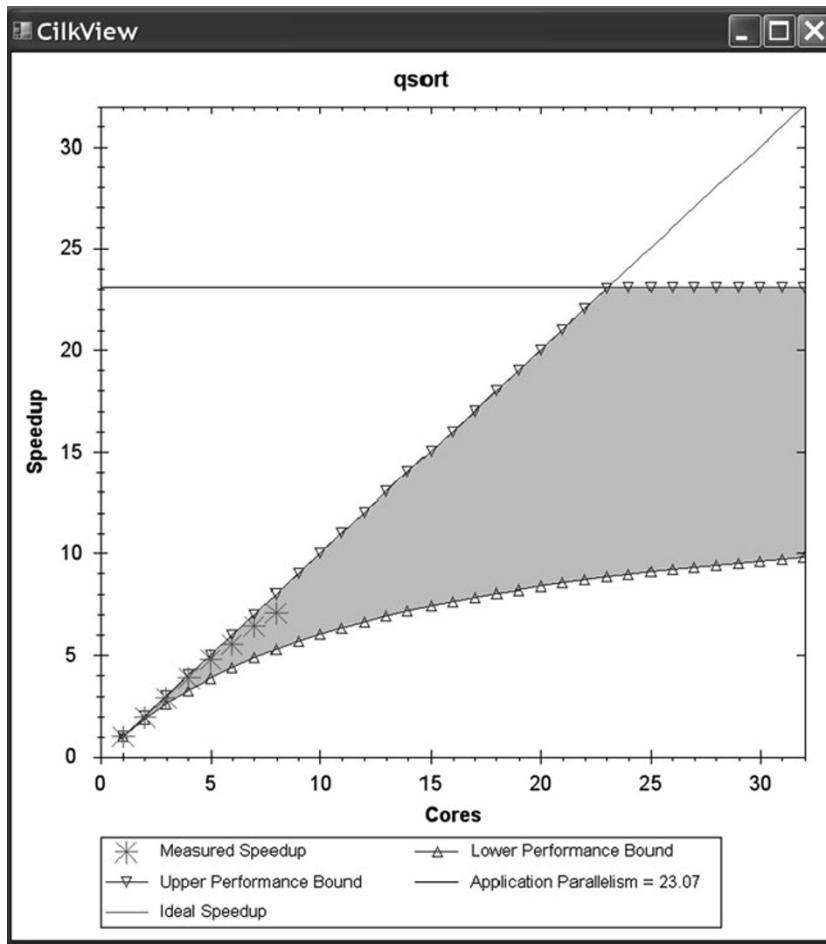
Specifically, for an application with T_1 work and T_∞ span running on a computer with P processors, the Cilk++ work-stealing scheduler achieves expected running time:

$$T_P \leq T_1/P + O(T_\infty). \quad (5)$$

If the parallelism T_1/T_∞ exceeds the number P of processors by a sufficient margin, this bound (proved in [16]), guarantees near-perfect linear speedup. To see why, assume that $T_1/T_\infty \gg P$, or equivalently, that $T_\infty \ll T_1/P$. Thus, in Inequality (5), the T_1/P term dominates the $O(T_\infty)$ term, and thus the running time is $T_P \approx T_1/P$, leading to a speedup of $T_1/T_P \approx P$.

The Cilkview scalability analyzer allows a programmer to determine the work and span of an application. Figure 3 shows the output of this tool running the quicksort program from Fig. 1 on 10 million numbers. The upper bound on speedup provided by the Work Law corresponds to the line of slope 1, and the upper bound provided by the Span Law corresponds to the horizontal line at 21.31. The performance analysis tool also provides an estimated lower bound on speedup – the lower curve in the figure – based on **burdened parallelism**, which takes into account the estimated cost of scheduling. Although quicksort seems naturally parallel, one can show that the expected parallelism for sorting n numbers is only $O(\lg n)$. Practical sorts with more parallelism exist, however. See [26, Chap. 27] for more details.

In addition to guaranteeing performance bounds, the Cilk++ runtime system also provides bounds on stack space. Specifically, on P processors, a Cilk++ program consumes at most P times the stack space of a single-processor execution. Specifically, let S_P denote the stack space required by a P -processor execution of a given Cilk++ application. Thus, S_1 is the stack space required by the C++ serialization of a Cilk++ program.



Cilk. Fig. 3 Parallelism profile of quicksort produced by the Cilkview scalability analyzer

The space guarantee of Cilk++’s work-stealing scheduler is

$$S_P \leq S_I P.$$

For specific applications, tighter bounds on space can be shown [12]. Work-stealing schedulers that provide better space bounds at the expense of higher communication are also known [10, 71].

To illustrate why a space bound is important, consider the following simple code fragment:

```
for (int i=0; i<1000*1000*1000; ++i) {
    cilk_spawn foo(i);
}
cilk_sync;
```

This code conceptually creates one billion invocations of `foo` that operate logically in parallel. Executing

on one processor, however, this Cilk++ code uses no more stack space than a serial C++ execution, that is, the call depth is of whichever invocation of `foo` requires the deepest stack. On two processors, it requires at most twice this space, and so on. This guarantee contrasts with the lack of a guarantee by more naive schedulers. For example, some schedulers would schedule this code fragment by creating a work queue of one billion tasks, one for each iteration of the subroutine `foo`, before executing even the first iteration, thus suffering exorbitant memory use.

Work Stealing

Cilk++’s work-stealing scheduler operates as follows. When the runtime system starts up, it allocates as many operating-system threads, called *workers*, as there are

processors (although the programmer can override this default decision). Each worker's stack operates like a work queue. When a subroutine is spawned, the subroutine's activation frame containing its local variables is pushed onto the bottom of the stack. When it returns, the frame is popped off the bottom. Thus, in the common case, Cilk++ operates just like C++ and imposes little overhead.

When a worker runs out of work, however, it becomes a *thief* and “steals” the top frame from another *victim* worker's stack. Thus, the stack is in fact a double-ended queue, with the worker operating on the bottom and thieves stealing from the top. This strategy has the great advantage that all communication and synchronization is incurred only when a worker runs out of work. If an application exhibits sufficient parallelism, one can prove mathematically [16, 40] that stealing is infrequent, and thus the cost of communication and synchronization to effect a steal is negligible.

The dynamic load-balancing capability provided by the Cilk++ runtime system adapts well in real-world multiprogrammed computing environments. If a worker becomes descheduled by the operating system (for example, because another application starts to run), the work of that worker can be stolen away by other workers. Thus, Cilk++ programs tend to “play nicely” with other jobs on the system.

Cilk++'s runtime system also makes Cilk++ programs *performance-composable*. Suppose that a programmer develops a parallel library in Cilk++. That library can be called not only from a serial program or the serial portion of a parallel program, it can be invoked multiple times in parallel and continue to exhibit good speedup. In contrast, some concurrency platforms constrain library code to run on a given number of processors, and if multiple instances of the library execute simultaneously, they end up thrashing as they compete for processor resources.

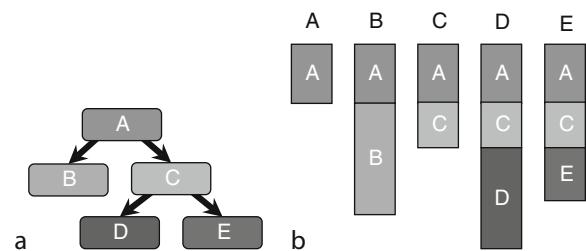
Stacks and Cactus Stacks

An execution of a serial Algol-like language, such as C [54] or C++ [80], can be viewed as a “walk” of an *invocation tree*, which dynamically unfolds during execution and relates function instances by the “calls” relation: If function instance A calls function instance B, then A is a *parent* of the *child* B in the invocation tree. Such serial languages admit a simple array-based stack for

allocating function activation frames. When a function is called, the stack pointer is advanced, and when the function returns, the original stack pointer is restored. This style of execution is space efficient, because all the children of a given function can use and reuse the same region of the stack. The compact linear-stack representation is possible only because in a serial language, a function has at most one extant child function at any time.

The notion of an invocation tree can be extended to include spawns, as well as calls, but unlike the serial walk of an invocation tree, a parallel execution unfolds the invocation tree more haphazardly and in parallel. Since multiple children of a function may exist simultaneously, a linear-stack data structure no longer suffices for storing activation frames. Instead, the tree of extant activation frames forms a *cactus stack* [46], as shown in Fig. 4. The implementation of cactus stacks is a well-understood problem for which low-overhead implementations exist [40, 43].

Although a work-stealing scheduler can guarantees bounds on both time and stack space [16, 40], implementations that meet these bounds – including Cilk-1, Cilk-5, and Cilk++ – suffer from interoperability problems with legacy (and third-party) serial binary executables that have been compiled to use a linear stack. (Although Fortress, Java Fork/Join Framework, TPL, and X10 employ work stealing, they do not suffer from the same interoperability problems, because they are



Cilk. Fig. 4 A cactus stack. (a) The invocation tree, where function A invokes B and C, and C invokes D and E. (b) The view of the stack by each of the five functions. In a serial execution, only one view is active at any given time, because only one function executes at a time. In a parallel execution, however, if some of the invocations are spawns, then multiple views may be active simultaneously

byte-code interpreted by a virtual-machine environment.) Transitioning from serial code (using a linear stack) to parallel code (using a cactus stack) is problematic, because the type of stack impacts the calling conventions used to allocate activation frames and pass arguments. The property of allowing arbitrary calling between parallel and serial code – including especially legacy (and third-party) serial binaries – is called ***serial-parallel reciprocity***, or ***SP-reciprocity*** for short.

SP-reciprocity is especially important if one wishes to multicore-enable legacy object-oriented environments by parallelizing an object’s member functions. For example, suppose that a function A allocates a new object x whose type has a member function `foo()`, which is parallelized. Now, suppose that A is linked with a legacy binary containing a function B, and A passes x to B, which proceeds to invoke `x.foo()`. Without SP-reciprocity, this simple callback does not work.

Cilk-5 and Cilk++ do not support SP-reciprocity. Cilk Plus supports SP-reciprocity by sacrificing the space bound Inequality (Eq. 4) and using multiple linear stacks to implement the cactus stack. Although no provable bound exists to date on the number of linear stacks needed, empirical studies indicate that $10P$ linear stacks suffice.

The research prototype Cilk-M [60] provides both SP-reciprocity and good theoretical bounds on time and space. Cilk-M uses thread-local memory mapping to implement cactus stacks. Benchmark results indicate that the performance of the prototype Cilk-M runtime system is comparable to the Cilk 5.4.6 system, and the consumption of stack space is modest.

Race Detection

The Cilk++ development environment includes a ***race detector***, called Cilkscreen, a powerful debugging tool that greatly simplifies the task of ensuring that a parallel application is correct. A ***data race*** [73] exists if logically parallel strands access the same shared location, the two strands hold no locks in common, and at least one of the strands writes to the location. A data race is usually a bug, because the program may exhibit unexpected, nondeterministic behavior depending on how the strands are scheduled. Serial code containing nonlocal variables is particularly prone to the introduction of data races when the code is parallelized.

As an example of a race bug, suppose that line 13 in Fig. 1 is replaced with the following line:

```
qsort(max(begin + 1, middle-1), end);
```

The resulting serial code is still correct, but the parallel code now contains a race bug, because the two subproblems overlap, which could cause an error during execution.

Race conditions have been studied extensively [24, 27, 31, 32, 34, 36, 48, 67, 69, 72, 74, 75, 78]. They are pernicious and occur nondeterministically. A program with a race bug may execute successfully millions of times during testing, only to raise its head after the application is shipped. Even after detecting a race bug, writing regression tests to ensure its continued absence is difficult.

The Cilkscreen race detector is based on provably good algorithms [9, 24, 35] developed originally for MIT Cilk. In a single serial execution on a test input for a deterministic program, Cilkscreen guarantees to report a race bug if the race bug is ***exposed***: that is, if two different schedulings of the parallel code would produce different results. Cilkscreen uses efficient data structures to track the series-parallel relationships of the executing application during a serial execution of the parallel code. As the application executes, Cilkscreen uses dynamic instrumentation [20, 65] to intercept every load and store executed at user level. Metadata in the Cilk++ binaries allows Cilkscreen to identify the parallel control constructs in the executing application precisely, track the series-parallel relationships of strands, and report races precisely. Additional metadata allows the race to be localized in the application source code.

Reducer Hyperobjects

Many serial programs use ***nonlocal variables***, which are variables that are bound outside of the scope of the function, method, or class in which they are used. If a variable is bound outside of all local scopes, it is a ***global variable***. Nonlocal variables have long been considered a problematic programming practice [85], but programmers often find them convenient to use, because they can be accessed at the leaves of a computation without the overhead and complexity of passing them as parameters through all the internal nodes. Thus, nonlocal variables have persisted in serial programming. In the world of parallel computing, nonlocal variables may inhibit

```

1  bool has_property(Node *);
2  std::list<Node *> output_list;
3  //...
4  void walk(Node *x)
5  {
6      if (x)
7      {
8          if (has_property(x))
9          {
10             output_list.push_back(x);
11         }
12         walk(x->left);
13         walk(x->right);
14     }
15 }
```

Cilk. Fig. 5 C++ code to create a list of all the nodes in a binary tree that satisfy a given property

```

1  bool has_property(Node *);
2  std::list<Node *> output_list;
3  // ...
4  void walk(Node *x)
5  {
6      if (x)
7      {
8          if (has_property(x))
9          {
10             output_list.push_back(x);
11         }
12         cilk_spawn walk(x->left);
13         walk(x->right);
14         cilk_sync;
15     }
16 }
```

Cilk. Fig. 6 A naive Cilk++ parallelization of the code in Fig. 5. This code has a data race in line 10

otherwise independent parts of a multithreaded program from operating in parallel, because they introduce races. This section describes Cilk++ reducer hyperobjects [39], which can mitigate races on nonlocal variables without creating lock contention or requiring code restructuring.

As an example of how a nonlocal variable can introduce a data race, consider the problem of walking a binary tree to make a list of those nodes that satisfy a given property. A C++ code to solve the problem is abstracted in Fig. 5. If the node x being visited is non-null, the code checks whether x has the desired property in line 8, and if so, it appends x to the list stored in the global variable `output_list` in line 10. Then, it recursively visits the left and right children of x in lines 12 and 13.

Figure 6 illustrates a straightforward parallelization of this code in Cilk++. In line 12 of the figure, the `walk` function is spawned recursively on the left child, while the parent continues on to execute an ordinary recursive call of `walk` in line 13. As the recursion unfolds, the running program generates a tree of parallel execution that follows the structure of the binary tree. Unfortunately, this naive parallelization contains a data race. Specifically, two parallel instantiations of `walk` may attempt to update the shared global variable `output_list` in parallel at line 10.

The traditional solution to fixing this kind of data race is to associate a mutual-exclusion lock (mutex) L with `output_list`, as is shown in Fig. 7. Before updating `output_list`, the mutex L is acquired in

```

1  bool has_property(Node *);
2  std::list<Node *> output_list;
3  mutex L;
4  // ...
5  void walk(Node *x)
6  {
7      if (x)
8      {
9          if (has_property(x))
10         {
11             L.lock();
12             output_list.push_back(x);
13             L.unlock();
14         }
15         cilk_spawnwalk(x->left);
16         walk(x->right);
17         cilk_sync;
18     }
19 }
```

Cilk. Fig. 7 Cilk++ code that solves the race condition using a mutex

line 11, and after the update, it is released in line 13. Although this code is now correct, the mutex may create a bottleneck in the computation. If there are many nodes that have the desired property, the contention on the mutex can destroy all the parallelism. For example, on one set of test inputs for a real-world tree-walking code that performs collision-detection of mechanical assemblies, lock contention actually degraded performance on 4 processors so that it was worse than running on a single processor. In addition, the locking solution has the problem that it jumbles up the order of list elements. That might be okay for some applications, but

other programs may depend on the order produced by the serial execution.

An alternative to locking is to restructure the code to accumulate the output lists in each subcomputation and concatenate them when the computations return. If one is careful, it is also possible to keep the order of elements in the list the same as in the serial execution. For the simple tree-walking code, code restructuring may suffice, but for many larger codes, disrupting the original logic can be time-consuming and tedious undertaking, and it may require expert skill, making it impractical for parallelizing large legacy codes.

Cilk++ provides a novel approach [39] to avoiding data races in code with nonlocal variables. A Cilk++ *reducer hyperobject* is a linguistic construct that allows many strands to coordinate in updating a shared variable or data structure independently by providing them different but coordinated views of the same object. The state of a hyperobject as seen by a strand of an execution is called the strand's "view" of the object at the time the strand is executing. A strand can access and change any of its view's state independently, without synchronizing with other strands. Throughout the execution of a strand, the strand's view of the reducer is private, thereby providing isolation from other strands. When two or more strands join, their different views are combined according to a system- or user-defined `reduce()` method. Thus, reducers preserve the advantages of parallelism without forcing

the programmer to restructure the logic of his or her program.

As an example, [Figure 8](#) shows how the tree-walking code from [Fig. 5](#) can be parallelized using a reducer. Line 3 declares `output_list` to be a reducer hyperobject for list appending. The `reducer_list_append` class implements a `reduce` function that concatenates two lists, but the programmer of the tree-walking code need not be aware of how this class is implemented. All the programmer does is identify the global variables as the appropriate type of reducer when they are declared. No logic needs to be restructured, and if the programmer fails to catch all the use instances, the compiler reports a type error.

This parallelization takes advantage of the fact that list appending is associative. That is, if a list L_1 is appended to a list L_2 and the result appended to L_3 , it is the same as if list L_1 were appended to the result of appending L_2 to L_3 . As the Cilk++ runtime system load-balances this computation over the available processors, it ensures that each branch of the recursive computation has access to a private view of the variable `output_list`, eliminating races on this global variable without requiring locks. When the branches synchronize, the private views are reduced (combined) by concatenating the lists, and Cilk++ carefully maintains the proper ordering so that the resulting list contains the identical elements in the same order as in a serial execution.

```

1 #include <reducer_list.h>
2 bool has_property(Node *);
3 cilk::hyperobject<cilk::reducer_list_append<Node*>>
    output_list;

4 // ...
5 void walk(Node *x)
6 {
7     if (x)
8     {
9         if (has_property(x))
10        {
11            output_list().push_back(x);
12        }
13        cilk_spawnwalk(x->left);
14        walk(x->right);
15        cilk_sync;
16    }
17 }
```

Cilk. Fig. 8 A Cilk++ parallelization of the code in [Fig. 5](#), which uses a reducer hyperobject to avoid data races

Conclusion

Multicore microprocessors are now commonplace, and Moore's Law is steadily increasing the pressure on software developers to multicore-enable their codebases. Cilk++ provides a simple but effective concurrency platform for multicore programming which leverages almost two decades of research on multithreaded programming. The Cilk++ model builds upon the sound theoretical framework of multithreaded dags, allowing parallelism to be quantified in terms of work and span. The Cilkscreen race detector allows race bugs to be detected and localized, and the Cilkview scalability analyzer allows the parallelism of an application to be quantitatively measured. Cilk++'s hyperobject library mitigates races on nonlocal variables. Although parallel programming will surely continue to evolve, Cilk++ today provides a full-featured suite of technology for multicore-enabling any compute-intensive application.

Acknowledgments

Thanks to the many students and postdocs who contributed to Cilk technology over the years. Space does not permit all to be listed. Thanks to the team at Cilk Arts and the new Cilk team at Intel. Special thanks to Matteo Frigo and Bradley C. Kuszmaul, who contributed mightily to developing and maintaining the MIT Cilk codebase. Many thanks to the DARPA and NSF sponsors of this research over the years.

Bibliography

1. Adkins D, Barton R, Dailey D, Frigo M, Joerg C, Leiserson C, Prokop H, Rinard M (1998) Cilk Pousse. Winner of the 1998 ICFP Programming Contest
2. Agrawal K, Fineman JT, Sukha J (2007) Nested parallelism in transactional memory. In: Proceedings of the 13th ACM SIGPLAN symposium on principles and Practice of parallel programming (PPoPP 2008), ACM, New York, pp 163–174
3. Agrawal K, He Y, Hsu WJ, Leiserson CE (2006) Adaptive scheduling with parallelism feedback. TOCS, 16(3):7:1–7:32
4. Agrawal K, Leiserson CE, Sukha J (2010) Helper locks for fork-join parallel programming. In: PPoPP'10, ACM, New York, pp 244–256
5. Agrawal K, Leiserson CE, Sukha J (2010) Executing task graphs using work stealing. In: IPDPS, Atlanta. IEEE, pp 1–12
6. Allen E, Chase D, Hallett J, Luchangco V, Maessen JW, Ryu S, Steele Jr. GL, Hochstadt ST (2008) The Fortress language specification, version 1.0. Sun Microsystems, Burlington
7. Amdahl G (1967) The validity of the single processor approach to achieving large-scale computing capabilities. In: Proceedings of the AFIPS spring joint computer conference, Atlantic City. ACM, New York, pp 483–485
8. Arnold K, Gosling J (1996) The Java programming language. Addison-Wesley, Reading
9. Bender MA, Fineman JT, Gilbert S, Leiserson CE (2004) On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In: Proceedings of the sixteenth annual ACM symposium on parallel algorithms and architectures (SPAA 2004), Barcelona, Spain. ACM, New York, pp 133–144
10. Blelloch GE, Gibbons PB, Matias Y, Narlikar GJ (1997) Space-efficient scheduling of parallelism with synchronization variables. In: Proceedings of the 9th annual ACM symposium on parallel algorithms and architectures (SPAA), Newport. ACM, New York, pp 12–23
11. Blumofe RD (1995) Executing multithreaded programs efficiently. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-677
12. Blumofe RD, Frigo M, Joerg CF, Leiserson CE, Randall KH (1996) An analysis of dag-consistent distributed shared-memory algorithms. In: SPAA'96, Padua, Italy. ACM press, New York, pp 297–308
13. Blumofe RD, Frigo M, Joerg CF, Leiserson CE, Randall KH (1996) Dag-consistent distributed shared memory. In: IPPS'96, Honolulu, Hawaii, pp 132–141
14. Blumofe RD, Joerg CF, Kuszmaul BC, Leiserson CE, Randall KH, Zhou Y (1996) Cilk: an efficient multithreaded runtime system. J Parallel Distribut Comp 37(1):55–69
15. Blumofe RD, Leiserson CE (1998) Space-efficient scheduling of multithreaded computations. SIAM J Comput 27(1):202–229
16. Blumofe RD, Leiserson CE (1999) Scheduling multithreaded computations by work stealing. J ACM 46(5):720–748
17. Blumofe RD, Papadopoulos D (1999) Hood: a user-level threads library for multi-programmed multiprocessors. Technical report, University of Texas, Austin
18. Blumofe RD, Park DS (1994) Scheduling large-scale parallel computations on networks of workstations. In: Proceedings of the third international symposium on high performance distributed computing (HPDC), San Francisco, pp 96–105
19. Brent RP (1974) The parallel evaluation of general arithmetic expressions. J ACM 21(2):201–206
20. Bruening D (2004) Efficient, transparent, and comprehensive runtime code manipulation. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology
21. Buluç A, Fineman JT, Frigo M, Gilbert JR, Leiserson CE (2009) Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In: SPAA. ACM, New York, pp 233–244

22. Charles P, Grothoff C, Saraswat V, Donawa C, Kielstra A, Ebcioğlu K, von Praun C, Sarkar V (2005) X10: an object-oriented approach to non-uniform cluster computing. In: OOPSLA '05. ACM, New York, pp 519–538
23. Cheng GI (1998) Algorithms for data-race detection in multithreaded programs. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge
24. Cheng GI, Feng M, Leiserson CE, Randall KH, Stark AF (1998) Detecting data races in Cilk programs that use locks. In: Proceedings of the ACM Symposium on parallel algorithms and architectures. ACM press, New York
25. Cilk Arts Inc. (2008) Cilk++ programmer's guide, release 1.0 edition, December 2008
26. Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to algorithms, 3rd edn. The MIT Press, Cambridge
27. Crummey JM (1991) On-the-fly detection of data races for programs with nested fork-join parallelism. In: Super-computing'91, Albuquerque. IEEE Computer Society Press, Los Alamitos, pp 24–33
28. Dailey D, Leiserson CE (2002) Using Cilk to write multiprocessor chess programs. *J Int Comput Chess Assoc* 24(4):236–237
29. Danaher JS, Lee ITA, Leiserson CE (2005) Exception handling in JCilk. In: Synchronization and concurrency in object-oriented languages (SCOOL), October 2005. Available at <http://hdl.handle.net/1802/2095>
30. Danaher JS (2005) The JCilk-1 runtime system. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge
31. Dinning A, Schonberg E (1990) An empirical comparison of monitoring algorithms for access anomaly detection. In: Proceedings of the ACM SIGPLAN symposium on principles and practice of parallel programming, San Diego, ACM, New York, pp 1–10
32. Dinning A, Schonberg E (1991) Detecting access anomalies in programs with critical sections. In: Proceedings of the ACM/ONR workshop on parallel and distributed debugging, ACM Press, New York, pp 85–96
33. Eager DL, Zahorjan J, Lazowska ED (1989) Speedup versus efficiency in parallel systems. *IEEE Trans Comput* 38(3): 408–423
34. Emrath PA, Ghosh S, Padua DA (1991) Event synchronization analysis for debugging parallel programs. In: Supercomputing '91, Albuquerque. IEEE Computer Society, Washington DC, pp 580–588
35. Feng M, Leiserson CE (1997) Efficient detection of determinacy races in Cilk programs. In: Proceedings of the ACM symposium on parallel algorithms and architectures, New Port. ACM, New York, pp 1–11
36. Fenster Y (1998) Detecting parallel access anomalies. Master's thesis, Hebrew University, Jerusalem
37. Frigo M (1999) A fast Fourier transform compiler. *ACM SIGPLAN Notices* 34(5):169–180
38. Frigo M (1999) Portable high-performance programs. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge
39. Frigo M, Halpern P, Leiserson CE, Berlin SL (2009) Reducers and other Cilk++ hyperobjects. In: SPAA'09, Calgary. ACM, New York, pp 79–90
40. Frigo M, Leiserson CE, Randall KH (1998) The implementation of the Cilk-5 multithreaded language. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation, Montreal, Canada, pp 212–223
41. Frigo M, Luchangco V (1998) Computation-centric memory models. In: SPAA'98. ACM, New York, pp 240–249
42. Garey MR, Johnson DS (1979) Computers and Intractability. W.H. Freeman and Company, San Francisco
43. Goldstein SC, Schausler KE, Culler D (1995) Enabling primitives for compiling parallel languages. In: LCR '95, Troy, NY, USA, Available from <http://en.scientificcommons.org/43004998>
44. Graham RL (1966) Bounds for certain multiprocessing anomalies. *Bell System Tech J* 45:1563–1581
45. Halbherr M, Zhou Y, Joerg CF (1994) MIMD-style parallel programming with continuation-passing threads. In: Proceedings of the 2nd international workshop on massive parallelism: hardware, software, and applications, Capri, Italy
46. Hauck EA, Dent BA (1968) Burroughs' B6500/B7500 stack mechanism. In: Proceedings of the AFIPS spring joint computer conference. AFIPS Press, Montvale, pp 245–251
47. He Y, Leiserson CE, Leiserson WM (2010) The Cilkview scalability analyzer. In: SPAA'10. ACM, New York, pp 145–156
48. Helmbold DP, McDowell CE, Wang JZ (1990) Analyzing traces with anonymous synchronization. In: Proceedings of the 1990 international conference on parallel processing, University Park, pp II.70–II.77
49. Institute of Electrical and Electronic Engineers (1996) Information technology – portable operating system interface (POSIX) – Part 1: system application program interface (API) [C language], 1996 edn. IEEE Standard 1003.1
50. Intel Corporation (2009) Intel Cilk++ SDK programmer's guide, October 2009. Intel Corporation, Document number: 322581-001US
51. Intel Corporation (2010) Intel R® C++ compiler 12.0 user and reference guides, September 2010. Intel Corporation, Document number: 323271-011US
52. Joerg C, Kuszmaul BC (1994) Massively parallel chess. In: Proceedings of the third DIMACS parallel implementation challenge, Rutgers University, New Jersey, October 17–19
53. Joerg CF (1996) The Cilk system for parallel multithreaded computing. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge. Available as MIT Laboratory for Computer Science technical report MIT/LCS/TR-701
54. Kernighan BW, Ritchie DM (1988) The C programming language, 2nd edn. Prentice Hall, Englewood Cliffs
55. Kuszmaul BC (1994) Synchronized MIMD computing. PhD thesis, MIT Department of EECS, Cambridge
56. Kuszmaul BC (1995) The StarTech massively parallel chess program. *J Int Comput Chess Assoc* 18(1):3–20

57. Kuszmaul BC (2007) Brief announcement: Cilk provides the “best overall productivity” for high performance computing (and won the HPC challenge award to prove it). In: SPAA’07. ACM, New York, pp 299–300
58. Lea D (2000) A Java fork/join framework. In: Java Grande Conference, Stanford University, Palo Alto, pp 36–43
59. Lee ITA (2005) The JCilk multithreaded language. Master’s thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge
60. Lee ITA, Wickizer SB, Huang Z, Leiserson CE (2010) Using memory mapping to support cactus stacks in work-stealing runtime systems. In: PACT’10, Vienna. ACM, New York, pp 411–420
61. Leijen D, Schulte W, Burckhardt S (2009) The design of a task parallel library. In: OOPSLA ’09, Orlando, FL. ACM, New York, pp 227–242
62. Leiserson CE (2010) The Cilk++ concurrency platform. *J Supercomput* 51(3):244–257
63. Leiserson CE, Abuhamdeh ZS, Douglas DC, Feynman CR, Ganmukhi MN, Hill JV, Hillis WD, Kuszmaul BC, St. Pierre MA, Wells DS, Wong MC, Yang SW, Zak R (1996) The network architecture of the Connection Machine CM-5. *J Parallel Distribut Comput* 33(2):145–158
64. Leiserson CE, Schardl TB (2010) A work-efficient parallel breadth-first search algorithm (or how to cope with the non-determinism of reducers). In: SPAA’10. ACM, New York, pp 303–314
65. Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K (2005) Pin: building customized program analysis tools with dynamic instrumentation. In: PLDI ’05: proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation, Chicago, ACM Press, New York, pp 190–200
66. Microsoft Corporation (2010) Parallel patterns library (PPL). Available at <http://msdn.microsoft.com/en-us/library/dd492418.aspx>
67. Miller BP, Choi JD (1988) A mechanism for efficient debugging of parallel programs. In: Proceedings of the 1988 ACM SIGPLAN conference on programming language design and implementation (PLDI), Atlanta. ACM, New York, pp 135–144
68. Miller RC (1995) A type-checking preprocessor for Cilk 2, a multithreaded C language. Master’s thesis, Massachusetts Institute of Technology Electrical Engineering and Computer Science, Cambridge
69. Min SL, Choi JD (1991) An efficient cache-based access anomaly detection scheme. In: Proceedings of the fourth international conference on architectural support for programming languages and operating systems (ASPLOS), Palo Alto., pp 235–244
70. Mohr E, Kranz DA, Halstead RH, Jr. (1991) Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE Trans Parallel Distribut Systems* 2(3):264–280
71. Narlikar G (1999) Space-efficient scheduling for parallel, multi-threaded computations. PhD thesis, Carnegie Mellon University, Pittsburgh
72. Netzer RHB, Ghosh S (1992) Efficient race condition detection for shared-memory programs with post/wait synchronization. In: Proceedings of the 1992 international conference on parallel processing, St. Charles, IL
73. Netzer RHB, Miller BP (1992) What are race conditions? *ACM Lett Program Lang Syst* 1(1):74–88
74. Nudler I, Rudolph L (1986) Tools for the efficient development of efficient parallel programs. In: Proceedings of the first Israeli conference on computer systems engineering, Tel Aviv
75. Perković D, Keleher P (1996) Online data-race detection via coherency guarantees. In: Proceedings of the second USENIX symposium on operating systems design and implementation (OSDI), Seattle, Washington
76. Randall KH (1998) Cilk: efficient multithreaded computing. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge
77. Reinders J (2007) Intel threading building blocks: Outfitting C++ for multi-core processor parallelism. O’Reilly, Sebastopol, CA
78. Savage S, Burrows M, Nelson G, Sobalvarro P, Anderson T (1997) Eraser: a dynamic race detector for multi-threaded programs. In: Proceedings of the sixteenth ACM symposium on operating systems principles (SOSP). ACM Press, New York, pp 27–37
79. Stark AF (1998) Debugging multithreaded programs that incorporate user-level locking. Master’s thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge
80. Stroustrup B (2000) The C++ programming language, 3rd edn. Addison-Wesley, Upper Saddle River
81. Supercomputing technologies group, Massachusetts Institute of Technology Laboratory for Computer Science (2006) Cilk 5.4.2.3 reference manual, April 2006
82. The MPI Forum (1993) MPI: a message passing interface. In: Supercomputing ’93, Portland, OR, pp 878–883
83. The MPI Forum (1996) MPI-2: Extensions to the message-passing interface. Technical Report, University of Tennessee, Knoxville
84. Tom Duff (1983). Duff’s device. Usenet posting. <http://www.lysator.liu.se/c/duffs-device.html>. Accessed 10 Nov 1983
85. Wulf W, Shaw M (1973) Global variable considered harmful. *SIGPLAN Notices* 8(2):28–34

Cilk Plus

► [Cilk](#)

Cilk++

► [Cilk](#)

Cilk-1

► [Cilk](#)

Cilk-5

► [Cilk](#)

Cilkscreen

► [Race Detectors for Cilk and Cilk++ Programs](#)

Cluster File Systems

► [File Systems](#)

Cluster of Workstations

► [Clusters](#)

Clusters

THOMAS L. STERLING
Louisiana State University, Baton Rouge,
LA, USA

Synonyms

[Beowulf clusters](#); [Beowulf-class clusters](#); [Commodity clusters](#); [COW, cluster of workstations](#); [Distributed computer](#); [Distributed memory computers](#); [Linux clusters](#); [Multicomputers](#); [NOW, network of workstations](#); [PC clusters](#); [Server farm](#)

Definition

A commodity cluster is a computing system comprising an integrated set of subsystems, each of which may be acquired commercially and is capable of independent and complete operation. The synthesis of the

commodity subsystems is achieved through the interconnection by one or more private networks that are themselves available from commercial sources. A commodity cluster incorporates a system software stack for resource management and application programming. Usually (but not always) such software is a mix of open source and proprietary components. Typically, a cluster is programmed through a message-passing application programming interface like MPI running on top of the operating system on each subsystem node.

Discussion

Introduction and Overview

A commodity cluster is a distributed computing system consisting of an integrated set of fully and independently operational and marketed computer subsystems (node) used together to perform a single application program or workload. These include compute nodes interconnected by a commercial network and possibly additional nodes configured for specific purposes such as login nodes, administrative nodes, and secondary storage managers. Each component is COTS (commercial off the shelf) and can be acquired separately on the open market. Clusters emerged in the mid-1990s to largely replace most other forms of high performance computing and now serve effectively in virtually every scientific user and commercial market sector. Except at the very top end where MPPs (Massively Parallel Processors) maintain a significant presence, clusters dominate high performance computing. This remarkable evolution is driven by a number of motivating factors, one of which is performance to cost achieved through the exploitation of economy of scale of mass marketed components harnessed en masse to deliver significant improvements in a multiple of operational properties through aggregation by integration. The major components then are the compute nodes, the interconnection network, the node operating system, the system management middleware, and the application programming interface. Clusters ride the continuous wave of technology advances to deliver ever increasing value per unit cost, exploiting progress in every technology class including microprocessors, memory, communication networks and switching, operating system software, and parallel numerical algorithms.

Motivation

The advent of the modern commodity cluster has been motivated by the opportunities it offers. Here are provided some of these that have been responsible for the rapid growth and ultimate domination of clusters in the domain of scalable computing.

Scalable Performance: Performance measured by throughput exhibits positive sensitivity to scale (number of cluster nodes) over broad range for many workloads and applications. Although narrower in range variation in performance measured in terms of response time can also be seen to scale with respect to the number of nodes employed.

Cost: A major driver for the use of clusters is the exploitation of economy of scale derived from mass market of the comprising node subsystems resulting in dramatic advantage in performance to cost with respect to custom parallel systems and low cost for implementation of parallel systems. This factor is driving the dominance of commodity clusters in the mid-range of the processing market and at the high end for major transaction processing and major scientific applications. It should be noted that there are custom cases for specific algorithms in which performance to cost is competitive with clusters.

Problem Size: Large applications require data sets larger than the capacity of single enterprise server mid-level systems. Clusters provide a quick and relatively inexpensive way of aggregating many boxes of memory, cluster nodes, to build systems containing large memory capacity. Where memory is the most expensive component of the system, the use of mass market memories is a cost effective way of achieving the largest possible memory system. These are distributed memory, not shared memory.

Mass Storage: Clusters make an excellent platform for building mass storage systems or applications that are based on secondary storage. The use of industry standards and low cost RAID disk systems attached to nodes of clusters is widely used, for example, by large on-demand search engines making up some of the largest systems in the world.

Early Technology Adoption: Rapid advances in microprocessor technology are most quickly captured by the mass-market products to exploit economy of scale and

achieve early return on investment. Cluster systems integrating such commodity subsystems serving as cluster nodes therefore can incorporate and are able to exploit.

Flexibility of Configuration: Because of the method by which clusters are constructed, using building blocks derived from already commercialized products and COTS (commercial off the shelf) interconnect networks, the end system deployment is highly flexible in its configuration and not limited by a few system offerings of vendors as is the case with custom MPPs. Number of nodes, topology, source of nodes, and many other attributes may be determined by the deploying institution and, indeed, can be modified over time, such as the addition of updated nodes or GPU accelerators. This further extends to the software stack that, if derived from open source components, can largely be defined, refined, and updated by the local site, again not restricted by vendor proprietary constraints.

Sum of the Parts: This notion reflects both the obvious attributes of the cluster itself but also more importantly the benefit of many contributing to a single class of systems, largely through open source framework even though commercial software targeting such framework also is of value.

Programming Model Commonality: A major feature of clusters is that they share the same distributed memory programming models with their custom MPP counterparts. Such application programming interfaces as PVM and MPI derived from the communicating sequential processing model are portable between the two classes of distributed memory system. Due to latency, bandwidth, and overhead considerations a commodity cluster of comparable size (number of nodes and memory capacity) to an MPP may deliver lower sustained performance than the MPP but will produce the same results.

Empowerment: This is the freedom many experience when working with clusters and has provided a strong impetus to acquiring skills and experimenting with such systems. The sense of empowerment is a consequence of the ability to implement and deploy clusters without constraints implicit with working with a particular vendor. Many young practitioners have been thus inspired with high school students finding ways to aggregate

surplus computing elements into locally implemented systems providing an important vehicle for education.

Coolness Factor: There is just something really cool about playing with clusters. Although a purely subjective factor, it nonetheless has driven much activity, especially among younger practitioners who react to many of the empowerment issues above but also to the attraction of open source software, unbounded performance opportunities, the sand box mindset that enables self-motivated experimentation, and a subtle attribute of the pieces of a cluster “talking” to each other on a cooperative basis to make something happen together. There is also a strong sense of community among contributors in the field, a basis for a sense of association that appeals to many people. Working with commodity clusters is fun!

Cluster System Architecture

Hardware Architecture

The hardware architecture of a cluster system is simple in concept, if not in the details of implementation. To first order, it consists of a set of compute nodes and an interconnection network at its core connected to secondary storage, an external local area network, and one or more administration terminals. One or more of the compute nodes are dedicated to serve as the master, host, or login node to handle user sessions.

The heart of the cluster is the COTS (commercial off the shelf) compute node. This subsystem is a fully operational stand-alone computer with the important attribute that it benefits from a market far greater than that of the cluster of which it is a part. The resulting economy of scale yields an exceptional performance to cost greatly in excess of alternative HPC strategies.

Each node is a fully functioning computer system in its own right. It incorporates as its major components:

- One or more multicore processor components that provides the primary computing capability of the node. By far, the most widely used processor family for clusters is the X86 architecture currently including the Intel Xeon (in multiple instantiations) and the AMD Opteron
- DRAM memory modules that provide the main storage of the node combined with a cache hierarchy supporting each of the processor cores

- Motherboard chip-set that provides all of the control features of the node, the intra-node communications, and the BIOS startup mechanisms for booting the system
- Network Interface Controller to connect the node to the cluster system private network
- Local disk storage for the node. Some nodes do not have this and are referred to as “diskless” or “stateless”
- Packaging, power, and cooling

A cluster is the integration of such nodes by means of a private system area network. The network consists of the NICs (described above) installed in the nodes, the switches that route packets, and the channels or cables connecting the switches to the nodes and to each other. The arrangement of the network components is described as the network topology, which for clusters is usually a tree (for smaller clusters) or a Clos network for larger clusters. Important network properties are its bandwidth and latency. Bandwidth establishes the amount of information that can be passed between all pairs of nodes simultaneously. Latency establishes the time it takes to move a packet between two nodes. The primary network types employed today in clusters is Ethernet in its myriad forms (primarily GigE) and the increasingly important Infiniband (IBA) network that provides lower overheads and shorter latencies.

Software Architecture

The cluster software structure matches that of the hardware architecture in integrating the functionality of the separate components in to a single operational system. The roles and responsibilities of cluster software range from the general managing of the individual hardware resources to providing programming environments for users and overall system administration. Typically every node supports its own operating system. Dominant among these is Linux. However, commercial versions of Unix such as Solaris, AIX, and HPUX are employed as well. Microsoft Windows is found on some clusters especially in the business community such as financial markets.

The programming environments at least for the scientific computing arena are based on the MPI

application programming interface, which is ubiquitous, stable, and portable. Application codes written in MPI (employing C, C++, or Fortran as the process language) are easily conveyed across systems of different scale (different number of nodes), different processor architectures, different network types and topologies, and even from clusters to MPPs with no or minimum changes for correctness of operation. Getting optimal performance from the different systems may require some code modifications. MPI describes communication patterns among processes on different nodes, the content of different messages, and their functionality as gather/scatter or collective operations. MPI has provided the first supercomputing common programming framework and is largely responsible for the success achieved by clusters today in industry and government sectors as well as academic research.

Additional software has been developed for system administration, workload scheduling, and initial installation and configuration.

History

The history of cluster computing has been driven by the opportunity to achieve rapid increase in one or more operational properties of a computer system through the aggregation and interoperability of replicated independent systems by means of interconnection networks. Such properties include but are not limited to throughput performance, total storage capacity, performance to cost, reliability, availability, and input/output communications rate. Indeed, the concept of clustering in its broadest sense as an abstraction predates computing altogether and has its antecedents in the genesis of human civilization in the domains of transportation, manufacturing, human enterprise (including military), and living environments (buildings and collections thereof). The motto of the USA found on every dollar bill, "E Pluribus Unum," is an emblem of clustering at least at a high level of abstraction. This general strategy of synthesis of existing systems to comprise much larger ensemble systems has been applied to computing systems since its second generation in the 1950s with SAGE and the IBM 7000 as its earliest exemplars. The modern commodity cluster emerged in the early to mid 1990s with the advent of critical enabling technologies derived from the previous decade with 1997

marking a milestone year of the first commodity cluster on the Top-500 list and the first Gordon Bell Prize awarded to a cluster based computation. Here is presented a brief history of clusters reflecting the diversity of ways they have been employed and the succession of enabling conditions that catalyzed their implementation, deployment, and use. For pedagogical purposes, this history is organized in four distinguishable periods: (1) prior to 1980, (2) 1980–1992, (3) 1993–2004, and (4) 2005 to present and probably beyond. The first was a period of exploration where some rudimentary forms of clustering were attempted and various incipient technologies were derived in their earliest forms that would one day become critical to successful cluster systems. The second period saw the first real clusters and the development of the enabling technologies, both hardware and software, that would ultimately in synthesis launch the cluster revolution as they matured. The third period introduced the classical period of clusters at a time when microprocessors were differentiated between those employed in workstations (NOW) and those used at the lowest cost personal computers or PCs (Beowulf) only to find this distinction to evaporate as microprocessor architectures converged. Equally important were the advances in network technologies, programming models, and middle ware. The final (fourth) ongoing period is witnessing the dominance of commodity clusters for scientific (STEM) and commercial high performance computing with the advent of new multicore processor sockets and GPU accelerators. Except for the highest end computing systems, clusters are likely to serve as the principal family of supercomputers across a broad range of uses and markets for the foreseeable future.

Exploratory Period: Before 1980

The earliest efforts to exploit the concept of clustering predated the true opportunities for improvement in performance-to-cost but addressed certain key niche requirements. It was also a period when some of the technologies that would eventually contribute to the wide spread uses of clusters were initially devised.

Perhaps the first example of a cluster of computers was the IBM SAGE computer system developed for the Department of Defense NORAD air defense system. Sage accepted radar information from a couple of dozen



sites using early wide area network technology and presented the strategic airspace status to operators using one of the first examples of digital video presentation displays. SAGE consisted of two CPUs connected, not to double the performance, but for reliability; if one failed, the other could immediately pick up the workload and continue the time critical processing of defense data. The SAGE CPUs were derived from the original MIT Whirlwind computer architecture but with extended word lengths and other features.

Many of the key technologies that would make up the primary components of later commodity clusters were first derived during this exploratory period. The microprocessor was first developed by Intel introduced in 1971 (4004) with the X86 family that has become the dominant CPU of commodity clusters marketed by 1978. The first local area network technology to be standardized, Ethernet, developed at Xerox PARC. The TCP protocol was also developed at this time. Together these two technologies established the foundations for networking and through their evolutionary descendants created the single most widely used network in clusters both integrating their constituent elements and their external input/output interfaces. Software technology also was developed at this time that would have long-term impact on clusters. Among these was the Unix operating system by Bell Labs that incorporated many of the service infrastructure capabilities that would be necessary for effective clustering. The programming languages Fortran and C, both widely used on clusters today were developed at this time. Finally, the fundamental model of computation that is used almost exclusively on clusters, Communicating Sequential Processes or CSP was formally described at this time upon which a number of future application programming interfaces for distributed memory systems including clusters would be based.

Enabling Period: 1980–1992

Throughout the enabling period, all of the technologies, both hardware and software, were developed that catalyzed the cluster explosion. The microprocessor evolved from the 16 bit to the 32 bit architectures with clock rates reaching 100 MHz. Local area network Ethernet technologies achieved 10 Mbps throughput with continuous improvements in bandwidth per cost as well as in switching and network interface controllers (NIC).

With Moore's Law, DRAM memory density continued its exponential growth with concomitant improvements in storage capacity per cost. Similar improvements were achieved in the arena of SCSI and EIDE disk storage.

Software advances were equally important during this period providing the initial basis for the first clusters. An advanced version of Unix, the DARPA sponsored BSD Unix from UC Berkeley, which included virtual memory, and network communications semantics was a key development that spurred many industry OS packages like Solaris from Sun Microsystems and the early open source Linux operating system that was to become a major software base for future clusters. Early implementations of programming tools based on the CSP model were developed both commercially and in academia including the influential PVM system jointly developed by Emory University and Oak Ridge National Laboratory. Originally targeted to early generation MPPs (Massively Parallel Processors), such application programming interfaces (API) were well suited for future clusters.

The use of local area networks (LAN), principally Ethernet, integrated working environments comprising distributed collections of workstations usually for personal use and shared resources such as printers, file servers, and external internet access. It was recognized by some that dependent on individual usage, many workstations lay idle for considerable periods and were potentially available for other purposes. Efforts were made to employ such aggregations of temporarily unused "workstation farms" for "cycle harvesting" by scheduling applications on unused workstations when available. Among many software packages developed for this purpose was Condor from the University of Wisconsin, still widely used today. Condor matches user job requirements with available system capabilities and schedules these pending jobs as suitable systems are made available.

Perhaps the earliest formal use of the term "cluster" occurred near the end of this period introduced by Digital Equipment Corporation (DEC) to relate to the development of a collection of VAX minicomputers interconnected by means of proprietary network hardware and software. The early Andromeda (M31) project at DEC explored this arena of opportunity followed by the VAXcluster product offerings. While the network

technology was custom, the cluster “nodes” were essentially off-the-shelf VAX computers (e.g., model 11/750) and therefore exhibited a critical property of future generation clusters to emerge a few years later.

By the end of the Enabling Period, essentially all technologies, system architectures, software frameworks, and methodologies had been developed, at least in experimental form or for ancillary purposes, which would lay the foundation for the initial forays into various forms of clusters and the eventual dominance of clusters as a principal means of achieving scalable systems.

Classical Period: 1993–2004

The emergence of the commodity cluster exploited the concepts of clustering, the aggregation of multiple like systems by means of interconnection techniques to form a super-system, and key enabling technologies to deliver a new and at the time unique capability measured in performance-to-cost. While clusters are pervasive today in scientific and commercial computing arenas, at the beginning of the classical period it was far from certain that clusters would be developed, let alone have any significant impact on high performance computing. At that time, “big iron,” the use of custom supercomputers, was ubiquitous with the cold war driving their development and much of their deployment. To many the cluster concept was counter intuitive and even counterproductive to the goals and methodology embodied by such machines as the CRI Cray YMP and the MasPar 2, both custom high-speed computer systems of the day. To some, the cluster paradigm was viewed as a threat to mainstream supercomputing and as it turned out, they were proved correct.

A number of early attempts at cluster implementation were conducted with two projects standing out as the pathfinders for what was to become the dominant form of scalable computing up until the present. These were the NOW (Network of Workstation) project at UC Berkeley and the Beowulf project at NASA (Goddard Space Flight Center and Jet Propulsion Laboratory). From the benefit of hindsight these system concepts were very similar but in the early to mid-1990s they were considered as quite distinct. NOW emphasized high-end workstations, high performance networks, and proprietary software. Beowulf emphasized low cost personal computers, mass market networks

(Ethernet), and the new trend in open source software including the Gnu editors and compilers and the Linux operating system. Both projects began in 1993, with significant systems deployed in 1994, and both had strong impact on the community, essentially defining the range of capabilities and techniques to be incorporated to this day. Among these was the choice between vendor and end user installations. At a time when there were few vendor offerings in the cluster market, most clusters were assembled and installed (both hardware and software) from component systems by the end user facility with early emphasis in the academic and government laboratory environments.

Another important advance was the development of the system area network (SAN) explicitly devised for interconnection networks in clusters. This was represented by Myrinet, a high speed low overhead network marketed in 1995 by Myricom initially for Sun workstations and later for general PCI connected nodes. Myrinet offered a superior performance lower latency and higher bandwidth interconnect to the competing Ethernet but at a higher cost. It was very successful in the emerging commercial marketplace but saw slower adoption by the cost sensitive Beowulf user community. Nevertheless, Myrinet narrowed the gap between commodity clusters and MPPs with their custom interconnects. Other commercial networks included Quadrics and SCI.

The Beowulf Project was initiated in 1993 at the NASA Goddard Space Flight Center in collaboration with the USRA Center for Excellence in Space Data and Information Systems (CESDIS) and the University of Maryland at College Park. The project was undertaken to explore the opportunity of employing mass market personal computer systems for science simulation and data analysis with a critical objective of low cost for dedicated user allocation. An initial set of performance and capacity requirements at a cost within \$50K (1994) hardware deployment. The first Beowulf system, Wiglaf, saw first life in the summer of 1994, and followed by Hrothgar (1995) and Hyglac (1996) all 16 nodes culminating in 1 Gigaflops sustained performance on a real world problem for under \$50K, a performance to cost gain of a factor of 20X–100X that of contemporary MPPs. All Beowulf systems of that time consisted of single microprocessor nodes, Intel X86 family microprocessors, Ethernet interconnect (10 Mbps followed

by 100 Mbps Fast Ethernet) with tree topology. These systems would grow to between 100 and 200 nodes, all still in their original “pizza box” or “tower” cases and usually stacked on shelves or sometimes inserted in racks. By 1997, a combined project of the California Institute of Technology and Los Alamos National Laboratory (LANL) won the first Gordon Bell Prize awarded to a cluster based computation. In the same year, the NOW project had the first cluster entry on the Top-500 List. The commodity cluster had been embraced by the mainstream HPC community. In 1999, *How to Build a Beowulf* by Sterling et al was published by MIT Press and represented the state of the art as it was then known providing a general methodology for realizing commodity clusters.

The most important advance in software for commodity cluster computing was the community-wide development and support for a general and widely accepted application programming interface, MPI (Message Passing Interface) that became the lingua franca for distributed memory system programming. The first MPI standard (MPI-1) was released in 1994 with the first implementations available as early as a year later. A critical factor in the success and early adoption of MPI by users and vendors alike was the MPICH library reference implementations that give early access and credible performance to appropriately crafted applications using the message-passing model. A more advanced but fully backward compatible MPI standard (MPI-2) was released in 1998 with implementations following. Within the scientific computing arena, MPI is ubiquitous on both commodity clusters and MPPs providing portability for users and stable target platforms and market for Independent Software Vendors.

During this period, advanced system software was developed to control cluster management. This middle layer of software such as PBS, the Maui scheduler, and the ROCKS deployment package was created out of a critical necessity for higher productivity of cluster installation, operation, and application. The do-it-yourself mentality while serving as a catalyst for initial exploration and application of clusters nonetheless suffered from inadequate methods and means of stable employment, expected of other commercial computer systems. This middleware went a long way in rectifying this shortcoming while providing both open source

and commercially maintained versions of the software for the widest usage.

As a commercial market for clusters emerged, vendors drove the next evolutionary step by developing and offering products better suited to clusters. Perhaps most importantly was the repackaging of the functional node in to a dense rack mountable unit that provided high density packaging, superior air cooling, superior reliability, improved performance and memory capacity, built in I/O and network ports, and both installed disk and diskless versions. Such nodes borrowed from a class of multiprocessor servers to provide multiple microprocessors in the same node sharing a common memory through cache coherence schemes (e.g., MESI). Offering alternative interconnection networks and different scale systems provided users with a wide space of choice and through mass market and economy of scale of these nodes excellent performance to cost. By 2004, commodity clusters comprised more than 50% of the Top-500 List of the world’s fastest computer systems.

Advanced Period: 2005 to Present (and Beyond)

In recent years, important changes in enabling technology have altered the form and function of commodity clusters. Prior to the advanced period that began in approximately 2005, performance gains were achieved primarily by technology advances driven in part by Moore’s law. Both clock rate and processor core complexity continued to increase at a steady predictable rate. This was augmented with improved high density packaging and corresponding increases in the number of nodes to yield performance gains of greater than 100X per decade. By the beginning of the current period essentially all of the processor architecture tricks had been exploited while limits on power consumption caused clock rates to flat-line around 2 GHz +/- 50%. At the same time, logic density continued to increase leading to multicore components providing multiple cores per socket. Initially MPI was thought to be sufficient to program multicore based commodity clusters but subtle changes in balance of communications resulting from these new system structures is suggesting that slowly alternative programming methods, perhaps augmenting rather than replacing MPI, such as Cilk, TBB, and

Concert, may be required to make best use of multicore based clusters.

A second important change is the recent interest in adapting special purpose graphical processor units (GPU) originally intended for the entertainment industry dependent on high-resolution and high-speed visualization for movie special effects and video games to more general purpose scientific computing application domain. This has resulted in the emergence of heterogeneous cluster architectures with individual nodes incorporating ancillary accelerators for speeding up, sometimes dramatically, aspects of the application workflow. This has challenged programming methodology once again with such supportive APIs as CUDA and the emerging OCL to assist programmers in exploiting these new devices.

A new system area network and community standard, Infiniband (IBA or IB), has emerged as the principal challenger to Ethernet and has largely replaced Myrinet as the high performance low latency and low overhead interconnection fabric. Today, Infiniband represents 36% of the installed base of systems on the Top-500 List surpassed only by Gigabit Ethernet at just over 50%. Early problems with initial IBA offerings slowed market penetration but that has largely been resolved and a mix of these two interconnect technologies may be anticipated with IB increasing its usage in the scientific cluster computing arena.

Commodity clusters now dominate high performance computing with greater than 80% of the HPC market overall. But at the highest end, MPPs dominate. This is due to the advantages of custom interconnect and packaging even though the basic multicore microprocessor and memory technologies employed in both are basically the same. Such MPP machines as the IBM Blue Gene/P and the Cray XT5 are optimized for the requirements of major computer centers where performance, power, size, and reliability are more important than normalized cost. With the reliance on increased number of cores for performance gain, custom MPPs are likely to find increased foothold at the highest realms of HPC replacing in dominance the commodity cluster.

A counter trend, however, is now asserting itself that is likely to increase cluster use even in some aspects of the very highest end of system performance, at least as measured in terms of throughput rather than strong

scaled (reduction in execution time) problems. Web search engines and internet accessed “Cloud Computing” that respond to very large remotely distributed user base calls for unprecedented scale in commodity clusters. This new market will have a dramatic albeit unpredictable effect on the future evolution of the commodity cluster. The only thing that is certain is that clusters will continue to serve the computing community in myriad ways as they evolve to serve diverse markets and user communities.

Related Entries

- ▶ [Distributed-Memory Multiprocessor](#)
- ▶ [MPI \(Message Passing Interface\)](#)
- ▶ [Network of Workstations](#)
- ▶ [Interconnection Networks](#)

Bibliography

1. Pfister GF (1995) In search of clusters: the coming battle in lowly parallel computing. Prentice-Hall, Inc., Upper Saddle River, New Jersey
2. Sterling TL, Salmon J, Becker DJ, Savarese DF (1999) How to build a Beowulf: a guide to the implementation and application of PC clusters. MIT Press, Cambridge, Massachusetts
3. Sterling T, Lusk E, Gropp W (eds) (2003) Beowulf cluster computing with Linux. 2. MIT Press, Cambridge, Massachusetts
4. Kronenberg N, Levey H, Strecker W, Merewood R (1987) The VAXcluster concept; an overview of a distributed system. *Dig Tech J* 1(3):7–21
5. Metcalfe RM, Boggs DR (1976) Ethernet: distributed packet switching for local computer networks. *Commun ACM* 19(7):395–404. DOI= <http://doi.acm.org/10.1145/360248.360253>
6. Gropp W, Lusk E, Skjellum A (1994) Using MPI: Portable parallel programming with the message-passing interface. MIT Press, Cambridge, Massachusetts
7. Torvalds L (1999) The Linux edge. *Commun ACM* 42(4):38–39. DOI= <http://doi.acm.org/10.1145/299157.299165>
8. Pfister G (2001) An introduction to the infiniband architecture. High Performance Mass Storage and Parallel I/O, IEEE Press, Los Alamitos
9. Papadopoulos PM, Katz MJ, Bruno G (2003) NPACI Rocks: tools and techniques for easily deploying manageable Linux clusters. *Concurr Comput Pract Exp* 15(7–8):707–725
10. Henderson RL (1995) Job scheduling under the portable batch system. In: Feitelson DG, Rudolph L (eds) Proceedings of the workshop on job scheduling strategies for parallel processing. Lecture notes in computer science, vol 949. Springer-Verlag, London, pp 279–294

11. Bode B, Halstead DM, Kendall R, Lei Z, Jackson D (2000) The portable batch scheduler and the Maui scheduler on Linux clusters. In Proceedings of the 4th annual Linux showcase & conference. vol 4. Atlanta, Georgia, 10–14 October 2000. Atlanta Linux Showcase. USENIX Association, Berkeley, CA, p 27
12. Everett RR, Zraket CA, Benington HD (1958) SAGE: a data-processing system for air defense. In: Papers and discussions presented at the December 9–13, 1957, eastern joint computer conference: computers with deadlines to meet. Washington, DC, 09–13 Dec 1957. IRE-ACM-AIEE '57 (Eastern). ACM, New York, pp 148–155. DOI= <http://doi.acm.org/10.1145/1457720.1457747>

CM Fortran

► Connection Machine Fortran

Cm* - The First Non-Uniform Memory Access Architecture

DANIEL P. SIEWIOREK¹, ED GEHRINGER²

¹Carnegie Mellon University, Pittsburgh, PA, USA

²North Carolina State University, Raleigh, NC, USA

Definition

The name Cm* derives from an architecture wherein each processor-memory pair was called a “computer module” that could be replicated any number of times, as denoted by the Klein star borrowed from set theory.

Discussion

Chronology

The Cm* architecture was described in the fall of 1973. It was designed to explore how microcomputers could be combined to form large digital systems. As the sixteen-processor multiprocessor C.mmp became operational at Carnegie Mellon University in the early 1970s and Digital Equipment Corporation introduced the LSI-11 (January 1975), the project became focused on building a modularly expandable multiprocessor for parallel processing research. The architecture was specified by March 1975, and the design was completed by the fall of 1975. A single-cluster system was operational by July

1976. A ten-processor, three cluster system and operation system were demonstrated in June 1978. All fifty processors in five clusters were operational by September 1979. The machine was used experimentally for applications until January 1986.

Cm* was one of the first large-scale general-purpose multiple-instruction/multiple-data (MIMD) processors [3]. Two complete operating systems – StarOS and Medusa – were developed along with a host of applications. Over a decade of experience representing more than 100 person-years of effort was accumulated in the emerging area of parallel processing. Cm* researchers explored issues ranging from programming parallel applications, to investigating interactions between the operating system and architecture, to developing an automated experimental environment that facilitated the construction of prototype applications [6].

Project Genesis: Historically, Cm* had its beginning in the register-transfer module (RTM) project [2]. RTMs are a module set for the systematic construction of digital systems at the register-transfer level. The successor to RTM project, and immediate forebear of Cm*, was the computer module (CM) project [2, 4], whose major objective was also the systematic construction of digital systems.

The prime assumption of the CM project was that a simple computer, a processor-memory pair, is an appropriate module for building large digital systems. Modular structures have several advantages, including reduced cost through faster system design, faster production, reduced inventories, and simplified maintenance. A second assumption of the CM project was that communication between modules should be at the level of a single memory reference. This allows fine-grained communication and supports the widest variety of interprocessor communication patterns.

Multiprocessors consist of many autonomous processors that address the same primary memory. In contrast, multicomputers are made up of autonomous computers communicating by means of messages through static or dynamic communication links. In particular, the Cm* architecture provides a continuum of memory sharing between processors ranging from no shared memory (multicomputer) to fully shared memory (multiprocessor).

Cm* - The First Non-Uniform Memory Access Architecture. **Table 1** Synchronization granularity and distributed computer structures

Grain size	Synchronization interval (instructions)	Distributed computer structure	Communication overhead (instructions)
Fine	1	Vector/array processor	1
Medium	10–100	Multiprocessor	1–10
Coarse	100–10,000	Multicomputer	100–10,000
Large	10,000–10 million	Network	10,000–10 million

One way of positioning a multiprocessor is to consider the synchronization granularity [7], or frequency of synchronization between tasks. Table 1 shows, for a variety of synchronization granularities, the best-suited distributed computer organization. Multiprocessor systems such as Cm* are more suitable for a medium grain of synchronization.

Multiprocessors and networks employ a variety of interconnection structures to couple processors with memory units. Among these are full interconnections, shared buses, multiple buses, crossbar switches, and multistage networks. See Siegel [8] for a complete discussion of this topic. The fully connected network couples each processor with each memory through a dedicated link. A shared bus is a single communication path to which both processors and memory are connected. Bus arbitration may either be done in a central arbiter or be distributed among the units. The bus bandwidth is divided between the processors and does not scale for large N .

A multiple shared bus is a set of shared buses connected by gateways or switches. The switches may be organized in many ways. The best-known topologies are crossbar switches and hierarchical switches. In crossbar switches, each processor and each memory unit is connected to a dedicated shared bus. Switches are used to connect the processor buses and memory buses into the form of a crossbar. In the 16-by-16 crosspoint switch of C.mmp, each switch connected a full bus of over 70 wires. Crosspoint switch complexity grows as N^2 where N is the number of processor/memories, and it does not scale for large N . With a hierarchical switch, a group of functional units is clustered around a single shared bus, and the clusters are interconnected with another shared bus. By repeating the process, a hierarchically organized multi-shared-bus is obtained. Bandwidth can be increased by adding another bus.

The number of bus connections grows approximately linearly with N .

Rather than have separate processors and memories, Cm* coupled processors and memories so that not all memory accesses encountered switch delays. Borrowing concepts from memory-mapped input/output, portions of the memory address space were assigned to local memory while the remainder of the address space could be mapped to any memory location in any other module – effectively turning Cm* into a software-managed hierarchical cache.

Two conceptually different operating systems were implemented for Cm*. Large portions of the operating-system kernels resided in firmware with operating-system calls triggered by accessing special memory addresses.

Cm* Architecture Evolution

Cm* is structured using processor-memory pairs called *computer modules* or Cm's (after the PMS notation of [1]). The memory local to a processor also serves as the shared memory in the system. Inherent in this structure is the assumption of program locality. The efficient use of the system depends on ensuring that most of the code and data referenced by a processor will be held local to that processor. Early Cm* measurements with various benchmarks applications indicate that local-memory hit ratios of 0.8–0.9 (the fraction of total memory references directed to local rather than shared memory) were readily achieved.

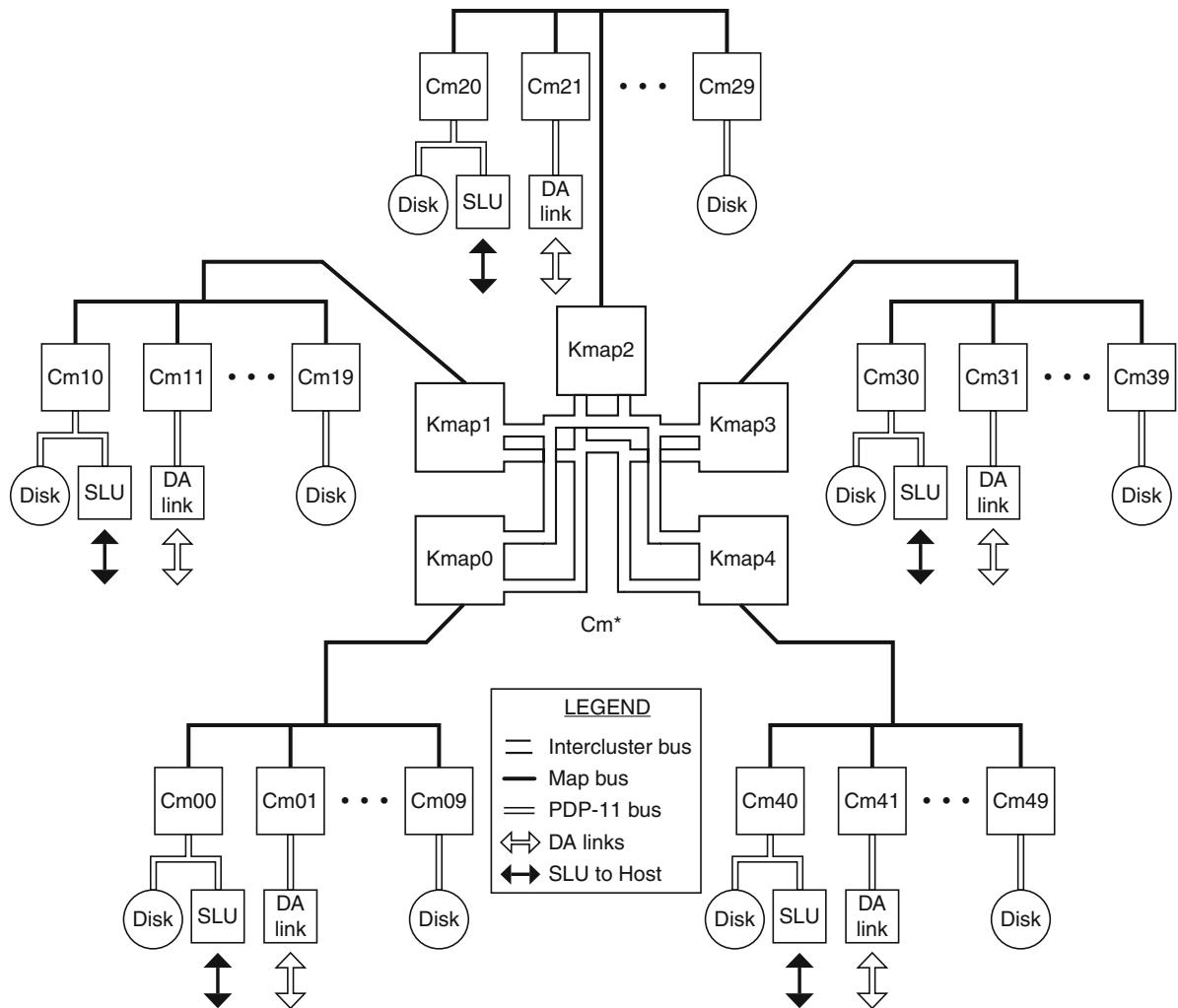
Initially, one self-contained module was envisioned that consisted of a processor, memory, and an intelligent interface. The result was termed a computer module (Cm). The address-mapping controller (Kmap) performed all the functions necessary for generating external memory requests and responding to external requests for its local memory. So that the capacity

for interprocessor communication would not be limited by any single communication path, each Kmap connected to two inter-Cm buses. Further investigation led to the conclusion that very little performance was lost by centralizing the modules' address-mapping and multiple-bus connection functions in a Kmap. The Kmap was shared by a number of computer modules, which saved at least half the cost of a one-Kmap-per-module design. The programmable high-performance Kmap was shared by several Cm's connected to an inter-Cm bus via simple interfaces (Slocals). The basic function of the Slocal was to act as a buffer between the processor and the inter-Cm bus and provide sufficient control functions to generate or respond to external memory requests.

Cm* Architecture

The original description of the design and implementation of Cm* appears in two papers [4, 10]; the design and switching of Cm* and a detailed account of one particular addressing structure are described by Swan [9].

Cm* consisted of 50 Cm's, connected together by a hierarchical, distributed switching structure depicted in Fig. 1. The lowest level of the switching hierarchy comprised Slocals, local switches that connect individual Cm's to the rest of the structure. Cm's were grouped together into clusters that were presided over by high-speed microprogrammable communication controllers called Kmaps. A Kmap provided the mechanism for Cm's in its cluster to communicate with each other, and cooperated with other Kmaps to service requests



Cm* - The First Non-Uniform Memory Access Architecture. Fig. 1 Cm* architecture

from its Cm's to access Cm's in nonlocal clusters. Besides this address-mapping function, since Kmaps were microprogrammable, it was usual to implement key operating-system functions in Kmap microcode.

All communication that involves Kmaps was performed via packet switching rather than circuit switching to avoid deadlock over dedicated switching paths. Packet-switched communication also allowed the processing of requests by the Kmaps to be overlapped, since switching paths did not need to be allocated for the duration of a request; this resulted in improved utilization of the switching structure. The interconnection structure between Cm* clusters was essentially arbitrary. The Kmap in each cluster had two bidirectional ports, each of which could be connected to a separate intercluster bus to achieve a variety of interconnection schemes. In the implemented configuration, all five Kmaps were connected to two intercluster buses, as shown in Fig. 1. The physical implementation of Cm* is shown in Fig. 2.

Cm's and Slocals

Each Cm was a processor-memory-switch combination, consisting of a standard off-the-shelf Digital Equipment Corporation (DEC) LSI-11 (the first single-board implementation of the PDP-11 instruction set), 64 K or 128 K bytes of memory, one or more I/O devices, and a custom-designed Slocal that connected the processor-memory combination to the rest of the system (Fig. 1). When the processor of a Cm initiated a memory reference, its Slocal was responsible for determining whether the reference was to be directed to local memory or out to the Kmap for further mapping. The Slocal used the four high-order bits of the processor's address, along with the current address space, to access a mapping table that determined whether the reference was to proceed locally or not. References that mapped to local memory proceeded with no loss of performance; references that mapped to another Cm were slower by a factor of three; and references that mapped to a Cm in another cluster were slowed again



Cm* - The First Non-Uniform Memory Access Architecture. Fig. 2 Physical structure of Cm* with four Cm's shown in the upper drawer and the K.map in the lower drawer



by a factor of three. These figures were the best possible ratios that could be achieved on Cm*, and therefore amount to constraints imposed by the hardware itself rather than to any microcode implementation features. All I/O devices in Cm* were connected to the various LSI-11 buses. Since there was no interprocessor communication mechanism other than the standard one for memory references, interrupts generated by an I/O device needed to be fielded by the processor to which the device is directly attached.

Kmaps: Transaction Controllers

Acting both as the switching center within a cluster, and as a node in a network of clusters, the Kmap served to coordinate synchronization and communication in Cm*. A fast (157-ns. cycle), horizontally microprogrammed (80-bit wide) microprocessor, the Kmap itself, consisted of three tightly coupled processors. The bus controller, or Kbus, acted as the arbitrator for the bus that connected Cm's in the local cluster to their Kmap; the Linc managed communication to and from the Kamp to other Kmaps; and the mapping processor, or Pmap, responded to requests from the Kbus and Linc, and performed most of the actual computation for a service request. The Pmap also directed the Kbus and Linc to perform any needed operations on behalf of the request being processed.

The Kmap was a transaction controller, sending and receiving message packets that contained requests and replies, following a protocol designed by the microprogrammer. The Kmap routinely mapped memory accesses by one processor into the local memory of another computer module. It needs to do so very rapidly and with little delay to achieve reasonable system performance. Thus, the Kmap contained some special hardware features designed to assist it in controlling many transactions at a high rate of speed.

The Kmap hardware supported eight separate Pmap processes, known as contexts, each with its own set of general-purpose registers and its own microsubroutine stack. Typically, each context was in charge of one transaction. When a context needed to wait for a message packet to return with the reply to some lower-level request, it had the Pmap switch to another context so that work on another transaction can proceed concurrently.

The Kmap also contained 5,120 words of random-access memory, called the data RAM, which the Pmap

can read and write with a delay of a few microinstructions. Because the data RAM was shared by all contexts, it typically held information that was of interest to more than one transaction, such as cached pieces of address-translation tables and mechanisms for synchronizing the use of other resources among different contexts.

Taken individually, each Kmap appeared to the Pmap microprogrammer as a nonpreemptive, hardware-scheduled multiprogramming system (since contexts were never interrupted, but suspended only when the microprogrammer directed). Taken collectively, the network of all Kmaps presented the microprogrammer with a distributed system based on message-packet intercommunication.

The Interface Between Kmap and Computer Module

The Pmap communicated with the computer modules in its cluster via the map bus, a packet-switched bus controlled by the Kbus. The Kbus fielded requests and replies from the computer modules, coordinated the transfer of data across the map bus between computer modules or between a computer module and a Pmap context, and kept track of which Pmap contexts are free to service new requests. Two queues, the Kbus out queue and the Pmap run queue, provided the interface between the Kbus and Pmap.

A request by a computer module to the Kmap is said to invoke some Kmap operation. The most frequent Kmap operation was the mapping of nonlocal access to a location in the physical memory of some computer module in the cluster. This was called a mapped reference. Each memory access issued by the processor of a computer module passed through its Slocal, which either routed the access directly to local memory or sent it out to the Kmap. A memory access handled by the Kmap also could be mapped back to the local memory of the issuing processor, but a direct local access was about three times faster.

For more complicated Kmap operations, the Pmap microprogrammer generally appropriated some subset of the computer module processor's virtual address space, designating specific addresses the processor could use to invoke other Kmap operations. These special Kmap operations were invoked in exactly the same manner as I/O operations are invoked on a computer that has memory-mapped device control registers.



The Interface Between Kmap and Kmap

Kmaps communicated with each other via an intercluster bus, a packet-switched bus that was jointly controlled by the Linc processors in each of the directly connected Kmaps. The Linc maintained queues of incoming and outgoing messages, interacted with the Kbus to activate and reactivate Pmap contexts, and provided the local storage for Pmap contexts to construct and inspect intercluster messages. Each Linc was interfaced to two independent intercluster buses.

An intercluster message contained up to eight 16-bit words of data, of which all words except the first were totally uninterpreted by the Linc. Each intercluster message was sent from an immediate source Kmap to an immediate destination Kmap. The number of the destination Kmap appeared in a fixed place in the message so that the Linc could determine which messages are sent to its cluster. Intercluster messages were of two types: forward messages, which invoked a new context at the destination Kmap, and return messages, which returned to a waiting context at the originating Kmap. A return message contained the context number of the to-be-reactivated Pmap context in a fixed place where the Linc could find it in order to inform the Kbus. These intercluster messages were designed to be used as a mechanism for implementing remote procedure calls between Kmaps.

In a configuration of the Cm* system, it was quite possible that two particular Kmaps had no intercluster bus in common and thus could not send messages directly to one another. Each Kmap connected directly to two intercluster buses, however, and as long as some path through a series of intermediate Kmaps could be found, the two Kmaps in question could still communicate, provided each of the intermediate Kmaps cooperated by forwarding the message closer to its ultimate destination.

Communication with Cm*

When software was first being developed on Cm*, there arose a need to let the software developer allocate certain resources (such as clusters and debugging tools) themselves and to protect these resources from being disturbed by other users. The Cm* Host system was a serial-line-oriented resource management facility. It

was initially implemented on a PDP-11/10 with 28 K words of memory.

Several communication lines connected the Host to components of Cm*, to terminals, and to other Carnegie Mellon University Computer Science Department computers. There were serial-line connections from the Cm* Host to ten individual Cm's, two of which were in each of the five clusters. Because the other 40 Cm's lacked serial-line connections, they needed to be loaded from other Cm's via a Kmap or from peripheral devices. The communication lines served two purposes. First, they allowed the user to communicate with other computers from Cm*, as some programs on Cm* interacted with programs on other machines. They also permitted the user to monitor debugging information on both machines simultaneously from one terminal.

In addition to its 50 Cm's, Cm* had three LSI-11's, known as hooks processors, which were used for debugging the Kmaps. These processors controlled the "hooks," which was a collective term for a set of hardware that was designed into each Kmap to permit complete external control and diagnosis. The hooks consisted of several control registers and other hardware within the Kmap, an LSI-11 interface to make the hardware accessible from an LSI-11, and a bidirectional hooks bus used to transmit information between the control hardware and the LSI-11 bus interface. The hooks appeared to an LSI-11 as a group of eight words in its physical address space. By reading and writing these words, the hooks processor had almost total control over the internals of the Kmap. It could load microcode; start, stop, and single-cycle the Pmap/Linc and Kbus clocks, read out most, and write some of the internal registers of the Pmap; disable certain error checks within the Pmap; and initialize the Kmap.

The diagnostic processor (DP) was added to the Cm* system to collect hardware reliability and availability information. It hosted a program called the Auto-Diagnostic Master, which ran diagnostics on Cm's that were otherwise idle. The DP was an LSI-11 with 28 K words of memory. It had two serial-line connections to the Cm* host. One connection provided a user interface through which one can request status reports about particular Cm's, particular clusters, or the entire Cm*. The second serial-line interface was used by the Auto-Diagnostic program as a command interface

to the Host. The program logged in over the second line, directs the Host to run diagnostics for it, and on operator request, transferred the statistics file to a PDP-10.

Because the system software developed for Cm* was written, compiled, and stored on a remote machine, object code frequently had to be transferred to Cm*. To facilitate high-speed transfers, a DA Link was developed to provide a 10-megabaud parallel DMA link between Cm* and a DEC-10. Between an unloaded LSI-11 and an unloaded DEC KL-10, these links could transfer more than 600,000 words per second. A file-transfer system residing on the DEC-10 provided reliable file transfers.

Cm* Experiments

An extensive set of experiments were conducted with Cm* involving two different operating systems. Many of the experiments focused on speedup, the ratio between the elapsed time required by the one-processor version of a parallel algorithm to the elapsed time for its N -processor counterpart. Speedup is usually between 1 and N . Three factors were responsible for the shape of the speedup curve (e.g., speedup as a function of the number of processors): algorithm penalty, implementation penalty, and their interaction [11].

The algorithm penalty is composed of the separation overhead (cost of process decomposition and data partitioning) and the reconstitution overhead (cost of the interchange and reporting of intermediate and final results). The implementation penalty is composed of access overhead (cost of accessing shared resources) and the contention for these shared resources. The interaction between the algorithm and implementation leads to two other types of overhead: the overhead of synchronization and the cost of adapting a parallel algorithm to a specific architectural implementation.

In addition, at the macro level, parallel algorithms on Cm* were divided into six classes: asynchronous, synchronous, multiphase, partitioning, pipeline, and transaction processing. These classes of parallel algorithm structures represented distinct cross-sections of the algorithm-overhead and implementation-overhead profiles. Once a parallel algorithm was classified into one of the six categories, one could predict how it would perform. Further, the characteristics of parallel

architectures were parameterized to predict the performance of a parallel implementation.

Summary

The Cm* architecture consisted of 50 computer modules, connected together by a distributed switching structure. The lowest level of the switching hierarchy consisted of Slocals, local switches that connected individual Cm's to the rest of the structure. Cm's were grouped together into clusters that were presided over by high-speed microprogrammable communication controllers called Kmaps. A Kmap provided the means for Cm's to communicate with other Kmaps to service requests for memory references to other clusters. In addition to address mapping, key operating-system functions generally were implemented in microcode.

Bibliography

1. Bell CG, Newell A (1971) Computer structures: readings and examples. McGraw-Hill, New York
2. Bell CG, Eggert JL, Grason J, Williams P (1972) The description and the use of register transfer modules (RTM's). IEEE Trans Comput C-21(5):495–500
3. Flynn MJ (1966) Very high-speed computing systems. Proc IEEE 54:1901–1909
4. Fuller SH, Siewiorek DP, Swan RJ (1973) Computer modules: an architecture for large digital modules. In: Proceedings of the first annual symposium on computer architecture, University of Florida, Gainesville. ACM/SIGARCH, ACM, New York, pp 231–236 (reprinted as Computer Architecture News 2 (4))
5. Fuller SH, Ousterhout JK, Raskin L, Rubinfeld P, Sindhu PS, Swan RJ (1978) Multi-microprocessors: an overview and working example. Proc IEEE 66(2):216–218
6. Gehringer EF, Siewiorek DP, Segall Z (1987) Parallel processing: the Cm* experience. Digital Press, Bedford
7. Mohan J, Jones AK, Gehringer EF, Segall ZZ (1985) Granularity of parallel computation. In: Gallizzi EL et al. (eds) Proceedings of the eighteenth Hawaii international conference on system sciences, Hawaii, vol 1. IEEE, Washington, DC, pp 249–256
8. Siegel HJ (1985) Interconnection networks for large-scale parallel processing. Lexington Books, Lexington
9. Swan RJ (1978) The switching structure and addressing architecture of an extensible multiprocessor, Cm*. PhD dissertation, Carnegie-Mellon University, Pittsburgh
10. Swan RJ, Fuller SH, Siewiorek DP (1977) Pittsburgh Cm*: a modular, multi-processor. In: Proceedings of the national computer conference, AFIPS, Texas, USA, pp 637–644
11. Vrsalovic D, Gehringer EF, Segall ZZ, Siewiorek DP (1985) The influence of parallel decomposition on the performance of multiprocessor systems. In: Proceedings of the 12th annual symposium on computer architecture, Boston. IEEE Computer Society Press, Los Alamitos, CA, pp 396–405

CML

► [Concurrent ML](#)

CM-Lisp

► [Connection Machine Lisp](#)

CnC

► [Concurrent Collections Programming Model](#)

Coarray Fortran

ROBERT W. NUMRICH

City University of New York, New York, NY, USA

Synonyms

[Co-array Fortran, CAF](#)

Definition

The coarray programming model is an extension to the Fortran language that provides an explicit syntax and an explicit execution model for the development of parallel application codes.

Introduction

Fortran 2008 contains the coarray parallel programming model as a standard feature of the language [11]. It is the first time that a parallel programming model has been added to the language as a supported feature, portable across all platforms. Compilers supporting the model are available or under development from all the major compiler vendors.

The coarray programming model consists of two new features added to the language, an extension of the normal array syntax to represent data decomposition plus an extension to the execution model to control parallel work distribution.

Execution model

The coarray execution model is based on the Single-Program-Multiple-Data (SPMD) model where replications of a single program execute independently within their separate memory spaces. Each replication is called an image, and an intrinsic function,

```
p = num_images()
```

returns the number of images at run-time. The run-time system assigns local memory to each image, and each image executes asynchronously along independent paths through the program based on its own data and its own unique image index. Another intrinsic function,

```
me = this_image()
```

returns the image index at run-time. Image indices start at image number one following Fortran's default rule for lower bounds.

At times the independent images need to interact with each other to coordinate their activities. The programmer inserts image control statements in the program at appropriate points to synchronize images and to maintain memory consistency across images. The statement,

```
sync all
```

for example, imposes a full barrier across all images. Each image, when it encounters this statement, completes all memory activity, both local and remote, and then registers its presence at the barrier. No image executes any statement following the barrier until all images have registered at the barrier. It is the easiest way for the programmer to maintain memory consistency across images because memory in each image has reached a well-defined state. The programmer is responsible for knowing what state is the correct state across a barrier.

The programmer may also want to synchronize among subsets of images. Pairwise interaction, for example, may be accomplished by one image synchronizing with its neighbor to the right,

```
sync images(me+1)
```

and its partner synchronizing

```
sync images(me-1)
```

with its neighbor to the left.

The programmer may also control interaction among processors by inserting critical segments into the program,

```
critical
  :
end critical
```

where only one image at a time may enter the segment. The programmer may also use intrinsic locks to introduce more sophisticated locked segments similar to critical segments.

Coarrays and Codimensions

A coarray is a variable,

```
real :: x[*]
```

declared with a codimension. The codimension in square brackets [*] means that there is a real variable with the same name x assigned to the local memory of each image. By default the images are numbered starting with image number one, and the asterisk means that the upper bound on the codimension equals the number of images at run-time. The programmer writes code that is independent of the number of images and never writes code that assumes a particular number of images. If the programmer wants to think of image indices starting with number zero, or any other number p, the alternative declaration,

```
real :: x[p:*
```

allows for complete flexibility. The upper bound for the codimension is always an asterisk, and image indices lie between p and num_images() -p+1.

A normal array in Fortran is a set of scalar variables,

```
real :: y(n)
```

each member of the set labeled by an integer dimension index. In the same way, a coarray is a set of general objects, scalars, for example, in declaration (7) or arrays,

```
real :: z(n) [p:*
```

in declaration (10). Each image has an object of the same name, same type, and same size assigned to its local memory, and each object in the set is labeled by an integer codimension index. Coarray variables may be declared for any type including derived types,

```
type(someType) :: Q[p:*
```

providing the programmer a powerful tool for defining distributed data structures. The only exception is that a variable with the pointer attribute may not be declared as a coarray.

Codimensions may also be multi-dimensional,

```
type(someType) :: R[0:q,0:*
```

allowing the programmer to think of images as logically decomposed into, for example, a two-dimensional grid [9]. The final codimension is always an asterisk. An alternative form of the intrinsic function,

```
[myP,myQ] = this_image(R)
```

returns image indices [myP, myQ] relative to the codimensions defined by the declaration statement. The programmer may want, for example, to identify the east-west neighbors [4, 10] with co-indices [myP+1, myQ] and [myP-1, myQ] and north-south neighbors with co-indices [myP, myQ+1] and [myP, myQ-1].

Writing Code for the Coarray Model

Coarray variables are visible across all images. The programmer may, therefore, write code that moves data from one image to another by pointing to it with a co-index. For example, the code

```
real :: x[*],y
me = this_image()
y = x[me+1]
```

moves the value of the variable x [me+1], located in the local memory of the image to the right, into the variable y, located in the local memory of the image executing the statement. The variable y exists in the local memory of each image but, since it is not a coarray variable, it is visible only locally to each image. An image may also move values in the other direction,

```
x[me-1] = y
```

defining the value of x [me-1] in the local memory of the image to the left.

An image executes each statement it encounters according to the normal execution rules of Fortran. The image index in square brackets has nothing to do with execution. The images to the right and to the left know nothing about what other images may be doing with values of variables in their local memories. It is the programmer's responsibility to insert image control

statements into the program to control memory activity so that the state of memory across images is the state the program expects for correct behavior. For example, suppose image number one reads the value of an input variable that every image needs,

```
real :: n[*]
me = this_image()
if(me == 1) then
  read(*,*), n
  sync images(*)
else
  sync images(1)
  n = n[1]
end if
```

After reading the coarray variable `n` from standard input, image number one executes a `sync images(*)` statement notifying all the other images that it has the value in its local copy of the variable. The other images, meanwhile, execute the `sync images(1)` statement waiting for a signal from image number one. When they receive it, they each individually read the value `n[1]` from image number one without regard to what other images may be doing. They each are guaranteed to receive the correct value of the variable.

Multiple codimensions express the relationship between north-south and east-west neighbors in a natural way. A coarray variable with declaration

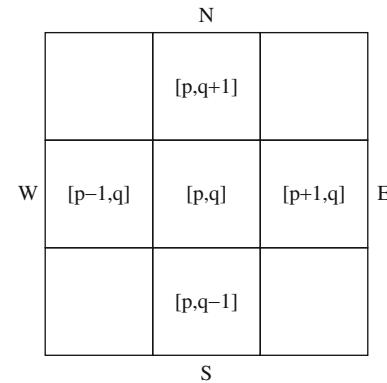
```
real :: v(0:m+1,0:n+1)[proc,*]
```

might represent a velocity field decomposed into a two-dimensional grid with overlapping halo cells around the edges to support, for example, a finite difference operator [4, 10]. At certain points in the calculation, these halo cells must be updated to maintain data consistency across images. With image indices as shown in Fig. 1, each image with image index `[p, q]` executes the halo exchange in the east-west direction,

```
v(:,n+1) = v(:,1)[p+1,q]
v(:,0)   = v(:,n)[p-1,q]
```

using a natural representation in coarray syntax. Notice that each statement moves an entire column of data from one image to another. The north-south exchange is similar.

Multiple codimensions also represent linear algebra operations naturally using blocked matrix decompositions [9]. The following code, for example,



Coarray Fortran. Fig. 1 Representation of a two-dimensional domain decomposition using codimensions for north-south-east-west directions

```
real,dimension(n,n),codimension[p,*]::a,b,c
do R=1,p
  c(:, :) = c(:, :) +
    matmul(a(:, :)[myP,R], b(:, :)[R,myQ])
end do
```

computes the matrix multiplication in parallel for a $[p, p]$ image decomposition and a global problem size that is a multiple of p . Each image accumulates the block of the result matrix that it owns by summing together the products of the blocks of the first matrix in the same row of images with the blocks of the second matrix in the same column of images [5]. When image indices are omitted from a coarray in a statement, such as `c(:, :)`, the indices default to the local values, `c(:, :) [myP, myQ]`, as in this example. This convention saves the programmer from specifying redundant information and helps the compiler to generate optimized code.

Object-Oriented Design with Coarrays

Starting with the Fortran 2003 standard [7], Fortran is now an object-oriented language. Objects declared as coarrays provide a powerful combination for defining distributed data structures with associated methods for communication between objects owned by different images.

Abstract Maps

The coarray model contains no distributed data structures. All objects are local objects assigned to the local memory of each image. Coarray objects are visible to

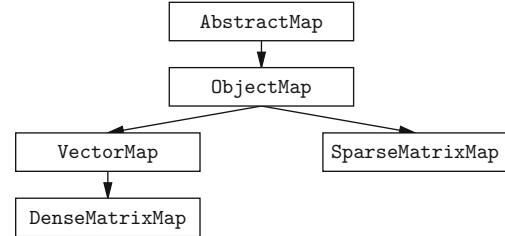
other images but other objects are not. To create distributed objects, the programmer defines derived types that are analogous with classes. Objects of this type may have a complicated structure that describes how the programmer wants data distributed across images. Multi-dimensional coarrays may not be enough to describe these structures.

One way to describe complicated distributions is through extensions of an abstract map,

```
type,abstract,public :: AbstractMap
contains
procedure(getNumberOfObjects),    &
  public,pass(this),deferred :: &
  getNumberOfObjects
proceduregetImageIndex),          &
  public,pass(this),deferred :: &
  getImageIndex
procedure(getLocalIndex),         &
  public,pass(this),deferred :: &
  getLocalIndex
  :
end type AbstractMap
abstract interface
  integer function getNumberOfObjects(this)
    import :: AbstractMap
    class(AbstractMap),intent(in) :: this
  end function GetNumberOfObjects
  :
end interface
```

This abstract object contains nothing more than a set of interfaces for deferred procedures that every map must contain. For example, the function `getNumberOfObjects()` returns the number of objects represented by the map. Another deferred function, `getImageIndex(k)`, returns the image index that owns global object `k` in its own local memory. Another function, `getLocalIndex(k)`, returns the object index used locally by the image that owns the global object `k`. Other functions, not listed, return the global object index corresponding to a local object, the number of objects owned by a specific image, and so forth. Any concrete map that extends this abstract map must implement each of these functions plus other procedures specific to a particular distribution.

Specific concrete maps are extensions of the abstract map as shown in Fig. 2. One extension, called an object map, might represent the decomposition of a set of unspecified objects. A specific kind of object might be a section of a vector with a corresponding map representing a particular decomposition such as a block-cyclic



Coarray Fortran. Fig. 2 Problem decomposition represented as extensions of an abstract map

distribution. The vector map might be extended to represent a block matrix structure with one vector map for the columns and another vector map for the rows. Another extension might represent a sparse vector or a sparse matrix structure [8, 9].

An object map is a composite function,

$$L^p = \Lambda^p(\Pi(G)),$$

loosely based on the Composite Design Pattern [4, 6] as shown in Fig. 3. A global set of n objects $G = \{G_1, \dots, G_n\}$ is first permuted,

$$\pi = \Pi(G),$$

such that

$$\pi_j = \Pi_j^i(G_i) \quad i, j = 1, \dots, n.$$

Subsets of these permuted objects are then projected to n_p local objects $L^p = \{L_1^p, \dots, L_{n_p}^p\}$ on specific images,

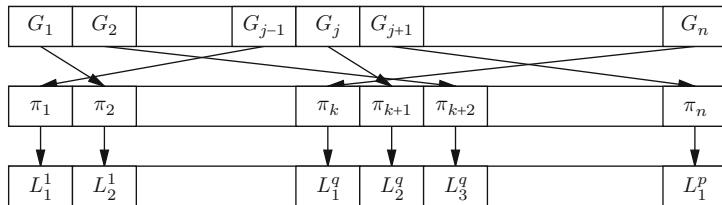
$$L^p = \Lambda^p(\pi),$$

such that

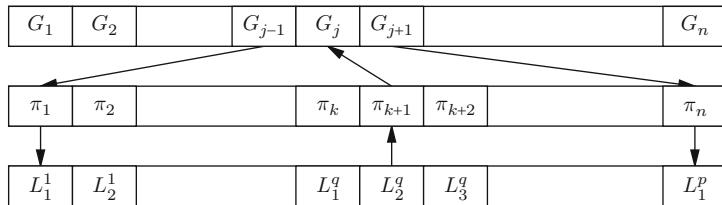
$$L_k^p = (\Lambda^p)_k^j(\pi_j), \\ k = 1, \dots, n_p, \\ p = 1, \dots, \text{num_images}()$$

represents local block k on image p .

The composite function has an inverse so that each image knows how its set of local objects is related to the set of global objects. Figure 4, for example, shows how image q locates its neighbor to the left for its second local object L_2^q as the first local object L_1^1 on image number 1 and its neighbor to the right as local object L_1^p on image number p . These relationships are encapsulated in procedures associated with each object that performs, for example, halo exchanges between distributed matrix objects.



Coarray Fortran. Fig. 3 A composite object map



Coarray Fortran. Fig. 4 Inverse map to nearest neighbors

Blocked Vector Maps

An important distributed object is a blocked vector. In this case the objects to be mapped are blocks of a global vector of size n where each block is a subsection of the global vector `block(n1 : n2)` with some rule for picking the lower bound $n1$ and the upper bound $n2$ for each block. These blocks are assigned to images, and the map keeps track of how many blocks go to each image, how the blocks are numbered locally on each image, and the relationship of the local blocks to the original global vector.

The programmer defines a Type `VectorMap` as an extension to the Type `AbstractMap` with whatever components it needs to define a map for a blocked vector and to implement the deferred functions, adding new functions as needed. For example, there may be a function, `getBlockSize(k)`, that returns the size of block k . The programmer also writes a constructor, a function with the same name as the type,

```
Type (VectorMap) :: map
map = VectorMap(n, blockSize)
```

that fills in all the required information for mapping a global vector of length n into blocks of size `blockSize` and assigning them to the local memory of each image. Other forms of the constructor, of course, might produce a more irregular decomposition of the global vector with different block sizes on each image and different numbers of blocks on each image.

Vector maps might also be extended to represent matrix maps using one vector map for rows of the matrix and another vector map for the columns of the matrix. Maps for irregular sparse matrices may also be designed where the objects represented in the map are sparse blocks of the matrix arranged and distributed in specific ways to match the requirements of specific algorithms for direct or iterative equation solvers. Furthermore, distributed data structures may be redefined dynamically to reflect changes in the physical system as the computation advances in time. A new map is created and the data is redistributed from the old map to the new map without having to change the details of the numerical algorithm.

Blocked Vectors

A blocked vector object,

```
Type :: BlockedVector
Type (VectorMap), pointer, private :: map
real, allocatable, target, private :: block(:, :)
contains
procedure, public, pass(this) :: getPointerToBlock
procedure, public, pass(this) :: haloExchange
:
end type BlockedVector
```

contains the data for each block in an allocatable component `block(:, :)` and a map that describes

the distribution of the global vector. One form of the constructor for a concrete object of this type

```
Type(BlockedVector) :: v
v = BlockedVector(map)
```

accepts an existing vector map as an argument. The constructor uses the map to determine the number of blocks and the block sizes for each image and allocates space appropriately,

```
Type(BlockedVector) &
function BlockedVector(map) result(this)
  Type(VectorMap), target, intent(in) :: map
  this%map=>map
  k=map%getNumberOfLocalBlocks()
  do i=1,k
    n = map%getBlockSize(i)
    allocate(this%block(0:n+1,i))
  end do
end function BlockedVector
```

It also assigns its vector map component to point to the dummy map argument. Notice that the constructor has allocated halo cells of width one to each block. The halo width may be an optional argument to the constructor and may in fact be zero.

Application to Partial Differential Equations

Consider a finite difference scheme applied to the one-dimensional shallow water equations [8]. The partial differential equations, defined by Cahn [4] and by Arakawa and Lamb [1], for the surface height $h(x, t)$ and the two velocity components $u(x, t)$ and $v(x, t)$, as functions of the space variable x and the time variable t , are the equations [1, Eqs. 23–25, p. 183]

$$\begin{aligned} \frac{\partial u}{\partial t} - Fv + G \frac{\partial h}{\partial x} &= 0 \\ \frac{\partial v}{\partial t} + Fu &= 0 \\ \frac{\partial h}{\partial t} + H \frac{\partial u}{\partial x} &= 0. \end{aligned}$$

In these equations, F is the Coriolis frequency, G is the acceleration of gravity, and H is the mean height of the surface, which is assumed to be small relative to the width of the space interval.

The fields u , v , h are represented as blocked vectors declared as coarrays,

```
type(BlockedVector), codimension[*] :: h, u, v
```

The constructor for the vector map builds a map,

```
type(VectorMap) :: map
map=VectorMap(n, k, p, w)
```

that describes a field with n grid points cut into blocks of size k distributed over p images. The halo width w equals one, wide enough for a two-point stencil for the first-order difference operator. Three calls to the constructor for a blocked vector create the three fields h , u , and v ,

```
h=BlockedVector(map)
u=BlockedVector(map)
v=BlockedVector(map)
h=h0
```

based on the same predefined vector map guaranteeing that all three fields have the same distribution. The statement $h=h0$ uses an overloaded assignment statement to initialize the field h from the array $h0(:)$ containing its initial values. The other fields have initial value zero set by their constructors.

Having created field objects, the programmer decides to let each image perform work on the local blocks that it owns. For each of its local blocks, an image obtains the length of the block and a pointer into the block, with or without halos depending on how it is used in the difference formula. Each image performs the appropriate finite difference operation independently of the others. Synchronization among images occurs within the halo exchange operation, which uses coarray syntax internally to update overlapping halo regions. With a loop over some predetermined number of time steps, $tMax$, the code might look like the following:

```
do t=1,tMax
  do b=1,u%getNumLocalBlocks()
    m = u%getBlockLength(b)
    hPtr => h%pointerToBlock(b)
    uPtr => u%pointerToBlockWithHalo(b)
    hPtr(1:m) = hPtr(1:m) &
    -0.5*H*(dt/dx)*(uPtr(2:m+1) &
    -uPtr(0:m))
  end do
  call h%haloExchange()
  do b=1,u%getNumLocalBlocks()
    m = u%getBlockLength(b)
    hPtr => h%pointerToBlockWithHalo(b)
    uPtr => u%pointerToBlock(b)
    vPtr => v%pointerToBlock(b)
    uPtr(1:m) = uPtr(1:m)+F*dt*vPtr(1:m) &
```

```

    - 0.5*G* (dt/dx) * (hPtr(2:m+1) &
    - hPtr(0:m))
vPtr(1:m) = vPtr(1:m) - F*dt*uPtr(1:m)
end do
call u%haloExchange()
call v%haloExchange()
end do

```

All the details of data distribution and all the details of how to exchange data between objects is hidden from the programmer. The blocked vector objects themselves contain all the necessary information, and the procedures associated with them know how to perform the required operations.

Summary

Fortran is a modern programming language. It is an object-oriented language, defined by the Fortran 2003 standard, and it is also a parallel language, defined by the Fortran 2008 standard. Programmers are able to develop modern application codes using object-oriented design combined with the coarray parallel programming model. Development within a single language avoids many of the complications that occur from differences between languages. New compilers will support the model and will make application codes portable across all platforms, and hardware vendors are designing new architectures that will support the model effectively.

Related Entries

- [Chapel \(Cray Inc. HPCS Language\)](#)
- [Fortran 90 and Its Successors](#)
- [MPI \(Message Passing Interface\)](#)
- [OpenMP](#)
- [SPMD Computational Model](#)
- [Titanium](#)
- [UPC](#)

Bibliography

1. Arakawa A, Lamb VR (1977) Computational design of the basic dynamical processes of the UCLA general circulation model. *Meth Comput Phys* 17:173–265
2. Balaji V, Clune TL, Numrich RW, Womack BT (2005) An architectural design pattern for problem decomposition. In: Workshop on patterns in high performance computing, Champaign-Urbana, 4–6 May 2005
3. Burton PM, Carruthers B, Fisher GS, Johnson BH, Numrich RW (2001). Converting the halo-update subroutine in the met office unified model to co-array fortran. In: Zwiehofer W, Kreitz N (eds) *Developments in teracomputing: proceedings of the ninth ECMWF workshop on the use of high performance computing in meteorology*, Reading, 13–17 Nov 2000. World Scientific, pp 177–188

4. Cahn A Jr (1945) An investigation of the free oscillations of a simple current system. *J Meteorol* 2(2):113–119
5. Fox GC, Otto SW, Hey AJG (1987) Matrix algorithms on a hypercube I: matrix multiplication. *Parallel Comput* 4:17–31
6. Gamma E, Helm R, Johnson R, Vlissides J (1995) *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Reading
7. Metcalf M, Reid J, Cohen M (2004) *Fortran 95/2003 explained*. Oxford University Press, Oxford
8. Numrich RW (2005) A parallel numerical library for co-array Fortran. In: *Parallel processing and applied mathematics: proceedings of the sixth international conference on parallel processing and applied mathematics (PPAM05)*, Poznan, 11–14 Sept 2005. Lecture notes in computer science, LNCS, vol 3911. Springer, pp 960–969
9. Numrich RW (2005) Parallel numerical algorithms based on tensor notation and co-array Fortran syntax. *Parallel Comput* 31:588–607
10. Numrich RW, Reid J, Kim K (1998) Writing a multigrid solver using co-array Fortran. In: Kågström B, Dongarra J, Elmroth E, Waśniewski J (eds) *Applied parallel computing: large scale scientific and industrial problems*, 4th international workshop, PARA98, Umeå. Lecture notes in computer science, vol 1541. Springer, pp 390–399
- II. Reid J (2010) Coarrays in the next Fortran standard. ISO/IEC JTC1/SC22/WG5 N1824

Code Generation

CÉDRIC BASTOUL

University Paris-Sud 11 - INRIA Saclay Île-de-France,
Orsay, France

Synonyms

[Polyhedra scanning](#)

Definition

Parallel code generation is the action of building a parallel code from an input sequential code according to some scheduling and placement information. Scheduling specifies the desired order of the statement instances with respect to each other in the target code. Placement specifies the desired target processor for each statement instance.

Discussion

Introduction

Exhibiting parallelism in a sequential code may require complex sequences of transformations. When they come from an expert in program optimization, they are usually expressed by way of directives like *tile* or *fuse* or *skew*. When they come from a compiler, they are typically formulated as functions that map every execution of every statement of the program in “time” (*scheduling*), to order them in a convenient way, and in “space” (*placement*), to distribute them among various processors.

Parallel code generation is the step in any program restructuring tool or parallelizing compiler that actually generates a target code which implements the user directives or the compiler mapping functions.

This task is challenging in many ways. *Feasibility* is the first challenge. Reconstructing a program with respect to scheduling and placement information at the statement execution level may seem an impossible problem at first sight. It has been addressed by working on a convenient representation of the problem itself, as it will be described momentarily. *Scalability* is another important issue. As compiler vendors try to ensure that compile time is (almost) linear in the code length, the code generation step must be fast enough to be integrated into production tools. *Quality* of the generated code must be paramount. The generated code should not be too long and should not include heavy control overhead that may offset the optimization it is enabling. Finally, *flexibility* must be provided to allow a large span of transformations on a large set of programs.

Representation of the Problem

To solve the code generation problem, it needs to be formalized in some way. A mathematical representation, known as the *polyhedral model* (also referred in the literature as the *polytope model* or the *polyhedron model*), is a convenient abstraction. It allows the description of the problem in a compact and expressive way. This representation is also the key to solving it efficiently thanks to powerful and scalable mathematical libraries as described in Sect. on ▶ Scanning Polyhedra. This model, with slight variations, has been used in most successful work on parallel code generation.

Two mathematical objects need to be defined for the parallel code generation problem. First, *iteration domains* provide the relevant information for code generation from the input code, they are detailed in Sect. on Representing Statement Instances: Iteration Domains. Second, *space-time mapping functions* provide the ordering and placement information to be implemented by the target code. They are described in Sect. on Representing Order and Placement: Mapping Functions.

Representing Statement Instances: Iteration Domains

The key aspect of the polyhedral model is to consider *statement instances*. A statement instance is *one* particular execution of a statement. Each instance of a statement that is enclosed inside a loop can be associated with the value of the outer loop counters (also called *iterators*). For instance, let us consider the polynomial multiply code in Fig. 1: the instance of statement S1 for $i = 2$ is $z[2] = 0$.

In the polyhedral model, statements are considered as functions of the outer loop counters that may produce statement instances: instead of simply “S1,” the notation $S1(i)$ is preferred. For instance, statement S1 for $i = 2$ is written $S1(2)$ and statement S2 for $i = 4$ and $j = 2$ is written $S1(4)_2$. The vector of the iterator values is called the *iteration vector*.

Obviously, dealing with statement instances does not mean that unrolling all loops is necessary. First because there would probably be too many instances to deal with, and second because the number of instances may not be known. For example, when the loops are bounded with constants that are unknown at compile time (called “parameters”), for instance, n in the example code in Fig. 1. A compact way to represent all the instances of a given statement is to consider the set of all possible values of its iteration vector. This set is called the statement’s *iteration domain*. It can be conveniently described by all the constraints on the various iterators that the statement depends on. When those constraints

```
do i = 1, n
  z[i] = 0
  do i = 1, n
    do j = 1, n
      z[i+j] = z[i+j] + x[i] * y[j] ! s2
    ! s1
```

Code Generation. Fig. 1 Polynomial multiply kernel

$x = a + b ! S1$ $y = c + d ! S2$ $z = a * b ! S3$	$\theta_{S1} = 1$ $\theta_{S2} = 2$ $\theta_{S3} = 1$	$t = 1$!\$OMP SECTIONS !\$OMP SECTION $x = a + b ! S1$!\$OMP SECTION $z = a * b ! S3$!\$OMP END SECTIONS $t = 2$ $y = c + d ! S2$
a	b	c
Original code	Scheduling	Target code

Code Generation. Fig. 2 One-Dimensional scheduling example

are affine and depend only on the outer loop counters and some parameters, the set of constraints defines a *polyhedron* (more precisely this is a \mathbb{Z} -*polyhedron*, but *polyhedron* is used for short). Hence the name “polyhedral model.” A matrix representation with the following general form for any statement S is used to facilitate the manipulation of the affine constraints:

$$\mathcal{D}_S = \{\mathbf{x}_S \in \mathbb{Z}^{n_S} \mid A_S \mathbf{x}_S + \mathbf{a}_S \geq \mathbf{0}\}$$

where \mathbf{x}_S is the n_S -dimensional iteration vector, A_S is a constant matrix and \mathbf{a}_S is a constant vector, possibly parametric. For instance, here are the iteration domains for the polynomial multiply example in Fig. 1:

- $\mathcal{D}_{S1} = \left\{ \begin{pmatrix} i \end{pmatrix} \in \mathbb{Z} \mid \begin{bmatrix} 1 \\ -1 \end{bmatrix} \begin{pmatrix} i \end{pmatrix} + \begin{pmatrix} -1 \\ n \end{pmatrix} \geq \mathbf{0} \right\}$
- $\mathcal{D}_{S2} = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2 \mid \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \\ -1 \\ n \end{pmatrix} \geq \mathbf{0} \right\}$

Representing Order and Placement: Mapping Functions

Iteration domains do not provide any information about the order in which statement instances have to be executed, nor do they inform about the processor that has to execute them. Such information is provided by other mathematical objects called *space-time mapping functions*. They associate each statement instance with a logical date *when* it has to be executed and a processor coordinate *where* it has to be executed. In the literature, the part of those functions dedicated to time is called *scheduling* while the part dedicated to space is called *placement* (or *allocation*, or *distribution*).

In the case of *scheduling*, the logical dates express at which time a statement instance has to be executed, with respect to the other statement instances. It is typically denoted θ_S for a given statement S . For instance, let us consider the three statements in Fig. 2a and their scheduling functions in Fig. 2b. The first and third statements have to be executed both at logical date 1. This means they can be executed in parallel at date 1 but they have to be executed before the second statement since its logical date is 2. The target code implementing this scheduling using OpenMP pragmas is shown in Fig. 2c, where a fictitious variable t stands for the time. It can be seen that at time $t = 1$, both $S1$ and $S3$ are run in parallel, while $S2$ is executed afterward at time $t = 2$.

Logical dates may be multidimensional, like clocks: the first dimension corresponds to days (most significant), next one is hours (less significant), the third to minutes, and so on. The order of multidimensional dates with a decreasing significance for each dimension is called the *lexicographic* order. Again, it is not possible to assign one logical date to each statement instance for two reasons: this would probably lead to an intractable number of logical dates and the number of instances may not be known at compile time. Hence, a more compact representation called the *scheduling function* is used. A scheduling function associates a logical date with each statement instance of a statement. They have the following form for a statement S :

$$\theta_S(\mathbf{x}_S) = T_S \mathbf{x}_S + \mathbf{t}_S,$$

where \mathbf{x}_S is the iteration vector, T_S is a constant matrix, and \mathbf{t}_S is a constant vector, possibly parametric. Scheduling functions can easily encode a wide range of usual transformations such as skewing, interchange,

reversal, shifting tiling, etc. Many program transformation frameworks have been proposed on top of such functions, the first significant one being UTF (Unified Transformation Framework) by Kelly and Pugh in 1993 [8].

Placement is similar to scheduling, only the semantics is different: instead of logical dates, a placement function π_S associates each instance of statement S with a processor coordinate corresponding to the processor that has to execute the instance.

A space-time mapping function σ_S is a multidimensional function embedding both space and time information for statement S : some dimensions are devoted to scheduling while some others are dedicated to placement. For instance, a compiler may suggest the following space-time mapping for the polynomial multiply code shown in Sect. on Representing Statement Instances: Iteration Domains. Its first dimension is a placement that corresponds to a wavefront parallelism for S_2 and improves locality by executing the initialization of an array element by S_1 on the same processor where it is used by S_2 . The second dimension is a very simple constant scheduling that ensures the initialization of the array element is done before its use (it is usual to add the identity schedule at the last dimensions; however this will not be necessary for the continuation of this example):

$$\begin{aligned} \bullet \quad \sigma_{S1} \left(\begin{array}{c} i \\ j \end{array} \right) &= \left[\begin{array}{c} 1 \\ 0 \end{array} \right] \left(\begin{array}{c} i \\ j \end{array} \right) + \left(\begin{array}{c} 0 \\ 0 \end{array} \right) \\ \bullet \quad \sigma_{S2} \left(\begin{array}{c} i \\ j \end{array} \right) &= \left[\begin{array}{cc} 1 & 1 \\ 0 & 0 \end{array} \right] \left(\begin{array}{c} i \\ j \end{array} \right) + \left(\begin{array}{c} 0 \\ 1 \end{array} \right) \end{aligned}$$

While working in the polyhedral representation, the semantics of each dimension is not relevant: After code generation, each dimension will be translated to some loops that can be post-processed to become parallel or sequential according to their semantics (obviously semantics information can be used to generate a better code, but this is out of the scope of this introduction).

Putting Everything Together

Iteration domains can be extracted directly by analyzing the input code. They represent for each statement the set of their instances. In particular, they do not encode

any ordering information: Iteration domains are nothing but “bags” of unordered statement instances. On the opposite, the space/time mapping functions, typically computed by an optimizing or parallelizing algorithm, provide the ordering information for each statement instance. It is necessary to collect all this information into a polyhedral representation before the actual code generation. There exist two ways to achieve this task. Let us consider an iteration domain defined by the system of affine constraints $Ax + a \geq \mathbf{0}$ and the transformation function leading to a target index $y = Tx$. Any of the following formulas can be chosen to build the target polyhedron \mathcal{T} that embeds both instance and ordering information:

Inverse Transformation By noticing that $x = T^{-1}y$ it follows that the transformed polyhedron in the new coordinate system can be defined by:

$$\mathcal{T} : \{y \mid [AT^{-1}]y + a \geq \mathbf{0}\}.$$

Generalized Change of Basis Alternatively, new dimensions corresponding to the ordering in leading positions can be introduced (note that in the following formula, constraints “above” the line are equalities while constraints “under” the line are inequalities):

$$\mathcal{T} : \left\{ \left(\begin{array}{c} y \\ x \end{array} \right) \mid \left[\begin{array}{c|c} I & -T \\ 0 & A \end{array} \right] \left(\begin{array}{c} y \\ x \end{array} \right) + \left(\begin{array}{c} -t \\ a \end{array} \right) \geq \mathbf{0} \right\}.$$

The inverse transformation solution has been introduced since the seminal work on parallel code generation by Ancourt and Irigoin [1]. It is simple and compact but has several issues: the transformation matrix must be invertible, and even when it is invertible, the target polyhedra may embed some integer points that have no corresponding elements in the iteration domain (this happens when the transformation matrix is not unimodular, i.e., whose determinant is neither +1 nor -1). This necessitates specific code generation processing, briefly discussed in Sect. on Fourier–Motzkin Elimination-Based Scanning Method. The second formula is attributed to Le Verge, who named it the *Generalized Change of Basis* [11]. It does not require any property on the transformation matrix. Nevertheless, the additional dimensions may increase the complexity of the code generation process. It has been rediscovered independently from Le Verge’s work and used in production code generators only recently [2].

Both formulas are used, and possibly mixed, in current code generation tools, depending on the desired transformation properties. For instance, to apply the space-time mapping of the polynomial multiply proposed in Sect. on Representing Order and Placement: Mapping Functions, it is convenient to use the Generalized Change of Basis because the transformation matrices are not invertible:

$$\bullet \quad T_{S1} = \left\{ \begin{pmatrix} p \\ t \\ i \end{pmatrix} \in \mathbb{Z}^2 \mid \left[\begin{array}{c|cc} 1 & 0 & -1 \\ 0 & 1 & 0 \\ \hline 0 & 0 & 1 \\ 0 & 0 & -1 \end{array} \right] \begin{pmatrix} p \\ t \\ i \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ -1 \\ n \end{pmatrix} \geq \mathbf{0} \right\}$$

$$\bullet \quad T_{S2} = \left\{ \begin{pmatrix} p \\ t \\ i \\ j \end{pmatrix} \in \mathbb{Z}^3 \mid \left[\begin{array}{c|cc} 1 & 0 & -1 & -1 \\ 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 \end{array} \right] \begin{pmatrix} p \\ t \\ i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \\ -1 \\ n \\ -1 \\ n \end{pmatrix} \geq \mathbf{0} \right\}$$

In the target polyhedra, whatever the chosen formula, the order of the dimensions is meaningful: The ordering is encoded as the lexicographic order of the integer points. Thus, the parallel code generation problem is reduced to generating a code that enumerates the integer points of several polyhedra, with respect to the lexicographic ordering.

Scanning Polyhedra

Once the target code information has been encoded into some polyhedra that embed the iteration spaces as well as the scheduling and placement constraints, the code generation problem translates to a *Polyhedra scanning* problem. The problem here is to find a code (preferably efficient) visiting each integral point of each polyhedra, once and only once, with respect to the lexicographic order. Three main methods have been successful in doing this. Fourier–Motzkin elimination-based techniques have been the very first, introduced by the seminal work of Ancourt and Irigoin [1]. They are discussed briefly in Sect. on Fourier–Motzkin Elimination-Based Scanning Method. While Fourier–Motzkin-based techniques aim at generating loop nests, an alternative method based on Parametric Integer Programming has been suggested by Boulet and Feautrier to generate lower-level codes [3]. This method is discussed briefly in

Sect. on Parametric Integer Programming-Based Scanning Method. Lastly, Quilleré, Rajopadhye, and Wilde showed how to take advantage of high-level polyhedral operations to generate efficient codes directly [14]. As this later technique is now widely adopted in production environments, it is discussed in some depth in Sect. on QRW-Based Scanning Method.

Fourier–Motzkin Elimination-Based Scanning Method

Ancourt and Irigoin [1] proposed in 1991 the first solution to the polyhedron scanning problem. Their work is based on the Fourier–Motzkin pair-wise elimination technique [15]. The scope of their method was quite restrictive since it could be applied to only one polyhedron, with unimodular transformation matrices. The basic idea was, for each dimension from the first one (outermost) to the last one (innermost), to project the polyhedron onto the axis and to deduce the corresponding loop bounds. For a given dimension i_k , the Fourier–Motzkin algorithm can establish that $L(i_1, \dots, i_{k-1}) + \mathbf{l} \leq c_k i_k$ and $c_k i_k \leq U(i_1, \dots, i_{k-1}) + \mathbf{u}$, where L and U are constant matrices, \mathbf{l} and \mathbf{u} are constant vectors of size m_l and m_u respectively, and c_k is a constant. Thus, the corresponding scanning code for the dimension i_k can be derived:

```

...
do  $i_k = \text{MAX}_{j=1}^{m_l} [(L_j(i_1, \dots, i_{k-1}) + l_j)/c_k],$ 
     $i_k \leq \text{MIN}_{j=1}^{m_u} [(U_j(i_1, \dots, i_{k-1}) + u_j)/c_k]$ 
...
Body
```

The main drawback of this method is the large amount of redundant control since eliminating a variable with the Fourier–Motzkin algorithm may generate up to $n^2/4$ constraints for the loop bounds where n is the initial number of constraints. Many of those constraints are redundant and it is necessary to remove them for efficiency.

Most further works tried to extend this first technique in order to reduce the redundant control and to deal with more general transformations. Le Fur presented a new redundant constraint elimination policy by using the simplex method [10]. Li and Pingali [13] as well as several other authors proposed to relax the unimodularity constraint of the transformation to an invertibility constraint by using the Hermite Normal

Form [15] to avoid scanning “holes” in the polyhedron. Griebl, Lengauer and Wetzel [5] relaxed the constraints of code generation further to transformation matrices with non-full rank, and also presented preliminary techniques for scanning several polyhedra using a single loop nest. Finally, Kelly, Pugh, and Rosser showed how to scan several polyhedra in the same code by generating a naive perfectly nested code and then (partly) eliminating redundant conditionals [9]. Their implementation relies on an extension of the Fourier–Motzkin technique called the Omega test. The implementation of their algorithm within the Omega calculator is one of the most popular parallel code generators [7].

Parametric Integer Programming-Based Scanning Method

Boulet and Feautrier proposed in 1998 a parallel code generation technique that relies on Parametric Integer Programming (PIP for short) to build a code for scanning polyhedra [3]. The PIP algorithm computes the lexicographic minimal integer point of a polyhedron. Because the minimum point may not be the same depending on the parameter values, it is returned as a tree of conditions on the parameters where each leaf is either the solution for the corresponding parameter constraints or \perp (called *bottom*), that is, no solution for those parameter constraints.

The basic idea of the Boulet and Feautrier algorithm (in the simplified case of scanning one polyhedron) is to find the first integer point of the polyhedron, called *first*, then to build a function *next* which for a given integer point returns the next integer point in the polyhedron according to the lexicographic ordering. Both *first* and *next* computations can be expressed as a problem of finding the lexicographic minimum in a polyhedron. Finally, the code can be built according to the following canvas, where x is an integer point of the polyhedron that represents the iteration domain:

```

 $x = \text{first}$ 
1 if  $x = \perp$  then goto 2
    Body
     $x = \text{next}$ 
    goto 1
2 ...

```

Generalizing this method to many polyhedra implies combining the different trees of conditions and subsequent additional control cost and code duplication. While this technique has no widely used implementation, it is quite different than the others since it does not aim at generating high-level loop statements. This property may be relevant for specific targets, for example, when the generated code is not the input of a compiler but of a high-level synthesis tool.

QRW-Based Scanning Method

Quilleré, Rajopadhye, and Wilde proposed in 2000 the first code generation algorithm that builds a target code without redundant control *directly* [14]. While previous schemes started from a generated code with some redundant control and then tried to improve it, their technique (referred as the QRW algorithm) never fails at removing control, and the processing is easier. Eventually it generates a better code more efficiently.

The QRW algorithm is a generalization to several polyhedra of the work of Le Verge, Van Dongen and Wilde on loop nest synthesis using polyhedral operations [12]. It relies on high-level polyhedral operations (polyhedral intersection, union, projection, etc.), which are available in various existing polyhedral libraries. The basic mechanism is, starting from (1) the list of polyhedra to scan and (2) a polyhedron encoding the constraints on the parameters called the *context*, to recursively generate each level of the abstract syntax tree of the scanning code (AST).

The algorithm is sketched in Fig. 3 and a simplified example is shown in Figs. 4–6. It corresponds to the generation of the code implementing the polynomial multiply space-time mapping introduced in Sect. on Representing Order and Placement: Mapping functions. Its input is the list of polyhedra to scan, the context and the first dimension to scan. This corresponds to Fig. 4 in our example, with the first dimension to scan being p . The first step of the algorithm intersects the polyhedra with the context to ensure no instance outside the context will be executed. Then it projects them onto the first dimension and separates the projections into disjoint polyhedra. For instance, for two polyhedra, this could correspond to one domain where the first polyhedron is “alone,” one domain where the second polyhedron is “alone” and one domain where the two polyhedra coexist. This is depicted in Fig. 5 for our example: it depicts the projection onto the p axis and the

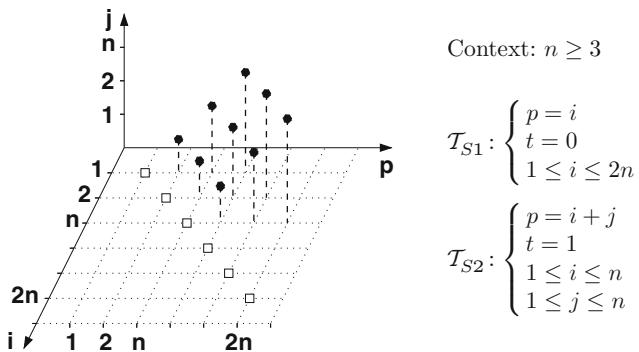
QRW: build a polyhedron scanning code AST without redundant control.

Input: a polyhedron list, a context C , the current dimension d .

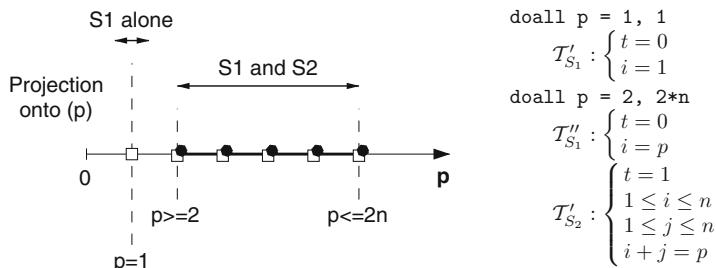
Output: the AST of the code scanning the input polyhedra.

1. Intersect each polyhedron in the list with the context C
2. Project the polyhedra onto the outermost d dimensions
3. Separate these projections into disjoint polyhedra (this generates loops for dimension d and new lists for dimension $d+1$)
4. Sort the loops to respect the lexicographic order
5. Recursively generate loop nests that scan each new list with dimension $d+1$, under the context of the dimension d
6. Return the AST for dimension d

Code Generation. Fig. 3 Sketch of the QRW Code Generation Algorithm



Code Generation. Fig. 4 QWR Code Generation Example (1/3): Polyhedra to Scan and Context Information. The graphical representation does not show the degenerated scheduling dimension t



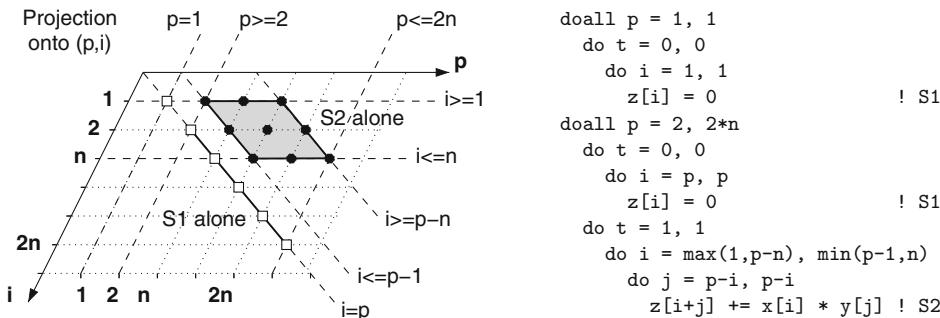
Code Generation. Fig. 5 QWR Code Generation Example (2/3): Intersection with the Context, Projection, and Separation onto the First Dimension. Two disjoint polyhedra are created: one where S_1 is alone on p (it has only one integer point but a loop is generated to scan it, for consistency) and one where S_1 and S_2 are together on p . In the right side, the new polyhedra to scan have been intersected with the context (for the next step, p is a parameter as well as n)

separation (it can be seen here that the domain where S_2 is “alone” is empty). The constraints on dimension p for the resulting polyhedra give directly the loop bounds.

As the semantics of the placement dimension is to distribute instances across different processors, this loop is parallel. Then the algorithm recursively generates

the next dimension loops for each disjoint polyhedron separately. The final result is shown in Fig. 6 for our example.

The QRW algorithm is simple and efficient in practice, despite the high theoretical complexity of most polyhedral operations. However, in its basic form, it



Code Generation. Fig. 6 QWR Code Generation Example (3/3): Recursion on the Next Dimensions. First, the projection/separation on (p, t) is done. It is trivial because t is a constant in every polyhedron: it only enforces disjunction and ordering of the polyhedra inside the second `doall` loop. Next the same processing is applied for (p, t, i) : the loop bounds of the remaining dimensions can be deduced from the graphical representation (the trivial dimension t is not shown)

tends to generate codes with costly modulo operations, and the separation process is likely to result in very long codes. Several extensions to this algorithm have been proposed to overcome those issues [2, 16]. CLooG, a popular implementation of the extended QRW technique demonstrated effectiveness of the algorithm [2]. It is now used in production environments such as in GCC or IBM XL.

Parallel Code Generation Today

For a long time, scheduling and placement techniques were many steps forward code generation capabilities. In 1992, Feautrier provided a general scheduling technique for multiple polyhedra and general affine functions [4]. At this time, the only code generation algorithm available had been designed in 1991 by Ancourt and Irigoin and supported only one polyhedron and unimodular scheduling functions [1]. Some scheduling functions had to wait for nearly one decade to be successfully applied by a code generator.

Since then, the challenge of feasibility has been tackled: State-of-the-art parallel code generators can handle any affine transformation for many iteration domains. Moreover, the scalability of code generators is good enough to enable parallel code generation as an option in production compilers. However, the quality of the generated code is still not guaranteed. Summarily, code generators are very good for simple (unimodular) transformations, reasonably good when the coefficients of the transformation functions are small and unpredictable in the general case. The flexibility challenge

is also solved only partly because only regular codes that fit the polyhedral model can be processed and only affine transformations can be applied.

Future Directions

Two challenges of parallel code generation are partly solved: quality and flexibility. To achieve the best results, autoparallelizers have to take into account some constraints related to code generation that may conflict with the extraction of parallelism, for example, limiting the absolute value of the transformation coefficients or relying on unimodular transformations. However, there exists an infinity of transformations that implement the same mapping but have different properties with respect to code generation. Finding “code generation friendly” equivalent transformations is a promising solution to enhance the generated code quality.

Several directions are under investigation to provide parallel code generation with more flexibility. Irregular extensions have been successfully implemented to some polyhedral code generators and ambitious techniques based on polynomials instead of affine expressions may be the next step for parallel code generation [6].

Related Entries

- ▶ [Loop Nest Parallelization](#)
- ▶ [Omega Test](#)
- ▶ [Parallelization, Automatic](#)
- ▶ [Polyhedron Model](#)
- ▶ [R-Stream Compiler](#)
- ▶ [Scheduling Algorithms](#)
- ▶ [Unimodular Transformations](#)

Bibliographic Notes and Further Reading

We detailed in Sect. on ►[Scanning Polyhedra](#) the three main techniques designed for parallel code generation. The reader will find a deeper level of details in the related papers. Kelly, Pugh, and Rosser's paper on code generation for multiple mappings provides the extensive description of the techniques behind the Omega Code Generator [9]. Boulet and Feautrier's paper on code generation without do-loops gives thorough depiction of the PIP-based code generation technique [3]. Finally, the details of the most powerful code generation technique known so far are provided in Quilleré, Rajopadhye, and Wilde's paper on generation of efficient nested loops from polyhedra [14]. This reading is complemented by Bastoul's paper, which details several extensions to their algorithm and demonstrates robustness of the extended technique for production compilers [2].

Bibliography

1. Ancourt C, Irigoin F (1991) Scanning polyhedra with DO loops. In: 3rd ACM SIGPLAN symposium on principles and practice of parallel programming, Cologne, Germany, pp 39–50
2. Bastoul C (2004) Code generation in the polyhedral model is easier than you think. In: IEEE international conference on parallel architectures and compilation techniques (PACT'04), Juan-les-Pins, pp 7–16
3. Boulet P, Feautrier P (1998) Scanning polyhedra without do-loops. In: IEEE international conference on parallel architectures and compilation techniques (PACT'98), Paris, France, pp 4–11
4. Feautrier P (1992) Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *Int J Parallel Program* 21(6):389–420
5. Griebel M, Lengauer C, Wetzel S (1998) Code generation in the polytope model. In: Proceedings of the international conference on parallel architectures and compilation techniques (PACT'98), pp 106–111
6. Größlinger A (2009) The challenges of non-linear parameters and variables in automatic loop parallelisation. Doctoral thesis, Fakultät für Informatik und Mathematik, Universität Passau
7. Kelly W, Maslov V, Pugh W, Rosser E, Shpeisman T, Wonnacott D (1996) The Omega library. Technical report, University of Maryland
8. Kelly W, Pugh W (1993) A framework for unifying reordering transformations. Technical Report UMIACS-TR-92-126.1, University of Maryland Institute for Advanced Computer Studies
9. Kelly W, Pugh W, Rosser E (1995) Code generation for multiple mappings. In: Frontiers'95 Symposium on the frontiers of massively parallel computation, McLean, VA
10. Le Fur M (1996) Scanning parameterized polyhedron using Fourier-Motzkin elimination. *Concurrency – Pract Exp* 8(6): 445–460
11. Le Verge H (1995) Recurrences on lattice polyhedra and their applications, April 1995. Unpublished work based on a manuscript written by H. Le Verge just before his untimely death in 1994
12. Le Verge H, Van Dongen V, Wilde D (1994) Loop nest synthesis using the polyhedral library. Technical Report 830, IRISA
13. Li W, Pingali K (1994) A singular loop transformation framework based on non-singular matrices. *Int J Parallel Program* 22(2):183–205
14. Quilleré F, Rajopadhye S, Wilde D (2000) Generation of efficient nested loops from polyhedra. *Int J Parallel Program* 28(5): 469–498
15. Schrijver A (1986) Theory of linear and integer programming. Wiley, New York
16. Vasilache N, Bastoul C, Cohen A (2006) Polyhedral code generation in the real world. In: Proceedings of the international conference on compiler construction (ETAPS CC'06), LNCS 3923, Vienna, Austria, pp 185–201

Collect

► Allgather

Collective Communication

ROBERT VAN DE GEIJN¹, JESPER LARSSON TRÄFF²

¹The University of Texas at Austin, Austin, TX, USA

²University of Vienna, Vienna, Austria

Synonyms

[Group communication](#); [Inter-process communication](#)

Definition

Collective communication is communication that involves a group of processing elements (termed *nodes* in this entry) and effects a data transfer between all or some of these processing elements. Data transfer may include the application of a reduction operator or other transformation of the data. Collective communication functionality is often exposed through library interfaces or language constructs. Collective communication is a natural extension of the message-passing paradigm.

Discussion

Introduction

Many commonly encountered communication patterns and computational operations involving data distributed across sets of processing elements (nodes) can be represented as *collective communication* in which all nodes in a (sub)set of nodes collaborate to carry out a specific data redistribution or data reduction operation. Making such operations available in parallel programming languages, interfaces, or libraries has a number of advantages:

- It simplifies parallel and distributed programming, relieving the user from explicitly expressing complex communication patterns by means of more primitive communication operations.
- It raises the level of abstraction at which algorithms can be expressed and abstracts away details of the underlying parallel communication system.
- It makes it easier to reason about algorithm correctness and communication cost.
- It improves the functional portability of applications and contributes toward performance portability.
- It makes it possible to schedule complex communication patterns to efficiently exploit capabilities and specific properties of the underlying communication system.
- It introduces a productive separation of concerns between the application and interface developers.

Collective communication operations are found in some form in most parallel programming interfaces and languages, notably in the *Message-Passing Interface* (MPI) [14], in Partitioned Global Address Space (PGAS) languages like UPC [5], in libraries for *Bulk Synchronous Processing* (BSP) [2, 11] and many others, but also in application-specific and special purpose libraries. By capturing patterns of communication and computation, collective communication is relevant not only for distributed memory parallel systems, but also for systems with shared memory between all or subsets of nodes. Collective communication indeed serves to abstract away such system characteristics, and can serve as a higher-level bridging model for ensuring (performance) portability of applications across systems with different communication capabilities. Research on efficient algorithms for collective communication under

various computation and communication models and on practical implementations has been intensive over the past decades, and many good algorithms have found their way into common practice.

Many algorithms and applications are naturally and effectively expressed in a coarse-grained style as a sequence of local computations and collective communication operations. It has been argued that collective communication is more fundamental than explicit point-to-point or one-sided communication for expressing parallel computations [7]. Collective communication is a natural generalization of message-passing point-to-point communication that explicitly involves two nodes, and one-sided communication that explicitly involves only one node.

A Motivating Example

Consider a matrix-vector multiplication $y = Ax$, where A is an $n \times n$ matrix and x and y are column vectors to be solved in parallel. Let the p compute nodes be indexed from 0 to $p - 1$, assume for simplicity that p divides n , and that the input is evenly distributed among the nodes as described below.

Algorithm 1

Partition A , x , and y :

$$A \rightarrow \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_{p-1} \end{pmatrix}, \quad x \rightarrow \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{p-1} \end{pmatrix},$$

$$\text{and } y \rightarrow \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{p-1} \end{pmatrix},$$

where A_k has n/p rows, and x_k and y_k are of size n/p . Assume that initially A_k , x_k , and y_k are assigned only to node k , $k = 0, \dots, p - 1$.

1. Collect all x_k 's on all nodes (allgather) so that the entire x is available on all nodes.

2. In parallel compute $y_k = A_k x$ locally on each node k , $k = 0, \dots, p - 1$.

Step 1 is an example of the collective communication operation commonly referred to as allgather. Each node contributes a subvector and after the operation all nodes have collected the full vector. Locally, each node performs $O(n/p \times n)$ arithmetic operations, and under reasonable communication assumptions $O(n)$ time is required for the allgather step (see entry on ► [Allgather](#)). The local memory consumption is $O(n^2/p + n)$.

Algorithm 2

Partition A , x , and y :

$$A \rightarrow \begin{pmatrix} A_0 & A_1 & \dots & A_{p-1} \end{pmatrix}, \quad x \rightarrow \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{p-1} \end{pmatrix},$$

and $y \rightarrow \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{p-1} \end{pmatrix}$,

where A_k has n/p columns, and x_k and y_k are of size n/p , and assume that initially A_k , x_k , and y_k are assigned only to node k , $k = 0, \dots, p - 1$.

1. In parallel, compute the n element vector $y^{(k)}$ on each node $k = 0, \dots, p - 1$ by $y^{(k)} = A_k x_k$.
2. Compute the vector $y = y^{(0)} + y^{(1)} + \dots + y^{(p-1)}$ and store the final subvector y_k on node k , $k = 0, \dots, p - 1$.

The operation in Step 2 is an example of a collective reduction operation known as reduce-scatter. This operation computes a global result by element-wise summing n element vector contributions from all participating nodes, leaving parts of the result scattered elements over the nodes. Locally, each node performs $O(n \times n/p)$ arithmetic operations, and under reasonable communication assumptions, the reduce-scatter step can be done in $O(n)$ time (see entry on ► [Reduce and scan](#)).

Unless the matrix A has additional structure that can be exploited both in the local computations and in the collective communication, neither Algorithm 1 nor 2 is scalable beyond $p \geq n$ compute nodes. Sequential matrix-vector multiplication takes $O(n^2)$ operations, and parallel efficiency cannot be maintained with increasing p beyond n if n is kept constant. The algorithms illustrate the use of collective operations. Exploiting both ideas and using instead a two-dimensional partitioning of the matrix lead to a scalable algorithm which naturally relies on collective communication, as discussed later in this entry.

Classification of Collective Operations

The basic collective communication operations that will be described next can be divided into either *redistribution* (pure data transfer) or *reduction* operations that also entail a computation on the transferred data. They can further be classified along the following lines.

Rooted/non-rooted: In *rooted* (or *asymmetric*) operations, a specified node (the *root*) is either the sole origin of data to be redistributed or the sole destination of data or results contributed by the nodes involved in the communication. In *non-rooted* (or *symmetric*) operations all nodes contribute and receive data.

For rooted collectives, it is typically assumed that all nodes know the identity of the root node.

Regular/irregular: A collective operation is *regular* if the amount of data contributed or received by each involved node is the same, whereas for *irregular* collective operations, different nodes can contribute and/or receive different amounts of data.

For regular collectives, it is typically assumed that all processes know the amount of data to be redistributed; irregular operations may not make this assumption, and each node may know only the amount of data it has to contribute or receive in the operation.

Homogeneous/nonhomogeneous data: The data involved may either be *homogeneous* as in simple arrays of elements of the same datatype, or *nonhomogeneous* and consist of elements of different types, possibly stored in a nonconsecutive layout.

Synchronizing/non-synchronizing: A collective operation is *synchronizing* if a node starting the collective operation cannot continue its operation before all other nodes have commenced and reached a certain point in the operation. In contrast, in *non-synchronizing* operations, each node may be allowed to continue as soon as data from/to that node required for the operation have been sent and received.

Blocking/nonblocking: *Blocking* collective operations keep each node engaged until it has completed its involvement in the operation. Collectives may also be *nonblocking*, meaning that initiation and completion of the operation are separated. As soon as a node has initiated a nonblocking operation it may perform other tasks, with collective communication logically taking place in the background. Collective communication with nonblocking semantics might be able to leverage this for overlapping communication with other useful (application level) computation.

The rooted/non-rooted dichotomy can alternatively be captured by the trichotomy all-to-all/all-to-one/one-to-all. Not all commonly found collective communication operations follow these taxonomies. Collective reduction operations can additionally be classified according to the types of operations and transformations that are allowed on the data. Finally, the power of programming interfaces with explicit collective communication is characterized not only by the types and set of collective operations included, but also on the

capability to perform collective communication on subsets of nodes.

Commonly Used Collective Communications

The semantics (before-after) of some commonly identified collective communication operations in scientific computing are explained in the following and depicted in Figs. 1–3. Here, x , $x^{(j)}$, and y are vectors of data, and x_k , $x_k^{(j)}$, and y_k are subvectors of x , $x^{(j)}$, and y , respectively. Input vector $x^{(j)}$ is assumed to be owned by node j which means that this vector is stored in some form in the local memory of node j .

Data Redistribution Operations

The first set of collective communications redistribute or duplicate data from one or more nodes to one or more nodes.

Broadcast: One node (the *root*) owns a vector of data, x , that is to be copied to all other nodes. After the operations, all nodes have a copy of that data.

Scatter: One node (the *root*) owns a vector of data, x , that is partitioned into subvectors, x_0, \dots, x_{p-1} . Upon completion, node k owns x_k , $k = 0, \dots, p - 1$.

Gather: The inverse of the scatter operation, the subvectors x_k are gathered into the complete vector x at the *root*.

Allgather: The allgather operation is like the gather operation, except that all nodes receive all of the data. The operation is also known as all-to-all broadcast, concatenation, and gossiping, and is equivalent

Operation	Before			After		
	Node 0	Node 1	Node 2	Node 0	Node 1	Node 2
Broadcast	x			x	x	x
Scatter	Node 0 x_0 x_1 x_2	Node 1	Node 2	Node 0 x_0	Node 1 x_1	Node 2 x_2
Gather	Node 0 x_0	Node 1 x_1	Node 2 x_2	Node 0 x_0 x_1 x_2	Node 1	Node 2

Collective Communication. Fig. 1 Rooted redistribution operations. For illustration node 0 is chosen as root

Operation	Before			After		
	Node 0	Node 1	Node 2	Node 0	Node 1	Node 2
Allgather	x_0	x_1	x_2	x_0	x_0	x_0
All-to-all	$x_0^{(0)}$ $x_1^{(0)}$ $x_2^{(0)}$	$x_0^{(1)}$ $x_1^{(1)}$ $x_2^{(1)}$	$x_0^{(2)}$ $x_1^{(2)}$ $x_2^{(2)}$	$x_0^{(0)}$ $x_0^{(1)}$ $x_0^{(2)}$	$x_1^{(0)}$ $x_1^{(1)}$ $x_1^{(2)}$	$x_2^{(0)}$ $x_2^{(1)}$ $x_2^{(2)}$
Permutation	$x^{(0)}$	$x^{(1)}$	$x^{(2)}$	$x^{(\pi^{-1}(0))}$	$x^{(\pi^{-1}(1))}$	$x^{(\pi^{-1}(2))}$

Collective Communication. Fig. 2 Non-rooted redistribution operations. For the permutation collective, π is a permutation of $\{0, \dots, p - 1\}$ mapping node i to node $\pi(i)$

Operation	Before			After		
	Node 0	Node 1	Node 2	Node 0	Node 1	Node 2
Reduce	$x^{(0)}$	$x^{(1)}$	$x^{(2)}$	$y = \bigoplus_{j=0}^{p-1} x^{(j)}$		
Reduce-scatter	$x_0^{(0)}$ $x_1^{(0)}$ $x_2^{(0)}$	$x_0^{(1)}$ $x_1^{(1)}$ $x_2^{(1)}$	$x_0^{(2)}$ $x_1^{(2)}$ $x_2^{(2)}$	$y_0 = \bigoplus_{j=0}^{p-1} x_0^{(j)}$	$y_1 = \bigoplus_{j=0}^{p-1} x_1^{(j)}$	$y_2 = \bigoplus_{j=0}^{p-1} x_2^{(j)}$
Allreduce	$x^{(0)}$	$x^{(1)}$	$x^{(2)}$	$y = \bigoplus_{j=0}^{p-1} x^{(j)}$	$y = \bigoplus_{j=0}^{p-1} x^{(j)}$	$y = \bigoplus_{j=0}^{p-1} x^{(j)}$
Prefix	$x^{(0)}$	$x^{(1)}$	$x^{(2)}$	$y^{(0)} = \bigoplus_{j=0}^0 x^{(j)}$	$y^{(1)} = \bigoplus_{j=0}^1 x^{(j)}$	$y^{(2)} = \bigoplus_{j=0}^2 x^{(j)}$

Collective Communication. Fig. 3 Rooted and non-rooted reduction operations. For illustration of the reduce collective, node 0 is chosen as root. Only the inclusive all prefix sums operation is depicted

to a gather to some root followed by a broadcast from that root, or to p (simultaneous) gather operations with roots $i = 0, \dots, p - 1$, each gathering the same vector x .

All-to-all: Each node $i, i = 0, \dots, p - 1$ sends subvector $x_k^{(i)}$ to node k for all $k = 0, \dots, p - 1$. It is sometimes emphasized that in contrast to the allgather operation, this is a *personalized* all-to-all *exchange* in that each node contributes a different subvector to each other node. The all-to-all operation is equivalent to p (simultaneous) scatter operations with roots $i = 0, \dots, p - 1$, or p (simultaneous) gather operations each scattering or gathering a different vector $x^{(i)}$. The operation is also sometimes referred to as a *transpose* (of the matrix of subvectors $x_i^{(j)}$).

Permutation: For a given permutation π of the set $\{0, \dots, p - 1\}$ each node $i, i = 0, \dots, p - 1$ sends vector x_i to node $\pi(i)$. After the operation, node $\pi(i)$ owns vector $x^{(\pi^{-1}(i))}$.

The input subvectors x_i and $x_i^{(j)}$ are not necessarily required to have the same number of elements. *Regular* redistribution operations would require subvectors to have the same size, whereas *irregular* collectives would not make this requirement.

Reduction Operations

Nodes often compute partial results that have to be *reduced* (combined) to yield a final result. This is often abstracted as collective communication in the following

way. Let \oplus be an associative, binary operator on the set of elements of the input vectors $x^{(j)}$. The operator is extended to full vectors by element-wise application. More precisely, let $x^{(j)}$ for $j = 0, \dots, p-1$ be vectors of size n . Then

$$x^{(j)} \oplus x^{(k)} = \begin{pmatrix} x_0^{(j)} \oplus x_0^{(k)} \\ x_1^{(j)} \oplus x_1^{(k)} \\ \vdots \\ x_{n-1}^{(j)} \oplus x_{n-1}^{(k)} \end{pmatrix}$$

for any j and $k, j, k = 0, \dots, p-1$.

Collective reduction operations now compute $y = \oplus_{j=0}^k x^{(j)}$, either for the specific $k = p-1$ or for all $k = 0, \dots, p-1$. By the associativity of \oplus this is well defined (brackets can be omitted); it may or may not be assumed/required that the operator \oplus be commutative. The collective reduction operations further differ in how the elements of y are left distributed among the nodes.

Reduce: A designated *root* node receives the result y .

For emphasis, this collective operation is sometimes called *reduction-to-one*.

Allreduce: The result y is duplicated to all nodes.

Reduce-scatter: The result y is distributed among the nodes so that node k ends up with a subvector y_k of y .

All prefix sums: All *prefix sums* $y^{(k)} = \oplus_{j=0}^k x^{(j)}$ are computed and the k th prefix $y^{(k)}$ stored at node $k, k = 0, \dots, p-1$. This collective communication operation is often termed *scan*.

The requirement that \oplus must be associative enables parallelization of the operations, since partial results can be computed concurrently and later combined. Likewise, commutativity may make more efficient implementations possible. Because the prefix sum for node k includes the input vector of node k itself, the operation described is sometimes referred to as the *inclusive* all prefix sums operation. An *exclusive* prefix-sums operation would compute $y^{(k)} = \oplus_{j=0}^{k-1} x^{(j)}$ with some special provision for node 0. Note that the k th inclusive prefix sum can trivially be computed from the exclusive k th prefix sums, but not vice versa, unless an inverse operation is given for \oplus .

Alternatively, some of the reduction patterns above could be formulated as operations on row vectors instead. In this formulation, a single element y would be computed by $y = \oplus_{j=0}^{p-1} \oplus_{i=0}^{n_j-1} x_i^{(j)}$, where n_j is the size of the vector $x^{(j)}$ residing with node j .

Barrier Synchronization

Sometimes applications require nodes to synchronize in the sense that no node shall continue beyond a certain point in its execution, e.g., enter the next stage in a computation before all nodes have reached that stage. A (semantic) barrier to ensure this is typically classified as a collective communication operation, although no data transfer or computation is implied.

Barrier is often used for temporal side effects, achieving an actual synchronization between a (large) set of nodes as sometimes required for bench-marking purposes.

A Motivating Example (continued)

We again discuss $y = Ax$ but now view the p nodes as forming a $r \times c$ mesh.

Algorithm 3

1. Partition A , x , and y :

$$A \rightarrow \begin{pmatrix} A_{00} & A_{01} & \dots & A_{0,p-1} \\ A_{10} & A_{11} & \dots & A_{1,p-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p-1,0} & A_{p-1,1} & \dots & A_{p-1,p-1} \end{pmatrix}, \quad x \rightarrow \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{p-1} \end{pmatrix},$$

$$\text{and } y \rightarrow \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{p-1} \end{pmatrix},$$

where A_{ij} has n/p rows and columns, and x_i and y_i are of size n/p .

2. Assume that initially A_{ij} , x_i , and y_i are assigned to nodes as illustrated in Fig. 4. Each node has $p/r \times p/c = p$ submatrices and one subvector of size n/p .
3. Allgather x_j 's within columns of nodes.

y_0	A_{00}	A_{01}	A_{02}	y_1	A_{03}	A_{04}	A_{05}	y_2	A_{06}	A_{07}	A_{08}
	A_{10}	A_{11}	A_{12}		A_{13}	A_{14}	A_{15}		A_{16}	A_{17}	A_{18}
	A_{20}	A_{21}	A_{22}		A_{23}	A_{24}	A_{25}		A_{26}	A_{27}	A_{28}
x_0				x_3				x_6			
y_3	A_{30}	A_{31}	A_{32}	y_4	A_{43}	A_{44}	A_{45}	y_5	A_{36}	A_{37}	A_{38}
	A_{40}	A_{41}	A_{42}		A_{53}	A_{54}	A_{55}		A_{46}	A_{47}	A_{48}
	A_{50}	A_{51}	A_{52}		A_{63}	A_{64}	A_{65}		A_{56}	A_{57}	A_{58}
x_1				x_4				x_7			
y_8	A_{60}	A_{61}	A_{62}	y_9	A_{63}	A_{64}	A_{65}	y_8	A_{66}	A_{67}	A_{68}
	A_{70}	A_{71}	A_{72}		A_{73}	A_{74}	A_{75}		A_{76}	A_{77}	A_{78}
	A_{80}	A_{81}	A_{82}		A_{83}	A_{84}	A_{85}		A_{86}	A_{87}	A_{88}
x_2				x_5				x_8			

Collective Communication. Fig. 4 Initial distribution of submatrices and vectors. Here $p = 9$, $r \times c = 3 \times 3$, and the boxes represent nodes

$y_0^{(0)}$	A_{00}	A_{01}	A_{02}	$y_0^{(1)}$	A_{00}	A_{01}	A_{02}	$y_0^{(2)}$	A_{06}	A_{07}	A_{08}
$y_1^{(0)}$	A_{10}	A_{11}	A_{12}	$y_1^{(1)}$	A_{13}	A_{14}	A_{15}	$y_1^{(2)}$	A_{16}	A_{17}	A_{18}
$y_2^{(0)}$	A_{20}	A_{21}	A_{22}	$y_2^{(1)}$	A_{23}	A_{24}	A_{25}	$y_2^{(2)}$	A_{26}	A_{27}	A_{28}
x_0	x_1	x_2		x_3	x_4	x_5		x_6	x_7	x_8	
$y_3^{(0)}$	A_{30}	A_{31}	A_{32}	$y_3^{(1)}$	A_{33}	A_{34}	A_{35}	$y_3^{(2)}$	A_{36}	A_{37}	A_{38}
$y_4^{(0)}$	A_{40}	A_{41}	A_{42}	$y_4^{(1)}$	A_{43}	A_{44}	A_{45}	$y_4^{(2)}$	A_{46}	A_{47}	A_{48}
$y_5^{(0)}$	A_{50}	A_{51}	A_{52}	$y_5^{(1)}$	A_{53}	A_{54}	A_{55}	$y_5^{(2)}$	A_{56}	A_{57}	A_{58}
x_0	x_1	x_2		x_3	x_4	x_5		x_6	x_7	x_8	
$y_6^{(0)}$	A_{60}	A_{61}	A_{62}	$y_6^{(1)}$	A_{63}	A_{64}	A_{65}	$y_6^{(2)}$	A_{66}	A_{67}	A_{68}
$y_7^{(0)}$	A_{70}	A_{71}	A_{72}	$y_7^{(1)}$	A_{73}	A_{74}	A_{75}	$y_7^{(2)}$	A_{76}	A_{77}	A_{78}
$y_8^{(0)}$	A_{80}	A_{81}	A_{82}	$y_8^{(1)}$	A_{83}	A_{84}	A_{85}	$y_8^{(2)}$	A_{86}	A_{87}	A_{88}
x_0	x_1	x_2		x_3	x_4	x_5		x_6	x_7	x_8	

Collective Communication. Fig. 5 After allgather of subvectors of x within columns and local matrix-vector multiplication

4. In parallel, compute the local part of each y_i on the appropriate node:

$$y_i^{(k)} = \sum_{\text{local } j} A_{ij}x_j$$

as illustrated in Fig. 5.

5. Reduce-scatter within rows of nodes to compute $y_i = \sum y_i^{(k)}$.

Since the extra local memory and time required for the gathered and reduced subvectors is only $O(cn/p)$ and $O(rn/p)$, respectively, this algorithm can be shown to be essentially scalable by keeping local memory use on each node constant and keeping the ratio of r and c constant. Notice that Algorithm 1 results by taking $c = 1$ and Algorithm 2 by taking $r = 1$.

The example illustrates that often collective communications must be performed within subsets (e.g., a row

or column of a mesh) of nodes. For parallel interfaces that do not support formation of subsets of nodes, the algorithm is difficult to express.

Interfaces

Message-Passing Interface (MPI)

A widely used parallel interface with an explicit, rich set of collective communication operations is the *Message-Passing Interface* (MPI) [14]. The MPI collectives are widely used in applications. MPI includes all collective communication operations of Figs. 1–3, except for permutation, as well as a barrier synchronization collective. Collective communication is effected by library calls, and for each operation, all nodes (in a set of nodes) eventually have to make the same call, also for the *rooted* operations for which an explicit root argument (that need to be given by all nodes involved

in the call) designates the root node. All collectives in MPI are *blocking* but *non-synchronizing*, except for `MPI_Barrier`, meaning that a node (*process* in MPI terminology) is allowed to continue as soon as it has contributed and received its required data. Collectives like `MPI_Allreduce` and `MPI_Allgather` are of course non-synchronizing only in the trivial case where all processes contribute data of size zero. The redistribution collectives, except for `MPI_Bcast`, come in both *regular* (e.g., `MPI_Allgather`) and *irregular* variants (e.g., `MPI_Allgatherv`), the latter termed *vector* variants and designated by the “v” suffix. For the regular collective operations, the size of the data is given at the call of the operation, such that nodes know the size of the data involved in the operations. For the irregular operations, each node only specifies the size of the data to be contributed and received by itself. This means that in irregular all-to-all communication, no node can from the outset know what communication will be entailed between other nodes, which makes it impossible to compute efficient communication schedules without additional communication. MPI incorporates general, powerful, and convenient mechanisms for collective communication with *nonhomogeneous* data, which again increases algorithmic and software complexity of MPI library implementations.

The MPI reduction operations correspond exactly to those listed in Fig. 3 and are all *regular*, except for `MPI_Reduce_scatter` which allows blocks (sub-vectors) of different sizes to be distributed across the nodes. A regular variant also exists. For the reduction operations, a set of standard, arithmetical and logical binary operators is predefined and it is furthermore possible for the application to introduce its own user-defined operators; these are required to be associative. MPI provides vague quality guarantees on the reductions, mostly relevant for reductions with floating point numbers, for instance, that the same result is computed on all nodes in an `MPI_Allreduce`, that the result is independent of the physical placement of the nodes in the system, or that all elements of result vectors are computed in the same order and so forth. Such requirements sometimes complicate the practical design and implementation of reduction collectives.

Finally, MPI has powerful mechanisms for forming named subsets of nodes (namely, process groups

and communicators), and collective communication can take place on any such subset of nodes.

Partitioned Global Address Space (PGAS) Languages

Partitioned Global Address Space (PGAS) languages strive to hide the details of communication from users to improve programmability and/or reduce overhead. PGAS languages and implementations nevertheless benefit from explicit collective communication, and such languages provide collective communication as either integral parts of the language as in X10 [17], or standard libraries as in Unified Parallel C (UPC) [5]. The standard UPC library provides the operations listed in Figs. 1 and 2, including a permutation collective (absent in MPI), in *regular* variants, and restricted to mostly *homogeneous* data. Placement of input and result vectors is mostly determined implicitly by the so-called affinity of the data elements. In particular the root node is the node to which the input (for `upc_all_broadcast`) or result (for `upc_all_gather`) has affinity. Blocking and synchronization properties can be flexibly controlled by a mode given with each collective call. Unlike MPI, the reduction operations `upc_all_reduce` and `upc_all_prefix_reduce` are operations on row vectors that produce a single result or prefix sum for each node. At the time of this writing UPC collectives are over the full set of nodes, i.e., UPC does not support the formation of node subsets over which collective communication can be performed.

Coarray Fortran 2.0 (CAF 2.0) addresses some of these issues and provides a full and very general set of collective operations that can be applied over arbitrary teams of nodes [12].

Bulk Synchronous Processing Libraries

Bulk synchronous or coarse-grained processing is a parallel algorithmic design approach that structures applications into phases of local computation on local data interleaved with phases of global exchange or routing. This approach naturally relies on collective communication as explicated by the exchange phases, but can additionally benefit from explicit collective communication for realizing special communication patterns. This is based on the observation that applications often entail specific communication patterns that are

both conveniently expressed as collective communication and much more efficiently implemented by specialized algorithms than by relying on the general routing communication of the next routing phase. Most practical realizations of the Bulk Synchronous Processing (BSP) model included explicit collective communication operations [2, 8, 11] as explained here, typically in a nonblocking form. Some BSP libraries [2] supported formation of subsets of nodes.

Numerical Libraries

Numerical libraries that explicitly have or expose collective communication operations include ScaLAPACK and PLAPACK. See the entries for these libraries.

Other Collective Interfaces and Frameworks

Other, more specialized collective communication operations than those described in this entry are sometimes included in higher-level interfaces and frameworks, or in application-specific libraries. For instance, libraries like PETSc and Trillinos have collective computation (reduction) operations, and I/O libraries like pnetCDF and HDF5 have collective data movement operations. As an example, algorithmic libraries may include merging and sorting primitives and parallel operations on data structures in their repertoire. The skeletal approach to parallel programming [15] expresses applications in terms of instances of higher-level generic patterns, many of which entail collective communication. Such approaches provide considerable expressive flexibility and often a very rich set of collective communication and computation patterns. A concrete example in this direction is the MapReduce framework [4].

Algorithms

Algorithmic and further details on some of the most important of the operations described above are given in related entries. Basic techniques for collective communication implementation are found under the Broadcast entry, which should be read first. The symmetric operations allgather and all-to-all are treated in separate entries, and discuss techniques not used by Broadcast. The Reduce to root operation can often be treated as the inverse of Broadcast (with additional operation). The fundamental all prefix sums collective

is treated more extensively in a separate entry, and completes the tour of basic techniques for the implementation of collective communication on distributed memory systems.

Related Entries

- ▶ [Allgather](#)
- ▶ [All-to-all](#)
- ▶ [BSP \(Bulk Synchronous Parallelism\)](#)
- ▶ [Collective Communication](#)
- ▶ [MPI \(Message Passing Interface\)](#)
- ▶ [PETSc \(Portable, Extensible Toolkit for Scientific Computation\)](#)
- ▶ [PGAS \(Partitioned Global Address Space\) Languages](#)
- ▶ [PLAPACK](#)
- ▶ [ScaLAPACK](#)
- ▶ [Scan for Distributed Memory, Message-Passing Systems](#)

Bibliographic Notes and Further Reading

The benefits of encapsulating certain collective communication patterns as specific collective operations in languages, interfaces and libraries have been realized from the early days of parallel computing, especially in scientific computing [6]. For distributed memory systems explicit collective communication is a natural extension to the message-passing paradigm. Convergence between many different message-passing languages and libraries in the early 1990s led to definition of a common *Message-Passing Interface* [10], well noted for the important role given to collective communication. The advantages of and need for interfaces and libraries with fast and efficient collective communication to better exploit the underlying, concrete communication system was realized and argued in a number of papers, e.g., [1, 13]. The study of algorithms for efficient collective communication likewise dates back to the early days of distributed memory parallel computing [6, 16]. Another early approach to systematic collective communication algorithm design can be found in [9]. A comprehensive discussion of practical implementations for most of the collective communications discussed here can be found in [3]. There has been a huge amount of research and development work on efficient realization of these collectives for a large variety of (often extremely high-performance) systems. It is a major goal

for MPI implementers to provide well-performing collective communication for the intended target systems. In [18] semantic relationships between different collective operations were used to urge implementers to provide consistently well-performing collective operations in order to enhance performance portability of applications written in MPI.

Bibliography

1. Bala V, Bruck J, Cypher R, Elustondo P, Ho A, Ho C-T, Kipnis S, Snir M (1995) CCL: A portable and tunable collective communications library for scalable parallel computers. *IEEE Trans Parallel Distrib Syst* 6(2):154–164
2. Bonorden O, Juurlink BHH, von Otte I, Rieping I (2003) The Paderborn University BSP (PUB) library. *Parallel Comput* 29(2):187–207
3. Chan E, Heimlich M, Purkayastha A, van de Geijn RA (2007) Collective communication: theory, practice, and experience. *Concurr Comput Pract Exp* 19(13):1749–1783
4. Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. *Commun ACM* 51(1):107–113
5. El-Ghazawi T, Carlson W, Sterling T, Yellick K (2005) UPC: distributed shared memory programming. Wiley, Hoboken
6. Fox G, Johnson M, Lyzenga G, Otto S, Salmon J, Walker D (1988) Solving problems on concurrent processors, vol 1. Prentice-Hall, Englewood Cliffs
7. Gorlatch S (2004) Send-receive considered harmful: Myths and realities of message passing. *ACM Trans Program Lang Syst* 26(1):47–56
8. Goudreau M, Lang K, Rao SB, Suel T, Tsantilas T (1999) Portable and efficient parallel computing using the BSP model. *IEEE Trans Comput* 48(7):670–689
9. Hambrusch SE, Hameed F, Khokhar AA (1995) Communication operations on coarse-grained mesh architectures. *Parallel Comput* 21(5):731–752
10. Hempel R, Hey AJG, McBryan O, Walker DW (1994) Special issue: message passing interfaces. *Parallel Comput* 20(4):415–678
11. Hill JMD, McColl B, Stefanescu DC, Goudreau MW, Lang K, Rao SB, Suel T, Tsantilas T, Bisseling RH (1998) BSPlib: The BSP programming library. *Parallel Comput* 24(14):1947–1980
12. Mellor-Crummey J, Adhianto I, Scherer III WN, Jin G (2009) A new vision for Coarray Fortran. In: Third conference on Partitioned Global Address Space Programming Models, Asburn, VA
13. Mitra P, Payne DG, Schuler L, van de Geijn R (1995) Fast collective communication libraries, please. In: Proceedings of the Intel supercomputer users' group meeting
14. MPI Forum. MPI: A message-passing interface standard, version 2.2. 4 Sept 2009. www.mpi-forum.org
15. Rabhi FA, Gorlatch S (eds) (2003) Patterns and skeletons for parallel and distributed computing. Springer-Verlag, London
16. Saad Y, Schultz MH (1989) Data communication in parallel architectures. *Parallel Comput* 11(2):131–150
17. Saraswat VA, Sarkar V, von Praun C (2007) X10: Concurrent programming for modern architectures. In: ACM SIGPLAN symposium on principles and practice of parallel programming (PPoPP), San Jose, p 271
18. Träff JL, Gropp WD, Thakur R (2010) Selfconsistent MPI performance guidelines. *IEEE Trans Parallel Distrib Syst* 21: 698–709

Collective Communication, Network Support For

DHABALESWAR K. PANDA, SAYANTAN SUR, HARI SUBRAMONI, KRISHNA KANDALLA
The Ohio State University, Columbus, OH, USA

Synonyms

[Inter-process Communication](#); [Network Architecture](#); [Network Offload](#)

Definition

Collective Communication involves more than one processor participating in one communication operation. Collective communication operations are: broadcast, barrier synchronization, reduction, gather, scatter, all-to-all complete exchange, and scan. Network support for collective communication provides architectural support for efficient and scalable collective operations with low processor overhead.

Discussion

Introduction

Collective communication operations aim at reducing both latency and network traffic with respect to the case where the same operations are implemented with a sequence of unicast messages. The significance of collective communication operations for scalable parallel systems has been emphasized by their inclusion in widely used parallel programming models, such as the Message Passing Interface (MPI) [27]. A large number of parallel applications depend on the performance benefits provided by collective operations. MPI libraries strive to provide the best possible performance for collective operations, since they are the key for good end

application performance. Consequently, a large number of algorithms and advanced network support for collective communication have been proposed. The commonly used collective communication operations are: broadcast, barrier synchronization, reduction, gather, scatter, all-to-all complete exchange, and scan.

Broadcast is a very common operation needed in a parallel system to distribute code and data from a host process to a set of processes before computation begins. This is especially useful for SPMD style programming. During the computational phase of distributed memory applications, this operation is used to distribute data among other processes. In the literature, it is also known as a *one-to-all* operation with non-personalized data movement. If multiple broadcasts happen at the same time from different processes, it is called as a *many-to-all* operation. If the data is distributed to only a subset of processes, the operation is called a *multicast*. In MPI, this operation is called MPI_Bcast.

Barrier synchronization is a common operation in parallel systems to synchronize execution of multiple processes. Primarily, it is used to support a producer-consumer relationship of shared data. Functionally, this operation involves *many-to-one* communication followed by a *one-to-many* communication operation. In MPI, this operation is called MPI_BARRIER.

Reduction operations are used widely in distributed memory systems and consist of computing a single value out of data sent by the different participating processes. Commonly used reduction operations are max, min, sum, or any user-defined operations. The MPI specification includes two forms of reduction MPI_Reduce and MPI_Allreduce. In MPI_Reduce, the result is available only on the root process, and in MPI_Allreduce, the result is available on all participating processes.

Gather operation is defined as gathering of data from a set of member processes by a member. This involves gathering personalized data (different data from different processes). In MPI, it is called MPI_Gather. *Scatter* is the reverse operation of Gather. MPI also provides MPI_Allgather that performs an all-to-all broadcast.

All-to-all complete exchange combines scatter and gather. In this operation, every member has some personalized data to send to every other group member. A sequence of interleaved gather and scatter steps

are used in this operation to achieve the desired data movement. This operation is used in distributed memory systems to *redistribute* data.

Scan operation involves reduction of data with ranks. Sometimes, this operation is also referred to as *parallel prefix* computation. Unlike regular reduction, this operation is carried out in such a way that a member of rank i only receives reduction results data associated with members of ranks 0 to i . This operation is used widely in image processing and visualization applications.

Architectural Support for Collective Communication

The importance of collective operations in parallel computing is underscored by the increasing network and architectural support for improving collective operation performance. There are three major types of network support for collective operations: (a) adapter support, (b) switch support, and (c) dedicated networks for collectives. The network support aims to improve latency and improve bandwidth. Another important performance metric for collective operations is tolerance to system noise. As the number of processes involved in a collective operation increases, short delays at individual processes start to accumulate over the various stages of the collective and adversely affect the overall time taken for the collective. This is called as *system noise*. Tolerance to noise means that the collective operations should be able to proceed regardless of other tasks running on the processor, a technique called *application-bypass*. These techniques can be enabled through intelligent network adapters, switches or by having dedicated collective communication networks. In addition to these, MPI libraries have advanced software support for bridging the gap between low-level architectural support and higher-level user collective operations.

Adapter-Based Support

Compute nodes are attached to the network via network adapters. Adapters are the closest to the end processors, and therefore can be used to perform a wide variety of network offload mechanisms. Intelligent adapters with processors can be used to perform complex operations on incoming network data. They can also be used to offload a series of operations that the adapter

then continues to perform, thus freeing up the host processor to return to computation. The following are some of the adapter-based network support that have been employed by leading high-performance network vendors over the past few decades.

Myrinet

Myrinet is a high-performance full-duplex network which uses NICs with programmable processors. Myrinet provides a user-level message passing system (GM) which uses the programmable NICs for much of the protocol processing. GM consists of three components: a kernel module, a user-level library, and a control program (MCP) which runs on the NIC processor. The driver loads the MCP on to the NIC when it is loaded. It is possible to modify the MCP and cross-compile for the network processor. Researchers have modified the control program to support advanced collective operations. The modified MCP can be accessed through new interfaces by the MPI library. Finally, collective operations of the end application can be offloaded to the NIC. In effect, the MCP (through multiple finite state machines) will tell the NIC the times at which it should initiate the various send and receive operations which form the basis of the collective operation. Myrinet also provides useful primitives such as *multi-send* which allows the user to offload a series of send operations to the NIC. Such primitives are extremely useful to design collective algorithms such as *barrier* and *broadcast*. In tree-based collective algorithms, although operations like *multi-send* reduce the overhead of the broadcast operations at the root processes, the intermediate nodes would still require involvement from the host processor to progress the collective operation. Researchers have devised *application-bypass* techniques through a modified MCP that enable intermediate network adapters in the collective tree to forward messages from the NIC itself, thereby mitigating effects of system noise.

Quadrics

The Quadrics interconnect had very advanced network adapter support. The Quadrics network access library is called *Elan*. Elan exposes a protected, programmable network interface to end users through vendor-provided programming libraries (QSNet). These libraries are built in a layered fashion with higher-level

programming libraries (*Elanlib*) built on top of lower level point-to-point communication primitives. Elanlib provides two barrier functions, *elan_gsync()* and *elan_hgsync()*. The latter takes advantage of the hardware broadcast primitive and provides a very efficient and scalable barrier operation. However, it requires that the calling processes are well synchronized in their stages of computation. Otherwise, it falls back on the *elan_gsync()* to complete the barrier with a tree-based gather-broadcast algorithm.

InfiniBand RDMA Enabled Collectives

Remote Direct Memory Access (RDMA) is a technique using which a process can directly access the memory locations of some other process, without participation of the remote process. InfiniBand is one of the networks that provides high-performance RDMA. Using the RDMA primitives, efficient collective operations can be designed. The main benefits of RDMA collectives over regular point-to-point collectives are: (a) communication calls can bypass several intermediate software layers, (b) reduce number of message copies to and from bounce buffers (bounce buffers can be directly accessed by InfiniBand), (c) reduce protocol handshakes, such as *rendezvous* for large MPI messages, and (d) reduce the total number of registration operations, i.e., perform one larger memory registration, instead of several smaller ones.

These mechanisms can further be enhanced for shared memory processor (SMP)-based systems by performing the collective operation in two steps. In the first step, all processes local to a node write their data to a common shared buffer. This is then exchanged over the network to remote nodes by designated processes called *leaders*. The third step involves the leader process distributing the data to the local processes through shared memory.

InfiniBand Collective Offload

Collective operations defined in the current MPI-2 standard are blocking operations. This implies that the applications need to wait until collective operations complete before doing any other compute tasks. Since collective operations involve many processes, their latency is bound to increase as systems scale. Several researchers have also observed that system noise can potentially affect the latency of blocking collective



operations, and hence the performance of end applications. It is widely believed that hiding the latency of collective operations can strongly benefit the performance of parallel applications. InfiniBand vendors, such as Mellanox, have recently introduced new network adapters, such as the ConnectX-2 that offer network-offload features. Host processors can create arbitrary task-lists comprising of send and receive operations and post them to the work request queue of the network interface. The network interface can independently execute these task lists without further intervention from the host processors. This *application-bypass* technique has the potential advantage of minimizing the effects of system noise along with offering more CPU cycles to applications for performing compute tasks.

MPI libraries are designed in a highly efficient, scalable manner to leverage the full functionality of these network interfaces. Non-blocking collective operations designed in this manner can potentially allow applications to scale better by countering the effects of system noise and allowing applications to hide the time required for collective operations by overlapping them with compute tasks. A new revision to the MPI standard, MPI-3, is likely to standardize non-blocking collective communication.

Switch-Based Support

Network switches form the backbone of the interconnect. High-performance networks have point-to-point links to maximize performance. Switches are responsible for routing packets from source nodes to destination nodes. Since switches are in a central position, it is a very attractive point to introduce network support for collective communication. The following architectural supports are available in Quadrics and InfiniBand switches for collective communication operations.

Quadrics

The Quadrics network features hardware support for collective communication. Each process can map a portion of the address space into global memory. These addresses constitute virtual shared memory. The switch provides hardware-based broadcast. This broadcast is reliable, i.e., no software mechanism is required to ensure that broadcast packets reach their destinations.

The broadcast mechanism replicates packets to particular switch output ports, with the capability of broadcasting to a subset of processors. Using the broadcast mechanism, both broadcast and barrier collectives can be implemented.

InfiniBand Multicast

Modern networks such as InfiniBand provide hardware-based *multicast* operation. The multicast operation gives the ability for a process to send a single message to a specific subset of processes which may be on different end nodes. Such a hardware-based multicast capability provides the following benefits: (a) only one send operation is needed to initiate the multicast, greatly reducing the host overhead at the sender, and (b) packets can be duplicated at the switch ports, reducing network traffic.

InfiniBand multicast does not guarantee reliable delivery. It is based on Unreliable Datagram (UD) transport. Software-based solutions to ensure reliability need to be provided. MPI library designers must take this into account. As with the RDMA-based approach, this scheme can also be enhanced for current generation of SMP systems. Such studies have already been done by researchers for IB systems and they show considerable improvement in performance as opposed to a non-SMP-aware approach.

InfiniBand Fabric Channel Accelerator

The Fabric Channel Accelerator technology from Voltaire (now merged with Mellanox) introduced this feature in 2009. The aim of this feature is to enable high-level optimizations in InfiniBand switches by employing a general purpose CPU on the switch itself. The CPU on the switch can then manage communications to different nodes. The individual nodes can connect to the CPU on the switch over Unreliable Datagram (UD) transport. The FCA library and algorithms then implement reliability on top of the unreliable transport. MPI libraries can talk with the FCA library to implement collective operations. Several commonly used collectives, such as barrier, broadcast, and reduce are supported.

Support Through Dedicated Networks

The most advanced support for collective operations is by dedicating entire networks for efficient implementation of collective operations. The communication

requirements of point-to-point and collective operations are often very different. Collective operations are very sensitive to latency and system noise. The following section describes the dedicated architectural support provided in IBM BlueGene system to accelerate collectives.

BlueGene

The IBM BlueGene system featured one of the most advanced support for scalable collective operations. The compute nodes were connected through five networks: a 3-D torus network for point-to-point messaging, a global collective network, a global barrier and interrupt network, and two other Ethernet networks for control and I/O. The global collective network is useful for speeding up commonly used MPI collective communications constructs. And the global barrier network quickly synchronizes state across all processes in the system. The BlueGene/L system had collective network with bandwidth up to 350 MBps with 1.5 μ s latency. The later BlueGene/P system featured collective network bandwidth of 850MBps and 1.3 μ s per tree traversal. The global barrier and interrupt network has a hardware latency of 0.65 μ s.

In addition to the hardware support, the IBM MPI libraries provide many advanced collective communication algorithms that map the hardware resources to user collective operations.

Software Optimizations

In addition to network support, software optimizations also play an important role in the performance of collective operations on modern platforms. There are two major ways by which software optimizations may be employed for collective operations: (a) algorithm design based on network topology, and (b) algorithm design in multi-core and many-core systems.

Large supercomputer systems must scale with increasing node count. One of the major problems with scaling is the number of links required to interconnect the system. An all-to-all connectivity *crossbar* connection requires N^2 links, which is clearly not scalable. In response, system designers have employed 2-D, 3-D meshes, k-dimensional Tori, and other interconnection topologies. The IBM BlueGene and Cray systems have used 3-D torus for their large-scale systems. Algorithms that implement collective operations must be designed

with these topologies in mind. The IBM BlueGene and Cray feature several such algorithmic optimizations.

Modern compute servers are based on multi/many-core architectures and offer high compute densities. Interestingly, these architectures have also introduced a new design space for collective algorithms. Since many processes share the same address space within a compute node, they can communicate directly through memory with smaller latency. Suppose we consider a MPI_Bcast operation. One or more processes within a node can be chosen as the *leaders* and they communicate over the network to receive the data from the root. These leaders then broadcast the data to the other processes that are within the same compute node, directly through memory. Researchers have also extended this concept to consider the network topology to further improve the latency of collective operations. These algorithms are particularly useful on large-scale systems, where compute nodes are organized across multiple racks and inter-rack communication are costlier.

Collectives Today

Owing to their ease of use and portability, collective communication operations are used extensively in scientific parallel applications. Hence, it is critical to improve the performance and scalability of collective operations. Most of the current research in this field is focused on two important aspects: (a) improving the latency of collective operations through enhanced software designs that leverage the advanced multi-core architectures and network features and (b) designing non-blocking collective operations that allow applications to achieve communication/computation overlap.

Many MPI libraries such as MVAPICH2 [23], OpenMPI [8], Intel-MPI use aggressive shared-memory-based designs to optimize latency of blocking collective operations on modern multi-core architectures. However, the performance of collective operations also strongly depends on the topology of interconnection networks. Supercomputing systems use complex network architectures ranging from fat-trees to 3-D Torus and hypercubes. Researchers are exploring alternatives to detect the system topology and design collective algorithms in a topology-aware manner to achieve lower latency.

As parallel applications are scaled out to tens of thousands of processes, it becomes necessary to hide the costs associated with collective operations. Non-blocking collective operations are widely believed to address this requirement. The goal of such an interface is to allow applications to perform compute tasks, while the collective operations are in progress. The biggest challenge of designing collective operations in this manner is that they need to be progressed, with little intervention from the host processors. Researchers are exploring ways to offload collective operations to modern network interfaces so that the host processors can be directly used to perform application-level compute tasks.

Future Directions

Given the current growth in multiprocessor chip designs, next-generation compute servers are expected to offer hundreds to thousands of compute cores. Next-generation networks are expected to offer a rich set of advanced features, apart from guaranteeing lower communication overheads. Designing algorithms to minimize the communication latency of blocking collective operations will continue to be a critical aspect of any communication library. At the same time, designing an efficient set of non-blocking collective operations that allow parallel applications to hide the communication latency promises to improve application performance. Communication libraries offering a high performance, scalable suite of collective operations can significantly improve the performance of scientific applications on next-generation systems. The upcoming MPI-3 standard will include non-blocking communication operations. In order to leverage the latency hiding properties of non-blocking collectives, end parallel applications will need to be redesigned.

Related Entries

- [Clusters](#)
- [Collective Communication](#)
- [Interconnection Networks](#)
- [Routing \(Including Deadlock Avoidance\)](#)

Bibliographic Notes and Further Reading

Optimizing collective operations across various networks and systems has been an area of active research

for several years. Researchers have investigated optimizing collective communication on the IBM BlueGene supercomputers in [1, 2, 5, 6, 21]. Designs that leveraged various features offered by Quadrics and Myrinet networks to improve the performance of collective operations were proposed in [3, 4, 12, 13, 34–37]. Researchers have explored the challenges associated with off-loading collective operations to network adapters in [9, 10, 31, 32]. Software-based optimizations that consider the network topology and multi-core architectures to improve collective operations were proposed by various researchers in [11, 18–20, 24–26, 28, 33]. The impact of system noise on the performance of collective operations and real-world applications were studied in [7, 14, 15, 29, 30]. Recently, researchers have proposed the need for an efficient non-blocking interface for collective operations in [16, 17].

Bibliography

1. Almasi G, Dozsa G, Chris Erway C, Steinmacher-Burow B (2005) Efficient implementation of Allreduce on BlueGene/L collective network. In: Recent advances in parallel virtual machine and message passing interface. Lecture notes in computer science, vol 3666. Springer, Berlin/Heidelberg, pp 57–66
2. Almasi G, Heidelberger P, Archer CJ, Martorell X, Chris Erway C, Moreira JE, Steinmacher-Burow B, Zheng Y (2005) Optimization of mpi collective communication on bluegene/l systems. In: Proceedings of the 19th annual international conference on Supercomputing, ICS '05, Cambridge, pp 253–262
3. Buntinas D, Panda DK, Duato J, Sadayappan P (2000) Broadcast/Multicast over Myrinet using NIC-Assisted multideestination messages. In: Proceedings of the 4th international workshop on network-based parallel computing: communication, architecture, and applications. Springer, London, pp 115–129
4. Buntinas D, Panda DK, Sadayappan P (2001) Performance benefits of NICBased barrier on Myrinet/GM. In: IPDPS, San Francisco, p 166
5. Faraj A, Kumar S, Smith B, Mamidala A, Gunnels J, Heidelberger P (2009) Mpi collective communications on the blue gene/p supercomputer: algorithms and optimizations. In: Proceedings of the 23rd international conference on supercomputing, ICS '09, Yorktown Heights, pp 489–490
6. Faraj A, Kumar S, Smith B, Mamidala A, Gunnels J (2009) MPI collective communications on the blue Gene/P supercomputer: algorithms and optimizations. In: Symposium on high-performance interconnects, New York, pp 63–72
7. Faraj A, Patarasuk P, Yuan X (2007) A study of process arrival patterns for MPI collective operations. In: Proceedings of the 21st annual international conference on supercomputing, Seattle, pp 168–179
8. Garbriel E, Fagg GE, Bosilica G, Angskun T, Dongarra JJ, Squyres JM, Sahay V, Kambadur P, Barrett B, Lumsdaine A, Castain RH,

- Daniel DJ, Graham RL, Woodall TS (2004) Open MPI: goals, concept, and design of a next generation MPI implementation. In: Proceedings of the 11th European PVM/MPI Users' group meeting, Budapest
9. Graham R, Poole S, Shamis P, Bloch G, Boch N, Chapman H, Kagan M, Shahar A, Rabinovitz I, Shainer G (2010) Overlapping computation and communication: Barrier algorithms and connectX-2 CORE-direct capabilities. In: Proceedings of the 22nd IEEE international parallel & distributed processing symposium, workshop on communication architectures for clusters (CAC)'10, Atlanta, GA, USA
 10. Graham R, Poole S, Shamis P, Bloch G, Boch N, Chapman H, Kagan M, Shahar A, Rabinovitz I, Shainer G (2010) ConnectX2 infiniband management queues: new support for network Ofaded collective operations. In: CCGrid'10, Melbourne, 17–20 May 2010
 11. Graham R, Shipman G (2008) MPI support for multi-core architectures: optimized shared memory collectives. In: Recent advances in parallel virtual machine and message passing interface, lecture notes in computer science, vol 5205/2008, Sorrento, pp 130–140
 12. Gunawan TS, Cai W (2003) Performance analysis of a Myrinet-based cluster, vol 6. Springer, Netherlands, pp 299–313
 13. Hoefer T, Mosch M, Mehlan T, Rehm W (2007) CollGM – A Myrinet/GM optimized collective component for Open MPI. In: Proceedings of 3rd KiCC Workshop 2007, RWTH Aachen
 14. Hoefer T, Schneider T, Lumsdaine A (2009) The impact of network noise at large-scale communication performance. In: Proceedings of IEEE International Symposium on parallel and distributed processing, Rome, pp 1–8
 15. Hoefer T, Schneider T, Lumsdaine A (2010) Characterizing the influence of system noise on large-scale applications by simulation. In: Proceedings of the 23rd annual international conference on Supercomputing, Greece
 16. Hoefer T, Squyres J, Bosilca G, Fagg G, Lumsdaine A, Rehm W (2006) Non-blocking collective operations for MPI-2. Technical report, Open Systems Lab, Indiana University
 17. Hoefer T, Squyres JM, Rehm W, Lumsdaine A (2006) A case for nonblocking collective operations. In: Frontiers of high performance computing and networking, ISPA 2006 workshops. Lecture notes in computer science, vol 4331/2006. Sorrento, Italy, pp 155–164
 18. Kandalla K, Subramoni H, Panda DK (2010) Designing topology-aware collective communication algorithms for large scale infini-band clusters: case studies wih Scatter and Gather. In: Workshop on Communication Architecture for Clusters, (CAC'10), Austin
 19. Kandalla K, Subramoni H, Santhanaraman G, Koop M, Panda DK (2009) Designing multi-leader-based Allgather algorithms for multi-core clusters. In: Proceedings of the 2009 IEEE international symposium on parallel and distributed processing, IEEE computer society, Washington, DC, pp 1–8
 20. Karonis NT, de Supinski BR, Foster I, Gropp W, Lusk E, Bresnahan J (2000) Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In: Proceedings of the 14th international symposium on parallel and distributed processing, Cancun, Mexico, p 377
 21. Kumar S, Sabharwal Y, Garg R, Heidelberger P (2008) Optimization of all-to-all communication on the Blue Gene/L Supercomputer . In: Proceedings of the 2008 37th international conference on parallel processing, Portland, pp 320–329
 22. Lawrence J, Yuan X (2008) An MPI tool for automatically discovering the switch level topologies of ethernet clusters. In: IPDPS workshop on system management techniques, processes, and services, Miami
 23. Liu J, Jiang W, Wyckoff P, Panda DK, Ashton D, Buntinas D, Gropp W, Toonen B (2004) Design and implementation of MPICH2 over infiniband with RDMA support. In: Proceedings of international parallel and distributed processing symposium (IPDPS '04), Santa Fe
 24. Mamidala A, Kumar R, Panda DK (2008) MPI collectives on modern multicore clusters: performance optimizations and communication characteristics. In: CCGrid, Lyon
 25. Mamidala AR, Chai L, Jin HW, Panda DK (2008) Efficient SMP-aware MPI-level broadcast over InfiniBand's hardware multicast. In: IEEE international symposium on parallel and distributed processing, IPDPS 2008, Greece, p 305
 26. Mamidala AR, Vishnu A, Panda DK (2006) Efficient shared memory and RDMA based design for MPI-allgather over infiniBand. In: 13th European PVM/MPI user's group meeting, vol 4192. Bonn, 17–20 Sept 2006
 27. Message Passing Interface Forum (1994) MPI: a message-passing interface standard, Mar 1994
 28. Patarasuk P, Yuan X (2007) Bandwidth efficient all-reduce operation on tree topologies. In: HIPS, Long Beach
 29. Patarasuk P, Yuan X (2008) Efficient MPI Bcast across different process arrival patterns. In: Proceedings of international parallel and distributed processing symposium (IPDPS), Miami, pp 1–11
 30. Petrini F, Kerbyson DJ, Pakin S (2003) The case of the missing supercomputer performance: achieving optimal performance on the 8,192 processors of ASCI Q. In: Proceedings of the 2003 ACM/IEEE conference on supercomputing, Washington, DC p 55
 31. Scott Hemmert K, Barrett BW, Underwood KD (2010) Using triggered operations to offload collective communication operations. In: Proceedings of the 17th european MPI users' group meeting conference on recent advances in the message passing interface, EuroMPI'10, Springer, Berlin/Heidelberg, pp 249–256
 32. Subramoni H, Kandalla K, Sur S, Panda DK (2010) Design and evaluation of generalized collective communication primitives with overlap using connectX-2 offload engine. In: The 18th annual symposium on high performance interconnects, HotI 2010, Santa Clara
 33. Thakur R, Gropp W (2003) Improving the performance of collective operations in MPICH. In: Recent advances in parallel virtual machine and message passing interface. Lecture notes in Computer Science, vol 2840/2003. Venice, Italy, pp 257–267
 34. Tipparaju V, Nieplocha J (2005) Optimizing all-to-all collective communication by exploiting concurrency in modern networks. In: Proceedings of the 2005 ACM/IEEE conference on supercomputing, SC '05, Washington, pp 506–513

35. Yu W, Buntinas D, Graham RL, Panda DK (2004) Efficient and scalable barrier over quadrics and myrinet with a new NIC-based collective message passing protocol. In: 18th international parallel and distributed processing symposium (IPDPS'04) – Workshop 8, vol 9. Santa Fe, p 182b
36. Yu W, Buntinas D, Panda DK (2003) High performance and reliable NIC-based multicast over myrinet/GM-2. In: Proceedings of the international conference on parallel processing, Kahosung, p 197
37. Yu W, Sur S, Panda DK, Aulwes RT, Graham RL (2003) High performance broadcast support in LA-MPI over quadrics. In: Los Alamos computer science institute symposium, LACSI 03, Santa Fe

COMA (Cache-Only Memory Architecture)

► [Cache-Only Memory Architecture \(COMA\)](#)

Combinatorial Search

LAXMIKANT V. KALÉ, PRITISH JETLEY
University of Illinois at Urbana-Champaign, Urbana, IL, USA

Synonyms

[State space search](#)

Definition

Combinatorial search involves the systematic exploration of the space of configurations, or *states*, of a problem domain. A set of *operators* can transform a given state to a series of successor states. The objective of the exploration is to find one, all, or optimal goal states satisfying certain desired properties, possibly along with a path from the start state to each goal. Combinatorial search has widespread applications in optimization, logic programming, and artificial intelligence.

Discussion

Given an implicitly defined set, combinatorial search involves finding one or more of its members that satisfy specific properties. More formally, it entails the systematic assignment of discrete values from a finite range to

each of a collection of variables. Each member of the set represents a configuration or *state* that the basic elements of the problem domain can assume. Therefore, the set is also called a *state space*. Combinatorial optimization problems lend themselves particularly well to expression in this *state space search* paradigm. The basic components of state space search are the *enumeration* of states and their *evaluation* for fitness according to the criteria defined by the problem. Since the state space of these problems can be extremely large, explicit generation of the entire space is often impossible due to memory and time constraints. Therefore, the set of possible states is generated on the fly, by transforming one state into another through the application of a suitable operator. However, in the absence of a mechanism to detect the generation of duplicates, this procedure might begin exploring a path of infinite length, thereby failing to terminate. This is the problem of *infinite regress*, and can be mitigated by one of two techniques. The first relies on the comparison of a generated state with its ancestors, which are stored either in a distributed table or passed from a parent to its newly-created child state. Another approach constrains the search procedure to consider only cost-bounded solutions. This is the case, for instance, in the Iterative Deepening A* algorithm discussed later. In certain situations, the search may be guided by a *heuristic* measure of distance between states: those considered closer to the goal may be given priority of consideration over those believed to be more distant.

Searching for All Feasible Solutions

A basic combinatorial search problem is one in which all feasible configurations of the search space are desired. An example is the *N*-Queens problem: find all configurations of *N* queens on an $N \times N$ chessboard such that no queen *attacks*, i.e., shares the same row, column, or diagonal, with another. In a particular formulation, a state of the problem is represented by a configuration of the chessboard in which each of the first k rows holds a non-attacking queen and the remaining $N - k$ are empty. The enumeration process can be described as a tree search. The root represents a state in which no queens have been placed on the board. The rest of the search tree is defined implicitly as follows: given a state s with k non-attacking queens placed in the first k rows, its child state has the same arrangement of queens as s , and an

additional queen placed in the next empty row. Defining the search tree in this fashion is advantageous: the tree can be preemptively pruned at nodes that cannot possibly yield solutions. In the running example, child states are only generated if they do not produce conflicts with previously placed queens.

The search strategy described above is a type of *multistep decision procedure* (MDP). Each step of the algorithm generates a new state s from its parent p by deciding an element of the configuration space. At step $k+1$ of N -Queens, this element is the position of a queen in row $k+1$ of the $N-k$ remaining rows. An MDP ensures that no duplicate states are generated. In particular, there can be no recurring states along the path from the start to the current state, so that infinite regress cannot occur.

The tree search defined above is useful even when the configurations are defined differently. Consider the problem of finding a knight's tour on the chessboard: Starting at a square and using only legal moves, can a knight visit every square on the board exactly once, returning finally to the starting square? This is a special instance of the Hamiltonian Circuit Problem: given a graph, a circular path must be found that visits every vertex exactly once. Here, the vertices correspond to squares on the chessboard. An edge (u, v) connects vertex u to v such that v can be reached in one knight-move from u . To enumerate all such tours, a tree search might define a state as a knight's path of length k originating at the start. Each child extends the path by adding a move to it.

Variable Selection

Given a particular node in the search tree, there is a choice of which branch to select when considering new children. This is called *variable selection*. Continuing with the N -Queens example, at step $k+1$, *any* of the remaining $N-k$ rows may be chosen as the recipient of the $k+1$ -th queen, not just the $k+1$ -th row on the chessboard. The size of the tree beneath the current node (and therefore the effort involved,) can be significantly affected by the choice of branching variable. A good heuristic is to select the most constrained variable at each step. In the N -Queens example, this would mean placing the next queen in the row with the fewest non-attacked squares.

Parallelization

In a sequential search, the tree is typically explored in its entirety by a depth-first procedure. A stack is used to hold unexplored children at each level. Given a search tree of depth d and branching factor b , this only requires $O(bd)$ memory, in contrast to the $O(b^d)$ size of the search space. A parallel implementation of this search procedure requires the distribution of work into discrete chunks called *tasks*. To ensure the efficient execution of these tasks on processors, the inter-related issues of task creation, *grainsize* control, and load balance must be considered. Grainsize can be defined roughly as the ratio of computation work to number of messages sent. There is a certain overhead in creating tasks, and a separate overhead if the task description is moved to another processor. Thus, it is important to keep the *average* grainsize above a certain threshold to limit the impact of parallel overhead. At the same time, no single task should be so large as to make all processors wait for its completion.

A parallel search may define a task as the subtree beneath a single tree node. Grainsize estimation then requires a simple metric which is correlated to the computational cost of a node. In N -Queens this could be the number of queens that remain to be placed. It is not an exact measure: sometimes a node with many queens remaining may still generate very little work under it, because of the impossibility of finding a solution there. With a suitable metric formulated, a threshold amount of work may be set, below which new tasks are not created; such subtrees are explored sequentially. Continuing with the N -Queens example, an efficient serial backtracking mechanism may be employed when, say, only $m \leq N$ queens remain. The threshold m could be estimated by the exploration of small parts of the search space.

Another technique defines a task as a set of frontier nodes. The exploration proceeds in a depth-first manner, maintaining an explicit stack. At some point, the stack may be split in two (or more) pieces, with each piece assigned to a new task on a possibly different processor. Typically, nodes near the bottom of the stack are used to create a new task. However, *vertical splitting* has also been proposed, wherein half of the unexplored branches at each level of the stack are picked to form starting positions for a new task. This strategy works

well for irregular search spaces, but can be expensive for deep stacks.

The literature discusses two methods to determine *when* stacks are split. The first combines grainsize control with the load balancing strategy: in “work-stealing” a processor’s stack is split upon receipt of a request for work from an idle processor. The idea was first described by Lin and Kumar [1] and later formalized by the Cilk system [2]. An alternative, due to Kalé et al. [3], aims to separate the two issues: the amount of work done by a task is tracked. Once an amount of work above a certain threshold T has been performed, the stack is split into k pieces, each assigned to a new task. This ensures that the grainsize is $T/(k + 1)$. However, care must be taken to avoid long chains of large tasks, i.e., when exactly one child has significant work, whereas others do not.

Load Balancing

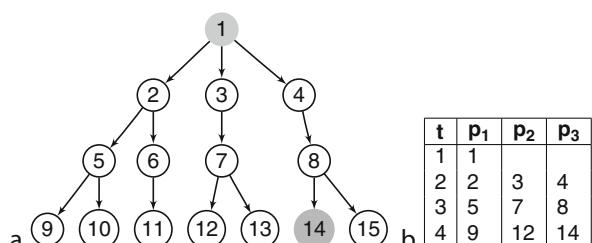
Early literature classifies load balancers as *sender-* or *receiver-initiated*. Work-stealing is a receiver-initiated load balancing technique. When a processor has no work, it steals work from a randomly chosen processor. It has been shown that random stealing of shallow nodes is asymptotically optimal, in that it leads to near-linear speedups [2]. The stealing mechanism could vary from messaging (on distributed memory systems) to exclusive shared queue access. In contrast, sender-initiated schemes assign newly spawned tasks to some processor, either randomly, or based on a load metric such as a queue size.

This taxonomic distinction fails to hold for load balancing schemes where each processor monitors the size of its own queue and that of its “neighbors,” periodically rebalancing them. Initially, a created task is placed on the creating processor’s queue. Unlike random assignment, this avoids unnecessary communication. Depending on the thresholds used to exchange queues and the periodicity of load exchanges, such schemes can tune themselves effectively. At one extreme, they can approximate work-stealing, where neighborhoods are global, an empty queue is used as a trigger, and rebalancing is done by selecting a random neighbor. Another advantage is their proactive agility: they increase efficiency by moving work before a processor goes idle. However, these schemes create more communication traffic.

Searching for Any Feasible Solution

Unlike the N -Queens or graph coloring problems where the objective is to identify *all* ways of satisfying given constraints, certain situations require the generation of *any* satisfactory configuration. A classic example is the 3-SAT problem. **Value ordering** becomes an important heuristic in this context: given b children of a node, explore the subtree of that child next whose subtree is most likely to contain a solution. Problem-specific figures of merit are used to order children. For instance, a simple 3-SAT heuristic might rank the two values possible for a variable (true or false) in decreasing order of the number of clauses satisfied by each.

The same parallelization techniques as outlined for the all-solutions search may be used, modifying them to terminate when a solution is found. However, note the *speculative* nature of the methods in this context: regions of the tree that may not be visited in a sequential procedure are searched concurrently in the hope that a solution may be found quicker. This can lead to anomalies in parallel performance [4], because nodes are not visited in the same order by the parallel algorithm. In the worst case, an added processor may generate useless work for other processors in the form of nodes that do not yield solutions. This situation is referred to as a *detrimental speedup anomaly*. On the other hand, as illustrated in Fig. 1, this addition of resources could lead to superlinear speedups (the *acceleration anomaly*.) Even on a fixed processor count, variations in the timing



Combinatorial Search. Fig. 1 Acceleration anomaly in parallel depth-first search: (a) illustrates the state space tree of a problem. Search begins at node 1; node 14 is the goal state. A serial depth-first search for the first solution takes 14 steps; (b) shows a schedule for the parallel depth-first search of the same tree with three processors. The search takes four steps, for a speedup of $14/4 = 3.5 > 3$

of load balancing could lead to widely differing execution spans between runs. Kalé et al. [3] obtain consistent speedups by prioritizing node exploration in the following manner: if a node with k children has priority q , each child's priority is obtained by appending its $\log k$ bit rank to q . This leads to a lexicographic ordering of tree nodes. Therefore, all descendants of a node's left child have higher priority than the right child's descendants. A prioritized queue (on a shared memory machine) or a prioritized load balancing scheme is used to steer processors toward high priority work. This scheme also leads to low memory usage: the search frontier forms a characteristic broom shape that sweeps the state space in accordance with the ordering on the nodes. Assuming a constant branching factor, the total memory usage with p processors is $O(p + d)$ rather than the $O(pd)$ requirement of the depth-first tree. On distributed memory machines, achieving this bound depends on the quality of the prioritized load balancer. Furthermore, since such schemes move work from the left part of the tree to all processors, they tend to have a sizable communication overhead. Therefore, a simple depth-first search may be more suitable in some cases.

Searching for an Optimal Solution

Some problems assign measures of fitness to feasible solutions. This renders certain solutions “better” than all others by some metric, so that the objective becomes the search for an *optimal* solution. Examples include integer programming, solving the 15-puzzle (or Rubik’s cube) with the fewest moves, and the search of a graph for a least cost Hamiltonian cycle. One could use the all-solutions methods discussed previously, and then select the best solution among all found. However, this is typically wasteful, and more efficient search methods are available.

A* Search and Iterative Deepening

In many problems, it is possible to define a heuristic function that, given a node n , computes a lower bound on the cost of any solution in the subtree beneath it. Such a function is called an *admissible* heuristic. For instance, in the 15-puzzle, the number of tiles that are out of place is an admissible heuristic: at least that many moves are needed to attain the goal state, since only one tile is shifted per move. (The “Manhattan distance” of each tile from its final position is a stronger heuristic.)

The lower bound on the *total* cost of a node is the sum of this value and the cost of arriving at it from the start. If unopened nodes are processed in ascending order of their lower-bounds, it can be shown that the first solution found is optimal [5]. This is called the A* search procedure. The A* strategy leads to an exponentially sized node queue; by contrast, depth-first search is highly memory efficient. The Iterative Deepening A* (IDA*) technique developed by Korf [6] combines the best properties of the two. It is applicable when the solution cost is quantized. For example, the number of moves needed in the 15-puzzle is an integer. If there is no solution of d moves, the procedure looks for a solution of $d + 2$ moves. (Because of parity arguments, the number of moves needed for a given starting state is known to be either even or odd.) A depth-first search bounded by a cost d may then be organized. An admissible heuristic is used to stop the search below nodes with cost lower bounds greater than d . If there is no solution of cost d , the bound is increased to the next possible value ($d+2$ for the 15-puzzle), and the search is restarted. Although there is duplication of the higher levels of the search tree, this technique is asymptotically optimal in the amount of work done [6]. Further, it offers control over the amount of memory used, while maintaining the property that the first solution found is the optimal one. Moreover, this method precludes the problem of infinite regress without the need for duplicate detection.

An effective parallelization of IDA* is described by Kalé et al. [3]. It uses bitvector prioritization to overlap execution of multiple iterations with different bounds. This ensures that the last iteration (the one in which the solution is found) is executed in a way that minimizes speculative loss, leading, as before, to consistent and monotonic speedups. In addition, the latter part of one iteration, where it winds down causing low processor utilization, is speculatively overlapped with the start of the next, thus improving efficiency.

Branch-and-Bound

Optimization problems can also be solved using the branch-and-bound technique, wherein properties of partial solutions are used to discard infeasible portions of the search tree. This can reduce the effort expended in finding optimal solutions. The main components of this mechanism are the *branching* and *bounding* procedures. Given a node n in the search tree, the branch

procedure generates a finite number of children. The bounding procedure assigns to each child a cost bound that determines when the child is explored, and whether it is explored at all. Nodes are pruned if they need not be explored. In addition, the bounding function must be *monotonic*: the bound of a node may be no better than the bound of its parent. An *exploration* mechanism is required to choose between the children of a node generated at each step. Such a mechanism may prioritize children based on their depth or their estimated ability to yield a solution. The procedure terminates when all subproblems have either been explored or pruned.

Consider the Traveling Salesman Problem. Whereas more efficient bounding techniques have been discussed in the literature, the following naïve procedure is presented for the purpose of exposition. A node n represents a partial solution comprising paths between cities such that they form a (possibly incomplete) tour. The children of n are enumerated by listing cities which can be visited from the most recently added destination. A partial solution has a cost bound equal to the length of the path that it represents. This is a monotonic lower bound, since the cost of a path through a child of n is at least as great as the cost of a path through n itself. The algorithm tracks the cost c of the cheapest complete tour encountered up to a certain point in the search. Notice that the tree can be pruned at children with lower bounds *greater* than c . Starting with an empty tour, all possible tours may be considered using this procedure.

The basic data structure in the branch-and-bound is a prioritized queue that stores the nodes comprising the frontier of the search. Anomalous speedups can result if these nodes are processed in an unordered fashion. The literature suggests *unambiguous* heuristic functions [7] for shared queues of nodes. Such a function h differentiates between nodes based on their values, so that for nodes n_1 and n_2 , $h(n_1) = h(n_2) \Leftrightarrow n_1 = n_2$. Even with consistent speedups, performance is bound by queue locking and access overheads. These can be mitigated by the use of concurrent priority queues.

In distributed memory implementations, the frontier may either be stored in a centralized or distributed manner. The former strategy usually engenders *master-slave* parallelism, where the master processor distributes work from a central node pool to worker processors.

This approach retains complete information about the global state of the search, thereby enabling an optimal exploration of the frontier without any speculative work. However, it is not scalable due to the bottleneck at the master. Distributing the node pool affords more autonomy to individual processors. Efficiency can be increased if a processor broadcasts newly encountered lower bounds to others. This helps other processors discard nodes that cannot yield optimal solutions. However, since it is hard to track node quality, effort might be wasted in exploring infeasible nodes for want of accurate bound information. *Quality equalization* may be performed to reduce speedup anomalies and distribute useful work equitably. This is either done periodically or when an associated trigger is activated. Equalization involves the movement of promising nodes between processors, which can be done in a hierarchical fashion to reduce communication. Grama and Kumar [8] and Kalé et al. [3] survey such parallel branch-and-bound techniques.

Bidirectional Search

Problems such as Rubik's Cube stipulate a priori the exact configuration of the goal state. In such problems, a *bidirectional* search may be employed to construct a path to the goal from the start state. By initiating two paths of search, one moving forward from the start state and the other backward from the goal, the size of the search space explored can be reduced substantially. On average, a unidirectional search of a tree of depth d and branching factor b visits $O(b^d)$ states to find an optimal solution. In contrast, by starting two opposing searches that meet, on the average, at depth $d/2$, only $O(b^{d/2})$ states are explored.

There are two main ways of organizing the forward and backward searches. The first method uses a backward depth-first procedure to exhaustively explore the tree starting from the goal, up to a certain height h above it. The states generated by this backward search are stored in the *intermediate goal layer*, the size of which is limited by the amount of memory available. The forward search is a (memory efficient) depth-first procedure or a best-first search with iterative deepening. Instead of checking for goal states, the forward search looks for each enumerated state in the intermediate layer. A match indicates the presence of a solution.

For distributed memory systems, the intermediate layer may either be replicated on every processor, or distributed across all available processors. Using replicas of the intermediate layer lowers the communication cost of the algorithm, but forces a reduction in the depth of the backward search. Kalé et al. [3] describe the use of distributed tables in multiprocessor bidirectional search.

A second class of bidirectional search schemes uses best-first techniques in either direction. This approach has its pitfalls: The frontiers of the forward and backward search may pass each other, resulting in two non-intersecting paths (in opposite directions) from the start to the goal. Therefore, instead of doing less work, the algorithm will have performed more work than a unidirectional search, yielding poor performance. To overcome this situation, *wave-shaping* algorithms attempt an intersection between the forward and backward search frontiers. Nelson and Toptsis [9] survey bidirectional search and provide parallel variants of pioneering uniprocessor techniques. Pohl presented the original non-wave-shaping uniprocessor bidirectional search. De Champeaux and Sint formulated a wave-shaping algorithm which estimates the distance between the advancing frontiers to encourage an intersection. This was improved upon by Politowski and Pohl to reduce the computational complexity of the heuristic.

Kaindl and Kainz [10] provide empirical evidence suggesting that bidirectional search is inefficient not because of non-intersecting frontiers, but because of the effort expended in trying to establish the optimality of the various paths constructed upon their meeting. Even with the most efficient implementations, bidirectional search can sometimes fail to provide the expected speedups, because of the structure of the problem's state space. Consider the game of peg-solitaire. The initial states of the game have many pegs but few spaces to allow jumps, so that there are few available moves. As the balance between free spaces and pegs becomes more even, more moves can be made and the branching factor increases. Toward the terminal stages of the game, few pegs remain on the board and the average branching factor once again reduces dramatically. Therefore, the state space of peg-solitaire does not form a tree that fans out with increasing depth. Consequently, a unidirectional search from the start state visits significantly

fewer than $O(b^d)$ nodes, and there is no practical advantage to using bidirectional search.

Game Tree Search

Game-playing can be represented by trees that trace the sequence of moves made by adversaries. Levels of these trees are marked MAX and MIN in an alternating fashion, depicting the moves made by the player and the opponent respectively. The leaves represent the various outcomes of the game, and are assigned values commensurate with their estimated favorability. However, the enumeration of all possible outcomes and paths to their corresponding leaves can be prohibitively expensive. Chess, for example, has on the order of 10^{43} states. Therefore, given a current state, modern game-playing strategies limit the search for good states to a certain *look-ahead* depth d from it. The *minimax principle* posits that under the assumption of rational play, an optimal strategy can be formulated by picking the most favorable child c of the current node n at each turn. A good approximation of the optimal move at n can be computed by evaluating all nodes under it up to a sufficient depth d , and choosing the best child c . Using a depth-first procedure, this requires $O(b^d)$ time and $O(bd)$ space, where b is the branching factor.

Alpha-Beta Pruning

The alpha-beta pruning procedure tracks the values of nodes encountered in the left-to-right, depth-first traversal of the tree to reduce the amount of work done in searching for optimal moves. For this purpose, the procedure tracks the lower bound on projected value of each MAX node (α) and the upper bound on the projected value for each MIN node (β). Consider a MAX node n that has two MIN children l and r . Suppose that the value of its left child l is found to be $v(l) = 15$. Then, the procedure may prune the tree at any children of r once it processes a c such that r is the parent of c and $v(c) < 15$. This pruning reduces the number of nodes significantly. However, the left-to-right order of tree traversal makes it challenging to formulate an efficient parallelization.

Parallel forms of the algorithm address the different kinds of node in the tree. *Principal Variation Splitting* evaluates the leftmost branch of a game tree before allowing the parallel evaluation of sibling nodes. Ideally,

the leftmost branch represents the optimal sequence of moves for the player (game tree branches are generally ordered so that the principal variation is the leftmost branch of the tree) and so yields the tightest bounds for the pruning of the rest of the tree, greatly reducing the amount of work done in parallel. Once the leftmost branch l has been scored, sibling nodes are evaluated in parallel using the bounds obtained from l . The siblings of a node n to its right may be sent refinements of bounds as these are calculated by n . The amount of parallelism is limited by the branching factor of the problem. Further, load imbalance between siblings causes synchronization delays.

The *Young Brothers Wait Concept* (YBWC) orders nodes similarly: the first child of a node (the *eldest* brother) must be explored before the others (the *younger* brothers) are examined. This scheme extends parallel evaluation beyond the principal variation nodes. Processors are said to *own* nodes if they are evaluating the subtrees beneath them. Initially, all processors except one, p_0 , are idle. Processor p_0 is given ownership of the root node. Idle processors request work from those that have satisfied the YBWC sibling order constraint, i.e., those processors that have evaluated at least one eldest brother n . This establishes a master–slave relationship between the sender and the recipient of work, and marks a *split-point* at N , which is the parent of n . The master and the slave cooperate to solve the tree under N . Further, a master that has become idle can request work from a slave that has not completed evaluation. Improved bound and cut-off information is shared with collaborating processors. A stronger formulation [11] of the parallelism constraint has been used to yield good speedups in chess-playing on distributed memory machines. *Dynamic Tree Splitting* uses a similar collaboration technique in the context of shared memory systems. However, split-points are chosen according to constraints expressed in terms of the α and β values of nodes.

The Deep Blue computer chess system [12] used a master-slave approach to parallelism. Upper levels of the tree were evaluated by the master, the slave processors being allotted work as the search grew deeper. To alleviate the performance bottleneck at the master, slave nodes were always kept busy with “on-deck” jobs. Since search was performed using a hybrid

software-hardware approach, load was balanced by pushing long hardware searches into software, and by sharing large pieces of work between workers. In 1997, Deep Blue defeated the then-reigning world champion Garry Kasparov.

AND-OR Tree Search

AND-OR trees arise naturally in the execution of logic programs, and in the problem-solving and planning literature within artificial intelligence. The min-max trees used in evaluating two-person games also reduce to AND-OR trees for the special case when the value of a node can only be a win or a loss. Another example of such trees arises in the graph coloring problem, when subgraphs may be colored independently following the coloring of a partitioning layer of vertices.

In logic programming terminology, the top-level query is a conjunction of literals called *goals*. Typically, multiple clauses are available to solve a given literal. Each clause is a conjunction of literals. The evaluation of a query then naturally leads to an AND-OR tree. There are two kinds of node in an AND-OR tree: an AND-node requires solutions to each of its children, whereas an OR-node requires a solution to at least one of its children. Search procedures used for simple state space search may be extended to this case. In particular, to *construct* a solution, an AND-node requires that information be sent up from its subtrees. A number of approaches to the effective organization of this search procedure have been surveyed, chiefly in the context of logic programming, by Gupta et al. [13].

An interesting issue concerns the dependence between the sub-problems represented by the children of AND nodes. Consider a query such as: $p(a, X), q(b, Y), r(c, X, Y, Z)$. Upper case letters represent variables that are instantiated by a solution. A solution to p may require that X have a value d , and a solution to q may require that Y have a value e . Unless there is a solution to r that also has $X = d$ and $Y = e$, these solutions to p and q are not useful. Therefore, the search space under r is constrained by using solutions produced by p and q . Whereas the subtrees corresponding to p and q can be explored in parallel, for r , a subtree can only be created for each instance produced by the cross-product of solutions to p and q . This “consumer-instance” parallelism is supported by the REDUCE-OR

process model [14]. Further, to avoid wasted work, the occurrence of duplicate states along distinct paths must be addressed. This issue is separate from the problem of infinite regress described earlier, which arose due to the generation of duplicates along the *same* path. To avoid duplication of effort, the newer state may be made a client for the result generated by the older instance. The issue of duplicates arises in other search patterns as well (Game Tree Search, IDA*, etc.) In these paradigms, it may be advisable to terminate one of these two paths, depending on their (under)estimated costs.

More Search Techniques

Several search techniques exist in addition to the ones described in this chapter. Path finding in the context of imperfect graph information is done using dynamic algorithms such as D* and LPA*. Little work has been done on the parallelization of these techniques.

Many *metaheuristic* techniques have been developed. The metaheuristic approach uses generalized search mechanisms, usually inspired by physical and natural phenomena, in lieu of problem-specific heuristics and techniques. Examples include Genetic Algorithms, Simulated Annealing, Local and Tabu Searches, etc. The use of graphics processors as offload devices to aid the solution of such state space search problems has also been considered recently.

Related Entries

- [Charm++](#)
- [Cilk](#)
- [Logic Languages](#)

Bibliography

1. Lin Y-J, Kumar V (1991) And-parallel execution of logic programs on a sharedmemory multiprocessor. *J Logic Program* 10(1–4):155–178
2. Blumofe RD, Joerg CF, Kuszmaul BC, Leiserson CE, Randall KH, Zhou Y (1996) Cilk: an efficient multithreaded runtime system. *J Parallel Distribut Comput* 37(1):55–69
3. Kalé LV, Ramkumar B, Saletore V, Sinha AB (1993) Prioritization in parallel symbolic computing. In: Ito T, Halstead R (eds) *Lecture notes in computer science*, vol 748. Springer-Verlag, Heidelberg, pp 12–41
4. Rao VN, Kumar V (1998) Superlinear speedup in parallel state-space search. In: *Proceedings of the eighth conference on foundations of software technology and theoretical computer science*, Springer, London, UK, pp 161–174

5. Dechter R, Pearl J (1985) Generalized best-first search strategies and the optimality of A*. *J ACM* 32(3):505–536
6. Korf RE (1985) Depth-first iterative-deepening: an optimal admissible tree search. *Artif Intell* 27:97–109
7. Li G-J, Wah BW (1986) Coping with anomalies in parallel branch-and-bound algorithms. *IEEE Trans Comput* 35(6):568–573
8. Grama A, Kumar V (1999) State of the art in parallel search techniques for discrete optimization problems. *IEEE Trans Knowl Data Eng* 11(1):28–35
9. Nelson PC, Toptsis AA (1992) Unidirectional and bidirectional search algorithms. *IEEE Softw* 9:77–83
10. Kaindl H, Kainz G (1997) Bidirectional heuristic search reconsidered. *J Artif Intell Res* 7:283–317
11. Feldmann R, Mysliwietz P, Monien B (1994) Studying overheads in massively parallel min/max-tree evaluation. In: *SPAA '94: proceedings of the sixth annual ACM symposium on parallel algorithms and architectures*, ACM Press, New York, pp 94–103
12. Campbell M, Hoane AJ, Hsu F-H (2002) Deep blue. *Artif Intell* 134(1–2):57–83
13. Gupta G, Pontelli E, Ali KAM, Carlsson M, Hermenegildo MV (2001) Parallel execution of prolog programs: a survey. *ACM Trans Program Lang Syst* 23(4):472–602
14. Kalé LV (1991) The REDUCE-OR process model for parallel execution of logic programs. *J Logic Program* 11(1):55–84

Commodity Clusters

- [Clusters](#)

Communicating Sequential Processes (CSP)

- [CSP \(Communicating Sequential Processes\)](#)

Community Atmosphere Model (CAM)

- [Community Climate System Model](#)

Community Climate Model (CCM)

- [Community Climate System Model](#)

Community Climate System Model

PATRICK H. WORLEY¹, MARIANA VERTENSTEIN²,
ANTHONY P. CRAIG²

¹Oak Ridge National Laboratory, Oak Ridge, TN, USA

²National Center for Atmospheric Research, Boulder, CO, USA

Synonyms

Community atmosphere model (CAM); Community climate model (CCM); Community climate system model (CCSM); Community earth system model (CESM); Community ice code (CICE); Community land model (CLM); Model coupling toolkit (MCT); Parallel I/O library (PIO); Parallel ocean program (POP)

Definition

The Community Climate System Model, CCSM, is a parallel climate model consisting of four parallel geo-physical component models (atmosphere, land, ocean, and sea ice) that exchange boundary data periodically through a parallel coupler component. It is a freely available community model developed by researchers funded by the US Department of Energy (DOE), the National Aeronautics and Space Administration (NASA), and the National Science Foundation (NSF), and maintained at the National Center for Atmospheric Research. CCSM targets a wide variety of computing platforms, ranging from workstations to the largest DOE, NASA, and NSF supercomputers.

Discussion

Introduction

Investigating the impact of climate change is a computationally expensive process, and as a result modern climate models are designed to take advantage of high performance computing (HPC) architectures. The Community Climate System Model (CCSM) is the best known of a series of climate models that have been developed by and maintained at the National Center for Atmospheric Research (NCAR), with contributions from external researchers funded by the US Department of Energy (DOE), National Aeronautics

and Space Administration, and National Science Foundation. In 2010 the model was extended and renamed the Community Earth System Model (CESM). This article focuses on CCSM, reserving a brief discussion of CESM for section ►“Community Earth System Model”.

CCSM consists of a system of four parallel geophysical component models (atmosphere, land, ocean, and sea ice) that exchange two-dimensional boundary data (flux and state information) periodically through a parallel coupler. The coupler coordinates the interaction and time evolution of the component models, and also serves to remap the boundary-exchange data in space. The atmosphere model is CAM, the Community Atmosphere Model. The ocean model is POP, the Parallel Ocean Program. The land model is CLM, the Community Land Model. The sea ice model is CICE, the Community Ice Code.

All component models are hybrid parallel application codes, using MPI, the Message Passing Interface, to define and coordinate distributed-memory parallelism and OpenMP to define and coordinate shared-memory parallelism. The components are all, for the most part, written in Fortran 95, though this is not a requirement. CCSM is unusual among parallel computational science models in that the hybrid message-passing/shared-memory parallel programming paradigm was implemented within many of the component models early on, and is now supported in all components except the coupler. It has been found that using both message-passing and shared-memory parallelism has been critical for achieving good performance on many past and current HPC architectures. Another unusual aspect of CCSM is that it is a community code that is evolving continually to evaluate and include new science. In consequence it has been very important that CCSM be easy to maintain and port to new systems, and that CCSM performance be easy to optimize for new systems or for changes in problem specification or processor count.

CCSM supports a large range of computational grid resolutions, with current production runs spanning a nominal 3° horizontal grid (approximately 4,600 grid points per vertical level for the atmosphere and land and 11,600 for the ocean and sea ice) for paleoclimate simulations, to 0.25° atmosphere and land horizontal grids (approximately 885,000 grid points per level) and 0.1° resolution ocean and sea ice grids

(approximately 8,640,000 grid points per level) that can resolve global tropical cyclones. Furthermore, the atmosphere component can span the range of altitudes from the Earth's surface to the thermosphere. CCSM also supports a variety of physical process options. Examples include different atmospheric chemistry packages and the option to run with a full carbon/nitrogen biogeochemical cycle in the ocean and land. [Figure 1](#) is a snapshot of precipitable water from a high resolution simulation.

For low resolution simulations and less expensive physical process options, CCSM can be run on small clusters and even on a laptop computer. For high resolution simulations and the more computationally expensive physics options, the largest available supercomputers are required, both to satisfy the memory requirements and to achieve a reasonable throughput rate for the simulations.

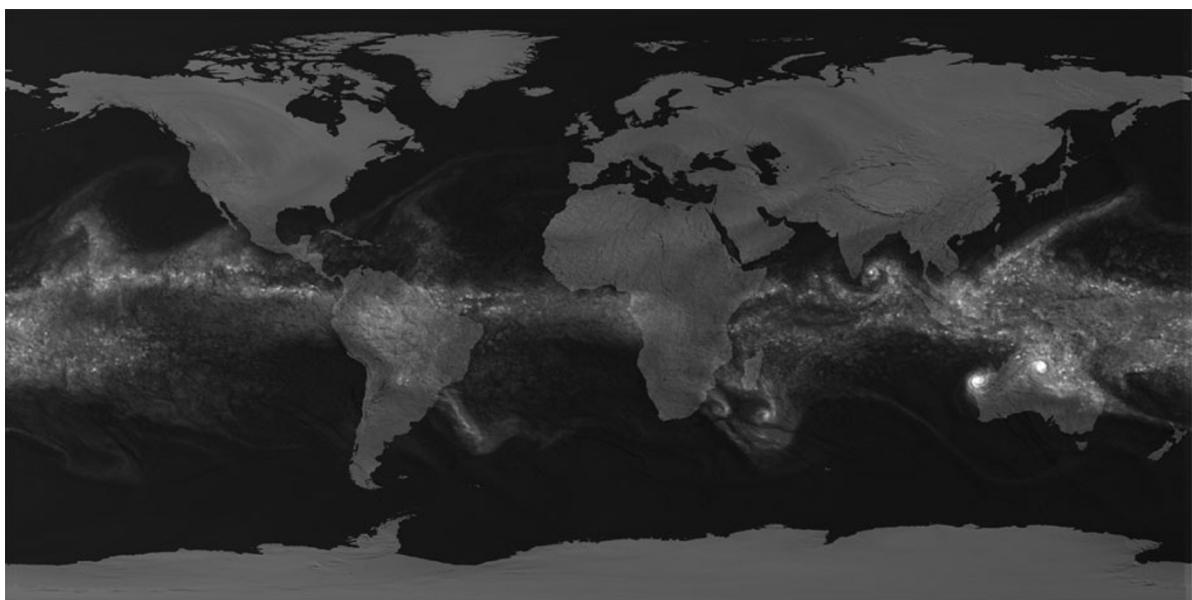
Each component model is a parallel application code in its own right, and was developed for the most part independently from the other component models. These are discussed in turn. Common technologies used in multiple components are also described.

Terminology

To help differentiate between MPI and OpenMP-based parallelism, processes will be referred to as MPI processes or tasks. Computational threads associated with an MPI process but which are spawned by the OpenMP runtime system will be referred to as OpenMP threads.

Each component model utilizes one or more spatial computational grids with at least two horizontal indices, e.g., representing longitude and latitude, and one vertical index. All grid points with a given horizontal location are referred to as a "vertical column." For some of the component models each grid point also represents a geographical region surrounding that physical location. In this case a grid point is also referred to as a "grid cell."

Component models approximate the solution at a discrete set of simulation times, with results at one simulation time dependent only on data and results associated with the same or earlier simulation times. Moreover, results for earlier simulation times are always completed before the computation for a later time is begun. The computation for a given simulation time is a "timestep."



Community Climate System Model. Fig. 1 A snapshot of the column integrated precipitable water from a high-resolution CCSM simulation. The snapshot is from early January and shows well-resolved winter-time cyclone activity in the Indian Ocean. Visualization courtesy of J. Daniel of Oak Ridge National Laboratory

The parallel implementation of each component model is based on one or more decompositions of a computational grid, with each MPI process assigned data and the responsibility of computing results associated with a subset of the grid. Computing the next timestep for a given subset of the grid will sometimes require data and results associated with grid points external to the subset. These external grid points are referred to as the “halo” for the grid subset. For a parallel algorithm based on grid decomposition, a “halo update” is the interprocess communication required to acquire the halo information that resides in the memory space of other processes.

Community Climate System Model

The first fully coupled system in the CCSM line of development, version 1 of the Community Climate System Model (CCSM1), was released in 1996. Although CCSM1 included support for both distributed-memory multiprocessor systems (via message passing) and parallel shared-memory vector systems, it was officially supported only on the NCAR Cray computing systems. A subsequent minor release added support for the SGI Origin 2000 system. CCSM2, the Community Climate System Model version 2, was released in 2002 and was the first version to use CAM, CLM, and POP for the atmosphere, land, and ocean component models, respectively. Target architectures included IBM SP systems, SGI Origin 2000, and Compaq/DEC Alphaserver. Version 3 (CCSM3) was released in June 2004 and version 4 (CCSM4) was released in April, 2010. CCSM4 was the first public release to use CICE for the sea ice component. CCSM4 theoretically runs on any distributed-memory system with an MPI library and a compiler capable of compiling CCSM and required libraries, such as netCDF. At the high end, CCSM4 ran on Cray XT, IBM BlueGene/P, and IBM Power6 cluster systems in 2010.

For science reasons, the atmosphere, land, and sea ice models are partially serialized in time, limiting the fraction of time when all four CCSM components can execute simultaneously. In the first three versions of CCSM each component model and the coupler were run as separate executables assigned to nonoverlapping processor sets. As of CCSM4, the entire system is now run as a single executable and there is greatly increased

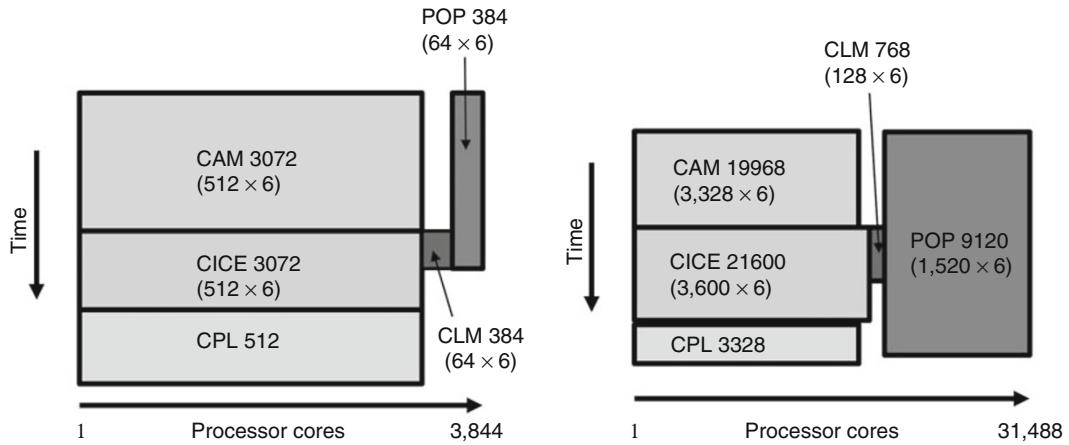
flexibility to select the component processor layout. It is typical for the atmosphere, land, and sea ice model to run on a common set of processors, while the ocean model runs concurrently on a disjoint set of processors. This is not a requirement, however, and CCSM can now run all components on disjoint processor subsets, all on the same processors, or any combination in between.

Each component model has its own performance characteristics, and the coupling itself adds to the complexity of the performance characterization. The first step in CCSM performance optimization is to determine the optimized performance of each of the component models for a number of different processor counts. This performance information is then used to determine how to assign processors to components in order to optimize the performance of CCSM as a whole. Cartoons of two example configurations for a Cray XT5 are displayed in Fig. 2. Figure 3 describes the performance scaling for the larger of the two problems used in Fig. 2.

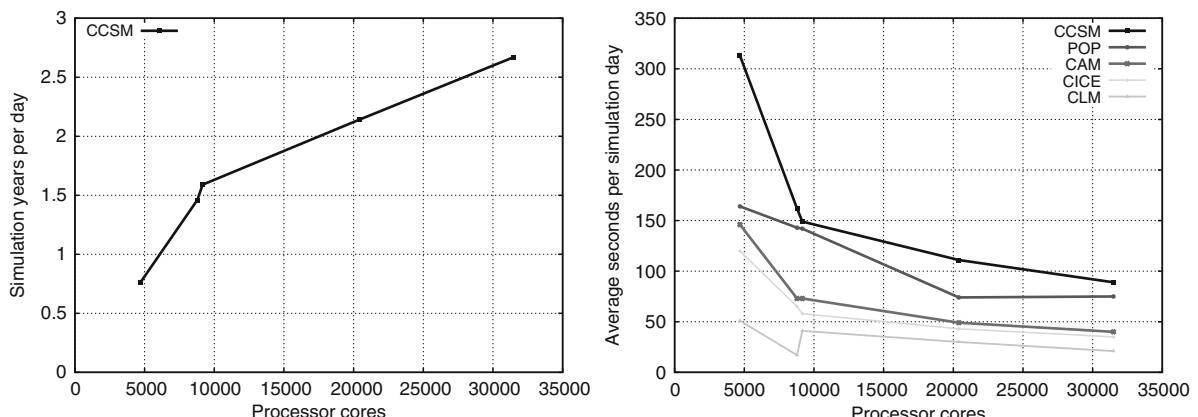
Community Atmosphere Model

CAM and its predecessors were all developed at NCAR. The first version (CCM0) was completed in 1982, followed by CCM1 in 1987, CCM2 in 1992, and CCM3 in 1996. CCM2 and prior versions targeted shared-memory parallel vector systems. An experimental distributed-memory parallel version of CCM2 was developed through the DOE CHAMMP (Computer Hardware, Advanced Mathematics, and Model Physics) program, targeting the Intel Delta initially and later ported to other systems including the Intel Paragon, IBM SP, and Cray T3E. However, CCM3 was the first official release with a distributed-memory parallelization option. In 2002, a new version was released and the name of the model was changed to the Community Atmosphere Model, to better reflect its role in the fully coupled climate system. CAM was also a significant departure from the CCM in terms of its software architecture, resulting in much improved extensibility, maintainability, portability, and parallel performance.

CAM is characterized by two computational phases: the dynamics, which advances the evolutionary equations for the atmospheric flow, and the physics, which approximates subgrid phenomena such as precipitation processes, clouds, long- and short-wave radiation, and



Community Climate System Model. Fig. 2 Cartoons of two example configurations for a Cray XT5 with two hex-core processors per compute node. The left configuration is for a simulation using one degree resolution horizontal grids on 3,844 processor cores. The right configuration is for a simulation using quarter degree resolution atmosphere/land grids and tenth degree ocean/sea ice grids on 31,488 processor cores. The number of cores used is listed per component, as well as the number of MPI tasks and number of OpenMP threads per task (*tasks* × *threads*). The time direction indicates the relative amount of time spent in each component, and whether components run sequentially or concurrently with each other. Note that the coupler will sometimes run sequentially with the ocean, for example, when the ocean and atmosphere are communicating, and sometimes concurrently



Community Climate System Model. Fig. 3 Example CCSM performance scaling on a Cray XT5 with two hex-core processors per compute node. The left graph is the throughput rate for a simulation using quarter degree resolution atmosphere/land and tenth degree ocean/sea ice horizontal grids. The right graph is the average wallclock seconds per simulation day for CCSM and for the atmosphere, land, ocean, and sea ice component models. For a given total processor core count, cores were assigned to each component so as to minimize total execution time. The number of cores assigned per component did not always increase with the total number of cores

turbulent mixing. Separate data structures and parallelization strategies are used for the dynamics and physics. The dynamics and physics are executed in

turn during each model simulation timestep, requiring that some data be rearranged between the two data structures each timestep.

CAM includes multiple compile-time options for the dynamics, referred to as dynamical cores or dycores. During 2010–2011 the most utilized dycore was a finite-volume flux-form semi-Lagrangian dynamical core (FV) formulated originally by Lin and Rood [8]. FV uses a tensor-product latitude \times longitude \times vertical-level computational grid over the sphere. A Lagrangian vertical coordinate is used to define flux volumes, within which the horizontal dynamics evolve. Vertical transport is modeled through evolution of the geopotential along each vertical column. A conservative Lagrangian surface remap is performed each model time step. Other supported dycores include an Eulerian global spectral method (EUL) and a spectral element method (SEM). Like FV, EUL uses a latitude \times longitude horizontal computational grid. In contrast, SEM uses a quasi-uniform “cubed sphere” horizontal grid that avoids the clustering of grid points near the poles that is typical of latitude \times longitude grids. All options as of June 2010 are described in [9].

The parallel implementation of the FV dycore is based on two-dimensional tensor-product “block” decompositions of the computational grid into a set of geographically contiguous subdomains. Each subdomain is assigned to a single process and no more than one block is assigned to any MPI process. A latitude-vertical decomposition is used for the main dynamical algorithms and a latitude-longitude decomposition is used for the Lagrangian surface remapping and (optionally) geopotential calculation. Halo updates are the primary MPI communications required by computation for a given decomposition. OpenMP is used for additional loop-level parallelism.

CAM physics is based on vertical columns and dependencies occur only in the vertical direction. Thus computations are independent between columns. The parallel implementation of the physics is based on a fine-grain latitude-longitude decomposition. Each subdomain, referred to as a “chunk,” is a collection of vertical columns. Multiple chunks can be assigned to a single MPI process, and OpenMP parallelism is applied to the loops over these chunks.

Transitioning from one grid decomposition to another, for example, latitude-vertical to latitude-longitude or dynamics to physics, may require that information be exchanged between processes. If the decompositions are very different, then every process may need to exchange data with every other process. If

they are similar, each process may need to communicate with only a small number of other processes (or possibly none at all).

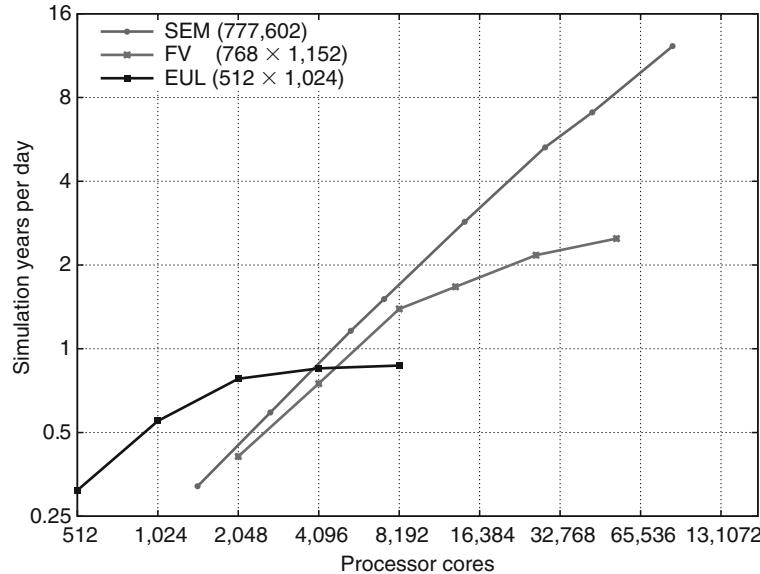
The computational cost in the physics is not uniform over the vertical columns, with the cost for an individual column depending on both geographic location and on simulation time. Some physics decompositions are better than others in equidistributing this cost (“balancing the load”) across the processes. A number of predefined physics decompositions are provided that attempt to minimize the combined effect of load imbalance and the communication cost of mapping to/from the dynamics decompositions. The optimal choice is a function of the problem specification, computer system, and number of processors used.

Historically CAM has been the performance bottleneck in fully coupled climate simulations. In consequence many performance optimization options have been implemented in the model, to make it easier to optimize performance for different problem specifications and target computer systems. Despite this flexibility, the parallel scalability of the model is limited when using either the FV or EUL dycores. As shown in Fig. 4, new dycores such as SEM demonstrate significantly improved scalability and will almost certainly become the default in the near future.

Community Land Model

CLM was developed at NCAR. It evolved from the effort to expand the strictly biogeophysical LSM, the NCAR Land Surface Model, to include the carbon cycle, vegetation dynamics, and river routing. The first version, CLM2.0, was released in 2002. Two major releases of the model have occurred since then, CLM3.0 and CLM4.0. The latter releases use MPI for distributed-memory parallelism and OpenMP for shared-memory parallelism. CLM is a single column (snow-soil-vegetation) model of the land surface, and in this aspect it is embarrassingly parallel.

Spatial land surface heterogeneity in CLM is represented as a nested subgrid hierarchy in which grid cells are composed of multiple landunits, snow/soil columns, and plant functional types (PFTs). Each grid cell can have a different number of landunits, each landunit can have a different number of columns, and each column can have multiple PFTs. The landunits represent the broadest spatial patterns of subgrid heterogeneity and



Community Climate System Model. Fig. 4 CAM throughput as a function of processor core count on an IBM BG/P with one quad-core processor per compute node for example simulations using the EUL, FV, and SEM dycores, respectively. Horizontal grid resolution used with each dycore is reported as either total number of grid points or as the dimensions of a two-dimensional grid. When using the FV dycore, the simulation used the same quarter degree grid as that used for Fig. 2. Grid resolutions when using the EUL and SEM dycores were chosen so that all simulations achieved comparable numerical accuracy. SEM data courtesy of M. Taylor of Sandia National Laboratories

currently are glacier, lake, wetland, urban, and vegetated. Columns represent potential variability in the soil and snow within a single landunit. Finally, PFTs represent differences in functional characteristics between categories of plants. Currently, up to 16 possible PFTs that differ in physiology and structure may coexist on a single column. All fluxes to and from the surface are defined at the PFT level.

Grid cells are grouped into blocks (called “clumps”) of nearly equal computational cost, and these clumps are subsequently assigned to MPI processes. When run serially or with MPI-only parallelism, each process has only one clump. When OpenMP is enabled, the number of clumps per process is set to the maximum number of OpenMP threads available.

The computational cost of a grid cell is approximately proportional to the number of plant functional types (PFTs) contained within it so some gridcells will be higher cost than others. Since similar PFTs tend to cluster geographically, balancing the workload across MPI processes requires that geographically distinct gridcells be assigned to each MPI process to improve load balance. In CLM, grid cells are distributed

to clumps in a segmented round robin approach with the expectation that each clump will contain grid cells from a variety of geographic locations. Clumps are then assigned in a round-robin fashion to the processes. This decomposition strategy provides performance portability across a wide range of computer architectures.

Parallel Ocean Program

POP is a descendant of the Bryan-Cox-Semtner class of models [1]. It was developed at Los Alamos National Laboratory (LANL) in 1992, designed specifically to take advantage of high-performance computer architectures. POP was written initially in a data parallel formulation using CM Fortran and targeting the Thinking Machines CM-2 and CM-5 systems, but has since moved to a traditional message-passing implementation.

POP approximates the three-dimensional primitive equations for fluid motions on a generalized orthogonal computational grid on the sphere. Each timestep of the model is split into two phases. A three-dimensional “baroclinic” phase uses an explicit time integration

method. A “barotropic” phase includes an implicit solution of the two-dimensional surface pressure using a preconditioned conjugate gradient solver.

The parallel implementation is based on a two-dimensional tensor-product “block” decomposition of the horizontal dimensions of the three-dimensional computational grid. Blocks are then distributed to MPI processes. The vertical dimension is not decomposed. The amount of work associated with a block is proportional to the number of grid cells located in the ocean. Grid cells located over land are “masked” and eliminated from the computational loops. OpenMP parallelism is applied to loops over blocks assigned to an MPI process. The number of MPI processes and OpenMP threads to be used are specified at compile-time. This information is used to generate enough blocks so that all computational threads are assigned work.

The parallel implementation of the baroclinic phase requires only limited nearest-neighbor MPI communication (for halo updates) and performance is dominated primarily by computation. The barotropic phase requires both halo updates and global sums (implemented with local sums and a call to MPI_Allreduce for a small number of scalars) for each iteration of the conjugate gradient algorithm. The solution of the implicit system can require hundreds of iterations, and parallel performance of the barotropic phase is dominated by the communication cost of the halo updates and global sum operations.

Two different approaches to domain decomposition are supported currently: “cartesian” and “spacecurve.” The cartesian option decomposes the grid onto a two-dimensional virtual processor grid, and then further subdivides the local subgrids into blocks to provide work for OpenMP threads. This generates an efficient decomposition for halo communication. The cartesian option tends to be most efficient when using one block per thread. The spacecurve option begins by eliminating blocks having only “land” grid cells. A space-filling curve ordering of the remaining blocks is then calculated, and an equipartition of this one-dimensional ordering of the blocks is used to assign blocks to processes. The land block elimination step tends to improve load balance compared to the cartesian option. This is most effective if the blocks are small, which can lead to assigning more than one block per thread. However, the spacecurve distribution, with potentially

smaller blocks and generally more complex communication patterns, can incur higher communication cost in halo updates. For any given POP resolution, the optimal block size and decomposition strategy depends on the computer architecture and the processor count.

Community Ice Code

The CICE sea ice model was developed at LANL in the mid-1990s, with version 2 released to the public in 1999. It was designed to be integrated into a coupled climate model, and, in particular, to be compatible with POP. Version 4 was selected to be the sea ice component in CCSM4, replacing the Community Sea Ice Model (CSIM) that was used in CCSM3. However CSIM is closely related to CICE and much of the following discussion applies to CSIM as well.

The CICE sea ice model is formulated on a two-dimensional horizontal grid representing the earth’s surface. An orthogonal vertical dimension exists to represent the sea ice thickness. Similar to POP, the parallel implementation decomposes the horizontal dimensions into equal-sized two-dimensional blocks that are then assigned to processes. The blocks can be distributed using relatively arbitrary algorithms. The vertical dimension is not decomposed.

CICE supports both distributed-memory and shared-memory parallelism over the same dimension, namely grid blocks. The primary interprocess communication operation is a halo update. No global sums are required in the prognostic calculation, but global sums are used when calculating diagnostics. Currently the CICE decomposition is static and set at initialization. The performance of the CICE model is highly dependent on both the block size and the decomposition.

The relative cost of computing on the sea ice grid varies significantly both spatially and temporally over a climate simulation because the sea ice distribution is changing constantly. It varies most dramatically on seasonal timescales but can also vary on interannual timescales. This has a huge impact on the load balance of the sea ice model in a statically decomposed model. The load balance will be generally optimized if grid cells from varied geographical locations (high latitude / tropics, northern / southern hemisphere) are assigned to each process. This is similar to the CLM and CAM physics decomposition load balance issues.

In addition, CICE performs regular and frequent halo updates with a resultant performance cost that depends on the amount of data communicated and the number of messages sent. Halo updates are written such that all data communicated between two processes are aggregated into a single message, thereby minimizing the total number of messages needed to carry out a halo update for more unusual decompositions. CICE halo communication can be minimized by assigning neighbor blocks to the same process, minimizing edge lengths, and minimizing the number of neighbors and, hence, messages communicated.

As in POP, the load balance and halo cost compete for optimal static load balance. At lower processor counts, load balance is more important. As processor count is increased, halo cost becomes increasingly important. As a compromise, strips of neighboring gridcells that span relatively large swaths of latitude tend to be grouped into blocks using a simple two-dimensional decomposition. At higher processor count, this tends to result in relatively long, skinny blocks on the physical grid and relatively high cell aspect ratios for communication. The optimal decomposition for any hardware and resolution depends strongly on the processor count. Weighted space-filling curves and other decompositions have been implemented in CICE and are being explored as a means to improve performance at higher resolution. In addition, other techniques such as dynamic load balancing and improved overlap of work and interprocess communication are being explored to improve performance and scalability.

Coupler

The CCSM coupler is responsible for several actions including rearranging data between different process sets, interpolating (mapping) data between different grids, merging data from different components, flux calculations, and diagnostics. Many of the algorithms are trivially parallel and require no communication between grid cells.

The first CCSM couplers of the late 1990s were run on a single processor and communication between components was implemented using PVM, the Parallel Virtual Machine. Eventually, PVM was replaced by MPI and the coupler was parallelized. The first coupler parallelized for MPI distributed-memory parallelism was released with CCSM3 in 2004 and was based

largely on the Model Coupling Toolkit (MCT) Library developed at Argonne National Laboratory. The coupler released as part of CCSM4 in 2010 underwent significant architectural revisions, but still relied heavily on the MCT library to support decompositions and parallel operations.

The coupler receives grid information in parallel at runtime from all of the model components. Domain decompositions are determined on the fly based upon the model resolutions, the component model decompositions, and the processors used by the coupler. The coupler treats the grids as one-dimensional arrays of grid points so relatively arbitrary decompositions can be implemented to optimize performance. The coupler does not require information about grid point connectivity within each domain. Work is either completely local or data is referenced by global grid indices for non local operations, as specified by input files.

Both rearrangement and mapping require inter-process communication. Rearrangement is handled by MCT “routers” that define the required message exchanges. Routers are created at model initialization and reused throughout the model run. All required data communications between a pair of processes are aggregated into a single message exchange, and data are rearranged using a minimal number of messages.

Mapping of fields between two grids is done in parallel using the MCT rearrangement logic described above. The required mapping weights are precomputed, read in during initialization, and decomposed onto the processes based upon two possible mapping implementations. The first implementation assigns weights defining the mapping from the source grid to the destination grid using the decomposition of the source grid. Partial sums computed on the source grid decomposition are then rearranged to the destination grid decomposition, where final sums are computed. The second implementation rearranges the data from the source grid onto the destination grid decomposition. The mapping of the source grid data to the destination grid is then computed using the destination grid decomposition. The number of floating point operations are nearly identical in both cases. The difference is whether data are mapped primarily using the source or destination grid decomposition. The most efficient mapping method depends mostly on the relative sizes of the source and destination grids.

Parallel I/O

An efficient parallel I/O subsystem is a critical component of a parallel application code. Limiting external storage accesses to a single master process creates a serial bottleneck, degrading parallel performance and scalability of the application as a whole, and/or exhausting local memory. Allowing all processes to access the external storage, especially access to the same file, can lead to failure or very poor performance when thousands (or hundreds of thousands) of processes are involved. To address this need for CCSM, a new parallel I/O library called PIO has been developed and included in CCSM4.

PIO was initially designed to allow better memory management for very high resolution simulations, by relaxing the requirement for retaining the memory corresponding to the global two-dimensional horizontal resolution on the master I/O task. Since then, PIO has developed into a general purpose parallel I/O library that serves as a software interface layer designed to encapsulate the complexities of parallel I/O and to make it easier to replace the lower level software backend. PIO has been implemented throughout the entire CCSM system and currently supports serial I/O using netCDF and parallel I/O using pnetCDF.

PIO calls are collective. An MPI communicator is set in a call to the PIO initialization routine and all tasks associated with that communicator must participate in all subsequent calls to PIO. One of the key features of PIO is that it takes the model's decomposition and redistributes it to an "I/O-friendly" decomposition on the requested number of I/O tasks. In using the PIO library, the user must specify the number of I/O tasks to be used, the stride or number of tasks between I/O tasks and whether the I/O will use the serial netCDF or pnetCDF library. By increasing the number of I/O tasks, the user can easily reduce the serial I/O memory bottleneck even with the use of serial netCDF.

Community Earth System Model

Version 1 of the CESM, the Community Earth System Model, was released in June, 2010. The CESM is a superset of CCSM4 in that it can be configured to run the same science scenarios as CCSM4. However, the CESM also contains options for a terrestrial carbon cycle and dynamic vegetation, atmospheric chemistry

and aerosol dynamics, and ocean ecosystems and biogeochemical coupling, all necessary for an earth system model, as distinct from a purely physical model like CCSM4. While enabling the new CESM model options will change the performance characteristics of a simulation, increasing the computational cost and amount of I/O primarily, the prior discussion of the parallel algorithms in the component models and in the coupled model apply to the CESM virtually unchanged.

Related Entries

- ▶ [Collective Communication](#)
- ▶ [Computational Sciences](#)
- ▶ [Distributed-Memory Multiprocessor](#)
- ▶ [Fortran 90 and Its Successors](#)
- ▶ [I/O](#)
- ▶ [Load Balancing, Distributed Memory](#)
- ▶ [Loop Nest Parallelization](#)
- ▶ [MPI \(Message Passing Interface\)](#)
- ▶ [NetCDF I/O Library, Parallel](#)
- ▶ [OpenMP](#)
- ▶ [Space-Filling Curves](#)
- ▶ [Shared-Memory Multiprocessors](#)

Bibliographic Notes and Further Reading

Washington and Parkinson [12] is a comprehensive introduction to climate modeling. The 2003 DOE report "A Science-Based Case for Large-Scale Simulation" [10] includes estimates of the computational requirements for continued progress in computational climate science, arguing for the necessity of parallel simulation models targeting the highest performing computing platforms. A more recent discussion of the need for high performance computing in computational climate science is available in [11].

Detailed information on CCSM and CESM is available at the URLs

<http://www.cesm.ucar.edu/models/ccsm4.0/> and

<http://www.cesm.ucar.edu/models/cesm1.0/>

respectively. For descriptions of the climate science capabilities of CCSM3, see Collins et al. [2]. Papers documenting CCSM4 and CESM1 are in preparation, and will be cited at the above URLs when available.

The parallel implementation and performance characteristics of CCSM and of the component models have been documented in numerous journal and proceedings articles. Early work was reported in the proceedings of a series of conferences on the use of parallel processing in meteorology that were held at the European Center for Medium Range Weather Forecasting beginning in the mid-1980s, in particular see [6, 7]. An overview of many elements of this early work is available in a 1995 special issue of *Parallel Computing* on parallel computing in climate and weather modeling [4].

A description of more current aspects of the software engineering of CCSM component models is contained in a 2005 special issue of the *International Journal of High Performance Computing Applications* on climate modeling [5]. A recent overview of the software design of CCSM as a whole appeared in [3].

Acknowledgments

This work has been coauthored by a contractor of the US government under contract No. DE-AC05-00OR22725, and was partially sponsored by the Climate and Environmental Sciences Division of the Office of Biological and Environmental Research and by the Office of Advanced Scientific Computing Research, both in the Office of Science, US Department of Energy, under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC. Accordingly, the US government retains a nonexclusive, royalty free license to publish or reproduce the published form of this contribution, or allow others to do so, for US government purposes. This work used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725, and of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the US Department of Energy under contract DE-AC02-06CH11357.

Bibliography

1. Semtner JA (1986) Finite-difference formulation of a world ocean model. In: O'Brien JJ (ed) Advanced physical oceanographic numerical modeling. NATO ASI Series. Reidel, Norwell, pp 187–202
2. Collins WD, Bitz CM, Blackmon ML, Bonan GB, Bretherton CS, Carton JA, Chang P, Doney SC, Hack JH, Henderson TB, Kiehl JT, Large WG, McKenna DS, Santer BD, Smith RD (2006) The community climate system model version 3 (CCSM3). *J Clim* 19:2122–2143
3. Drake J, Jones P, Vertenstein M, White J III, Worley P (2008) Software design for petascale climate science. In: Bader D (ed) Petascale computing: algorithms and applications. Chapman & Hall/CRC Press, New York, pp 125–146, chap 7
4. Drake JB, Foster IT (1995) Special issue on parallel computing in climate and weather modeling. *Parallel Comput* 21:1537–1724
5. Drake JB, Jones PW, Carr G (2005) Special issue on climate modeling. *Int J High Perform Comput Appl* 19:175–350
6. Hoffman G-R, Kauranne T (eds) (1993) Parallel supercomputing in atmospheric science. In: Proceedings of the fifth ECMWF workshop on use of parallel processors in meteorology. World Scientific, Singapore
7. Hoffman G-R, Kreitz N (eds) (1999) Making its mark – the use of parallel processors in meteorology. In: Proceedings of the seventh ECMWF workshop on use of parallel processors in meteorology. World Scientific, Singapore
8. Lin S-J (2004) A ‘vertically lagrangian’ finite-volume dynamical core for global models. *Mon Weather Rev* 132:2293–2307
9. Neale RB, Chen C-C et al (2010) Description of the NCAR Community Atmosphere Model (CAM 5.0), NCAR Tech Note NCAR/TN-???+STR, June 2010. National Center for Atmospheric Research, Boulder
10. Office of Science US, Department of Energy (2003). A science-based case for large-scale simulation. <http://www.pnl.gov/scales/>. Accessed 30 July 2003
11. Washington W, Bader D, Collins W, Drake J, Taylor M, Kirtman B, Williams D, Middleton D (2009) Scientific grand challenges: challenges in climate change science and the role of computing at the extreme scale, Tech. Rep. PNNL-18362. Pacific Northwest National Laboratory. <http://www.science.doe.gov/ascr/ProgramDocuments/Docs/ClimateReport.pdf>
12. Washington W, Parkinson C (2005) An introduction to three-dimensional climate modeling, 2nd edn. University Science Books, Sausalito

Community Climate System Model (CCSM)

► [Community Climate System Model](#)

Community Earth System Model (CESM)

► [Community Climate System Model](#)

Community Ice Code (CICE)

- ▶ [Community Climate System Model](#)

Community Land Model (CLM)

- ▶ [Community Climate System Model](#)

Compiler Optimizations for Array Languages

- ▶ [Array Languages, Compiler Techniques for](#)

Compilers

- ▶ [Array Languages, Compiler Techniques for](#)
- ▶ [Banerjee's Dependence Test](#)
- ▶ [Code Generation](#)
- ▶ [Dependence Abstractions](#)
- ▶ [Dependences](#)
- ▶ [Loop Nest Parallelization](#)
- ▶ [Modulo Scheduling and Loop Pipelining](#)
- ▶ [Omega Test](#)
- ▶ [Polyhedron Model](#)
- ▶ [Parallelization, Automatic](#)
- ▶ [Parallelization, Basic Block](#)
- ▶ [Parallelism Detection in Nested Loops, Optimal](#)
- ▶ [Parafraze](#)
- ▶ [Polaris](#)
- ▶ [R-Stream Compiler](#)
- ▶ [Speculative Parallelization of Loops](#)
- ▶ [Tiling](#)
- ▶ [Trace Scheduling](#)
- ▶ [Unimodular Transformations](#)

Complete Exchange

- ▶ [All-to-All](#)

Complex Event Processing

- ▶ [Stream Programming Languages](#)

Computational Biology

- ▶ [Bioinformatics](#)

Computational Chemistry

- ▶ [NWChem](#)

Computational Models

- ▶ [Models of Computation, Theoretical](#)

Computational Sciences

GEOFFREY FOX
Indiana University, Bloomington, IN, USA

Synonyms

[Applications and parallelism; Problem architectures](#)

Definition

Here it is asked which applications should run in parallel and correspondingly which areas of computational science will benefit from parallelism. In studying this it will be discovered which applications benefit from particular hardware and software choices. A driving principle is that in parallel programming, one must map problems into software and then into hardware. The architecture differences in source and target of these maps will affect the efficiency and ease of parallelism.

Discussion

Introduction

I have an application – can and should it be implemented on a parallel architecture and if so, how should

this be done and what are appropriate target hardware architectures, what is known about clever algorithms and what are recommended software technologies? Fox introduced in [1] a general approach to this question by considering problems and the computer infrastructure on which they are executed as complex systems. Namely each is a collection of entities and connections between them governed by some laws. The entities can be illustrated by mesh points, particles, and data points for problems; cores, networks, and storage locations for hardware; objects, instructions, and messages for software. The processes of deriving numerical models, generating the software to simulate model, compiling the software, generating the machine code, and finally executing the program on particular hardware can be considered as maps between different complex systems. Many performance models and analyses have been developed and these describe the quality of map. It is known that maps are essentially never perfect and describing principles for quantifying this is a goal of this entry. At a high level, it is understood that the architecture of problem and hardware/software must match; given this we have quantitative conditions that the performance of the parts of the hardware must be consistent with the problem. For example, if two mesh points in problem are strongly connected, then bandwidth between components of hardware to which they are mapped must be high. In this discussion, the issues of parallelism are being described and here there are two particularly interesting general results. Firstly a space (the domain of entities) and a time associated with a complex system are usually defined. Time is nature's time for the complex system that describes time dependent simulations. However, for linear algebra, time for that complex system is an iteration count. Note that for the simplest sequential computer hardware there is no space and just a time defined by the control flow. Thus in executing problems on computers one is typically mapping all or part of the space of the problem onto time for the computer and parallel computing corresponding case where both problem and computer have well defined spatial extent. Mapping is usually never 1:1 and reversible, and "information is lost" as one maps one system into another. In particular, one fundamental reason why automatic parallelism can be hard is that the mapping of problem into software has thrown away key information about the space-time structure of original problem. Language designers in this field try to

find languages that preserve key information needed for parallelism while hardware designers design computers that can work around this loss of information. For an example, use of arrays in many data parallel languages from APL, HPF, to Sawzall can be viewed as a way to preserve spatial structure of problems when expressed in these languages. In this article, these issues will not be discussed in depth but rather it will be discussed what is possible with "knowledgeable users" mapping problems to computers or particular programming paradigms.

Simple Example

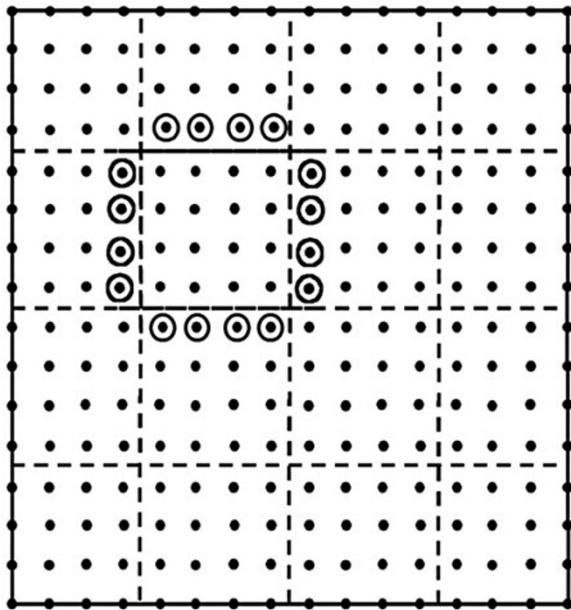
The simple case of a problem whose complex system spatial structure is represented as a 2D mesh is considered. This comes in material science when one considers local forces between a regular array of particles or in the finite difference approach to solving Laplace or Poisson's equation in two dimensions. There are many important subtleties such as adaptive meshes and hierarchical multigrid methods but in the simplest formulation such problems are set up as a regular grid of field values where the basic iterative update links nearest neighbors in two dimensions.

If the points are labeled by an index pair (i, j) , then Jacobi's method (not state of the art but chosen as simplicity allows a clear discussion) can be written

$$\phi(i, j) \text{ is replaced by } (\phi_{\text{Left}} + \phi_{\text{Right}} + \phi_{\text{Up}} + \phi_{\text{Down}})/4 \quad (1)$$

where $\phi_{\text{Left}} = \phi(i, j - 1)$ and similarly for ϕ_{Right} , ϕ_{Up} , and ϕ_{Down} .

Such problems would usually be implemented on a parallel architecture by a technique that is often called "domain decomposition" or "data parallelism" although these terms are not very precise. Parallelism is naturally found for such problems by dividing the domain up into parts and assigning each part to a different processors as seen in Fig. 1. Here the problem represented as a 16×16 mesh is solved on a 4×4 mesh of processors. For problems coming from nature this geometric view is intuitive as say in a weather simulation, the atmosphere over California evolves independently from that over Indiana and so can be simulated on separate processors. This is only true for short time extrapolations – eventually information flows between these sites and



Computational Sciences. Fig. 1 Communication structure for 2D complex system example. The dots are the 256 points in the problem. Shown by dashed lines is the division into 16 processors. The circled points are the halo or ghost grid points communicated to processor they surround

their dynamics are mixed. Of course it is the communication of data between the processors (either directly in a distributed memory or implicitly in a shared memory) that implements this eventual mixing.

Such block data decompositions typically lead to a SPMD (Single Program Multiple Data) structure with each processor executing the same code but on different data points and with differing boundary conditions. In this type of problem, processors at the edge of the (4×4) mesh do not see quite the same communication and compute complexity as the “general case” of an inside processor shown in Fig. 1. For the local nearest neighbor structure of Eq. 1, one needs to communicate the ring of halo points shown in figure. As computation grows like the number of points (grain size) n in each processor and communication like the number on edge (proportional to \sqrt{n}), the time “wasted” communicating decreases as a fraction of the total as the grain size n increases. Further one can usually “block” the communication to transmit all the needed points in

a few messages as latency can be an important part of communication overhead.

Note that this type of data decomposition implies the so-called “owner’s-compute” rule. Here each data point is imagined as being owned by the processor to which the decomposition assigns it. The owner of a given data-point is then responsible for performing the computation that “updates” its corresponding data values. This produces a common scenario where parallel program consists of a loop over iterations divided into compute-communicate phases:

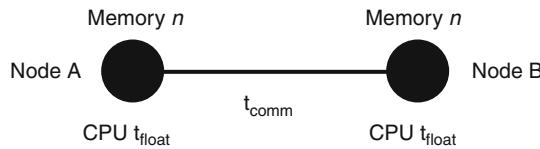
- *Communicate*: At the start of each iteration, first communicate any outside data values needed to update the data values at points owned by this processor.
- *Compute*: Perform update of data values with each processor operating without need to further synchronize with other machines.

This general structure is preserved even in many complex physical simulations with fixed albeit irregular decompositions. Dynamic decompositions introduce a further step where data values are migrated between processors to ensure load balance but this is usually still followed by similar communicate-compute phases. The communication phase naturally synchronizes the operation of the parallel processors and provides an efficient barrier point which naturally scales. The above discussion uses a terminology natural for distributed memory hardware or message passing programming models. With a shared memory model like OpenMP, communication would be implicit and the “communication phase” above would be implemented as a barrier synchronization.

Performance Model

Our current Poisson equation example can be used to illustrate some simple techniques that allow estimates of the performance of many parallel programs.

As shown in Fig. 2, the node of a parallel machine is characterized by a parameter t_{float} , which is time taken for a single floating point operation. t_{float} is of course not very well defined as depends on the effectiveness of cache, possible use of fused multiply-add and other issues. This implies that this measure will have some application dependence reflecting the goodness of the



Computational Sciences. Fig. 2 Parameters determining performance of loosely synchronous problems

match of the problem to the node architecture. We let n be the grain size – the number of data points owned by a typical processor. Communication performance – whether through a shared or distributed memory architecture – can be parameterized as

$$\begin{aligned} & \text{Time to communicate } N_{\text{comm}} \text{ words} \\ &= t_{\text{latency}} + N_{\text{comm}} \cdot t_{\text{comm}} \end{aligned} \quad (2)$$

This equation ignores issues like bus or switch contention but is a reasonable model in many cases. Latencies t_{latency} can be around $1\mu\text{s}$ on high performance systems but is measured in milliseconds in a geographically distributed grid. t_{comm} is time to communicate a single word and for large enough messages, the latency term can be ignored which will be done in the following.

Parallel performance is dependent on load balancing and communication and both can be discussed but here it is focused on communication with problem of Fig. 1 generalized to N_{proc} processors arranged in an $\sqrt{N_{\text{proc}}}$ by $\sqrt{N_{\text{proc}}}$ grid with a total of N mesh points and the grain size $n = N/N_{\text{proc}}$. Let $T(N_{\text{proc}})$ be the execution time on N_{proc} processors and two contributions are found to this ignoring small load imbalances from edge processors. There is a calculation time expressed as $n \cdot t_{\text{calc}}$ with $t_{\text{calc}} = 4t_{\text{float}}$ as the time to execute the basic update Eq. 1. In addition the parallel program has communication overhead, which adds to $T(N_{\text{proc}})$ a term $4\sqrt{n} \cdot t_{\text{comm}}$. Now the speed up formula is found:

$$\begin{aligned} S(N_{\text{proc}}) &= T(1)/T(N_{\text{proc}}) \\ &= N_{\text{proc}}/(1 + t_{\text{comm}}/(\sqrt{n} \cdot t_{\text{float}})) \end{aligned} \quad (3)$$

It is noted that this analysis ignores the possibility available on some computers of overlapping communication and computation which is straightforwardly included. The above formalism can be generalized most conveniently using the notation that

$$S(N_{\text{proc}}) = \varepsilon \cdot N_{\text{proc}} = N_{\text{proc}}/(1 + f), \quad (4)$$

which defines efficiency ε and overhead f . Note that it is preferred to discuss overhead rather than speed-up or efficiency as one typically gets simpler models for f as the effects of parallelism are additive to f but for example occur in the denominator of Eq. 3 for speedup and efficiency. The communication part f_{comm} of the overhead f is given by combining Eqs. 3 and 4 as

$$f_{\text{comm}} = t_{\text{comm}}/(\sqrt{n} \cdot t_{\text{float}}) \quad (5)$$

Note that in many instances, f_{comm} can be thought of as simply the ratio of parallel communication to parallel computation. This equation can be generalized to essentially all problems we will later term loosely synchronous. Then in each coupled communicate-compute phases of such problems, one finds that the overhead takes the form:

$$f_{\text{comm}} = \text{constant} \cdot t_{\text{comm}}/(n^{1/d} \cdot t_{\text{float}}) \quad (6)$$

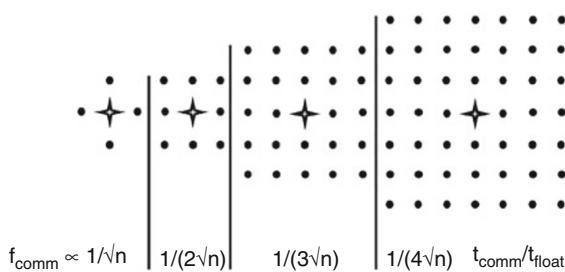
Here d is an appropriate (complexity or information) dimension, which is equal to the geometric dimension for partial differential based equations or other geometrically local algorithms such as particle dynamics. A particularly important case in practice is the 3D value $d = 3$ when $n^{-1/d}$ is just surface/volume in three dimensions. However Eq. 6 describes many non geometrically local problems with for example the value $d = 2$ for the best decompositions for full matrix linear algebra and $d = 1$ for long range interaction problems. The Fast Fourier Transform FFT finds $n^{1/d}$ in Eq. 6 replaced by $\ln(n)$ corresponding to $d = \infty$.

From Eq. 4, it can be found that $S(N_{\text{proc}})$ increases linearly with N_{proc} as long as N_{proc} is increased with fixed f_{comm} which implies fixed grain size n , while t_{comm} and t_{float} are naturally fixed. This is scaled speedup where the problem size $N = n \cdot N_{\text{proc}}$ also increases linearly with N_{proc} . The continuing success of parallel computing even on very large machines can be considered as a consequence of equations like Eq. 6 as the formula for f_{comm} only depends on local node parameters and not on the number of processors. Thus as we scale up the number of processors keeping the node hardware and application grain size n fixed, we will get scaling performance – speedup proportional to N_{proc} . Note this implies that total problem size increases proportional to N_{proc} – the defining characteristic of scaled speedup.

Complex Applications Are Better for Parallelism

The simple problem described above is perhaps the one where the parallel issues are most obvious; however it is not the one where good parallel performance is easiest to obtain as the small computational complexity of the update Eq. 1 makes the communication overhead relatively more important. There is a fortunate general rule that as one increases the complexity of a problem, the computation needed grows faster than the communication overhead and this will be illustrated below. Jacobi iteration does have perhaps the smallest communication for problems of this class. However it has one of largest ratios of communication to computation and correspondingly high parallel overhead. Note one sees the same effect on a hierarchical (cache) memory machine, where problems such as Jacobi Iteration for simple equations can perform poorly as the number of operations performed on each word fetched into cache is proportional to number of links per entity and this is small (four in the 2-D mesh considered above) for this problem class.

As an illustration of the effect varying computational complexity, it can be seen in Fig. 3 how the above analysis is altered as one changes the update formula of Eq. 1. The size of the stencil parameterized can now be systematically increased by an integer l and it can be found how f_{comm} changes. In the case where points are particles the value of l corresponds to the range of their mutual force and in the case of discretization of partial differential equations l measures the order of the approximation.



Computational Sciences. Fig. 3 Communication structure as a function of stencil size. The figure shows 4 stencils with from left to right, range $l = 1, 2, 3$

The communication overhead is found to decrease systematically as shown in Fig. 3 as the range of the force increases. The general 2D result is:

$$f_{\text{comm}} \propto t_{\text{comm}} / (l \cdot \sqrt{n} \cdot t_{\text{float}}) \quad (7)$$

This is valid for l which is large compared to 1 but smaller than the length scale corresponding to region stored in each processor. In the interesting limit of an infinite range ($l \rightarrow \infty$) force, the analysis needs to be redone and one finds the result that is independent of the geometric dimension

$$f_{\text{comm}} \propto t_{\text{comm}} / (n \cdot t_{\text{float}}) \quad (8)$$

which is of the general form of Eq. 6 with complexity dimension $d = 1$. This is the best-understood case where the geometric and complexity dimensions are different. The overhead formula of Eq. 8 corresponds to the computationally intense $O(N^2)$ algorithms for evolving N-body problems. The amount of computation is so large that the ratio of communication to computation is extremely small.

Application Architectures

The analysis above can be applied to many SPMD problems and addresses the matching of “spatial” structure of applications and computers. This drives needed linkage of individual computers in a parallel system in terms of topology and performance of network. However this only works if we can match the temporal structure and this aspect is more qualitative and perhaps controversial. The simplest ideas here underlined the early SIMD (Single Instruction Multiple Data) machines that were popular some 20 years ago. These are suitable for problems where each point of the complex system evolves with the same rule (mapping into machine instruction) at each time. There are many such problems including for example the Laplace solver discussed above. However many related problems do not fit this structure – called *synchronous* in [1] – with the simplest reason being heterogeneity in system requiring different computational approaches at different points. A huge number of scientific problems fit a more general classification – *loosely synchronous*.

Here SPMD applications can be seen which have the compute-communication stages described above but now the compute phases are different on different processors. One uses load balancing methods to ensure that the computational work on each node is balanced but not on each machine instruction but rather in a coarse grain fashion at every iteration or time-step – whatever defines the temporal evolution of the complete system. Loosely synchronous problems fit naturally MIMD machines with the communication stages at macroscopic “time-steps” of the application. This communication ensures the overall correct synchronization of the parallel application. Thus overhead formulae like Eqs. 5 and 6 describe both communication and synchronization overhead. As this overhead only depends on local parameters of the application, it is understood why loosely synchronously can get good scalable performance on the largest supercomputers. Such applications need no expensive global synchronization steps. Essentially all linear algebra, particle dynamics and differential equation solvers fall in the loosely synchronous class. Note synchronous problems are still around but they are run on MIMD (Multiple Instruction Multiple Data) machines with the SPMD model.

A third class of problems – termed *asynchronous* – consists of asynchronously interacting objects and is often people’s view of a typical parallel problem. It probably does describe the concurrent threads in a modern operating system and some important applications such as event driven simulations and areas like search in computer games and graph algorithms. Shared memory is natural for asynchronous problems due to low latency often needed to perform dynamic synchronization. It wasn’t clear in the past but now it appears this category is not very common in large scale parallel problems of importance. The surprise of some at the practical success of parallel computing can perhaps be understood from people thinking about asynchronous problems whereas its loosely synchronous and *pleasantly parallel* problems that dominate. The latter class is the simplest algorithmically with disconnected parallel components. However the importance of this category has probably grown since the original 1988 analysis [2] when it was estimated as 20% of all parallel computing. Both Grids and clouds are very natural for this class

which does not need high performance communication between different nodes. Parameter searches and many data analysis applications of independent observations fall into this class.

From the start, we have seen a fifth class – termed metaproblems – which refer to the coarse grain linkage of different “atomic” problems. Here synchronous, loosely synchronous, asynchronous and pleasingly parallel are the atomic classes. Metaproblems are very common and expected to grow in importance. One often uses a two level programming model in this case with the metaproblem linkage specified by workflow and the component problems with traditional parallel languages and runtimes. Grids or Clouds are suitable for metaproblems as coarse grain decomposition does not usually require stringent performance.

These five categories are summarized in Table 1 which also introduces a new category *MapReduce++* which has recently grown in importance to described data analysis. Nearly all the early work on parallel computing focused on simulation as opposed to data analysis (or what some call data intensive applications). Data analysis has exploded in importance recently [3] correspondingly to growth in number of instruments, sensors and human (the web) sources of data.

Summary

Problems are set up as computational or numerical systems and these can be considered as a “space” of linked entities evolving in time. The spatial structure (which is critical for performance) and the temporal structure which is critical to understand the class of software and computer needed were discussed. These were termed “basic complex systems” and characterized by their possibly dynamic spatial (geometric) and temporal structure. The difference between the structure of the original problem and that of computational system derived from it have been noted. Much of the past experience can be summarized in parallelizing applications by the conclusion:

Synchronous and Loosely Synchronous problems perform well on large parallel machines as long as the problem is large enough. For a given machine, there is

Computational Sciences. Table 1 Application classification

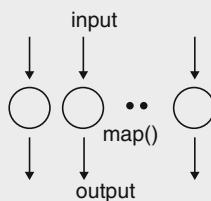
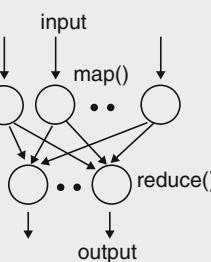
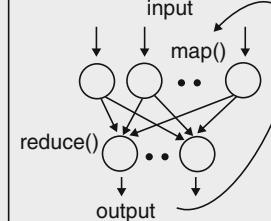
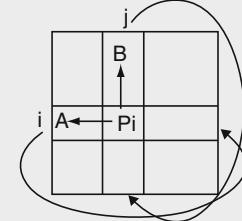
#	Class	Description	Machine Architecture
1	Synchronous	The problem class can be implemented with instruction level Lockstep Operation as in SIMD architectures	SIMD
2	Loosely Synchronous (or BSP Bulk Synchronous Processing)	These problems exhibit iterative Compute-Communication stages with independent compute (map) operations for each CPU that are synchronized with a communication step. This problem class covers many successful MPI applications including partial differential equation solution and particle dynamics applications.	MIMD on MPP (Massively Parallel Processor)
3	Asynchronous	Illustrated by Compute Chess and Integer Programming; Combinatorial Search often supported by dynamic threads. This is rarely important in scientific computing but at heart of operating systems and concurrency in consumer applications such as Microsoft Word.	Shared Memory
4	Pleasingly Parallel	Each component is independent. In 1988, Fox estimated this at 20% of the total number of applications [2] but that percentage has grown with the use of Grids and data analysis applications including for example the Large Hadron Collider analysis for particle physics.	Grids moving to Clouds
5	Metaproblems	These are coarse grain (asynchronous or dataflow) combinations of classes (1–4) and (6). This area has also grown in importance and is well supported by Grids and described by workflow of Section 3.5.	Grids of Clusters
6	MapReduce++	It describes file(database) to file(database) operations which has three subcategories given below and in Table 2 . (6a) Pleasingly Parallel Map Only – similar to category 4 (6b) Map followed by reductions (6c) Iterative “Map followed by reductions” – Extension of Current Technologies that supports much linear algebra and data mining	Data-intensive Clouds (a) Master-Worker or MapReduce (b) MapReduce (c) Twister

The MapReduce++ category has three subdivisions (a) “map only” applications similar to pleasingly parallel category; (b) The classic MapReduce with file to file operations consisting of parallel maps followed by parallel reduce operations; (c) captures the extended MapReduce introduced in [4–10]. Note this category has the same complex system structure as loosely synchronous or pleasingly parallel problems but is distinguished by the reading and writing of data. This comparison is made clearer in [Table 2](#). Note nearly all early work on parallel computing discussed computing with data on memory. MapReduce and languages like Sawzall [11] and Pig-Latin [12] emphasize the parallel processing of data on disks – a field that until recently was only covered by database community

a typical sub-domain size (i.e. the grain size or size of that part of the problem stored on each node) above which one can expect to get good performance. There will be a roughly constant ratio of parallel speedup to N_{proc} if one scales the problem with fixed sub-domain

size and total size proportional to N_{proc} . This conclusion has been enriched by study of grids and clouds with an emphasize on pleasingly parallel and MapReduce++ style problems often with a data intensive focus. These also parallelize well.

Computational Sciences. Table 2 Comparison of MapReduce++ subcategories and Loosely Synchronous category

Map-only	Classic MapReduce	Iterative MapReduce	Loosely Synchronous
 <ul style="list-style-type: none"> • Document conversion (e.g. PDF->HTML) • Brute force searches in cryptography • Parametric sweeps • Gene assembly • Much data analysis of independent samples 	 <ul style="list-style-type: none"> • High Energy Physics (HEP) Histograms • Distributed search • Distributed sort • Information retrieval • Calculation of Pairwise Distances for sequences (BLAST) 	 <ul style="list-style-type: none"> • Expectation maximization algorithms • Linear Algebra • Datamining including • Clustering • K-means • Multidimensional Scaling (MDS) 	 <ul style="list-style-type: none"> • Many MPI scientific applications utilizing wide variety of communication constructs including local interactions • Solving differential equations and • Particle dynamics with short range forces
Domain of MapReduce and Iterative Extensions			

Bibliographic Notes and Further Reading

The approach followed here was developed in [1, 13] with further details in [2, 14]. The extension to include data intensive applications was given in [8, 15]. There are many good discussions of speedup including Gustafson's seminal work [16] and the lack of it – Amdahl's law [17]. The recent spate of papers on MapReduce [4, 7] and its applications and extensions [4–12, 15] allow one to extend the discussion of parallelism from simulation (which implicitly dominated the early work) to data analysis [3].

Bibliography

- Fox GC, Williams RD, Messina PC (1994) Parallel computing works!. Morgan Kaufmann, San Francisco. http://www.old-npac.org/copywrite/pew/node278.html#SECTION_0014400000000000000000
- Fox GC (1988) What have we learnt from using real parallel machines to solve real problems. In: Fox GC (ed) Third conference on hypercube concurrent computers and applications, vol. 2. ACM, New York, pp 897–955
- Gray J, Hey T, Tansley S, Tolle K (2010) The fourth paradigm: data-intensive scientific discovery. Accessed 21 Oct 2010. Available from: <http://research.microsoft.com/en-us/collaboration/fourthparadigm/>
- Ekanayake J, Li H, Zhang B, Gunarathne T, Bae S, Qiu J, Fox G (2010) Twister: a runtime for iterative MapReduce. In: Proceedings of the first international workshop on MapReduce and its applications of ACM HPDC 2010 conference. ACM, Chicago, 20–25 Jun 2010. <http://grids.ucs.indiana.edu/ptliupages/publications/hpdc-camera-ready-submission.pdf>
- Yingyi B, Howe B, Balazinska M, Ernst MD (2010) HaLoop: efficient iterative data processing on large clusters. In: The 36th international conference on very large data bases, VLDB Endowment, vol 3, Singapore, 13–17 Sept 2010. http://www.ics.uci.edu/~yingyb/papers/HaLoop_camera_ready.pdf
- Zhang B, Ruan Y, Tak-Lon W, Qiu J, Hughes A, Fox G (2010) Applying twister to scientific applications. In: CloudCom 2010. IUPUI Conference Center, Indianapolis, 30 Nov–3 Dec 2010. <http://grids.ucs.indiana.edu/ptliupages/publications/PID1510523.pdf>
- Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Commun ACM 51(1):107–113
- Ekanayake J (2010) Architecture and performance of runtime environments for data intensive scalable computing. Ph. D. thesis, School of Informatics and Computing, Indiana University, Bloomington, Dec 2010. http://grids.ucs.indiana.edu/ptliupages/publications/thesis_jaliya_v24.pdf
- Malewicz G, Austern MH, Bik AJC, Dehnert JC, Horn I, Leiser N, Czajkowski G (2010) Pregel: a system for large-scale graph processing. In: International conference on management of data, Indianapolis, pp 135–146
- Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I (2010) Spark: cluster computing with working sets. In: Second USENIX workshop on hot topics in cloud computing

- (HotCloud '10), Boston, 22 Jun 2010. <http://www.cs.berkeley.edu/~franklin/Papers/hotcloud.pdf>
11. Pike R, Dorward S, Griesemer R, Quinlan S (2005) Interpreting the data: parallel analysis with sawzall. *Sci Program J* (Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure) 13(4):227–298. <http://iospress.metapress.com/content/99VJKGKAE3JKVU9T>
 12. Olston C, Reed B, Srivastava U, Kumar R, Tomkins A (2008) Pig Latin: a not-so-foreign language for data processing. In: Proceedings of the 2008 ACM SIGMOD international conference on management of data. ACM, Vancouver, pp 1099–1110. <http://portal.acm.org/citation.cfm?id=1376726>
 13. Dongarra J, Foster I, Fox G, Gropp W, Kennedy K, Torczon L, White A (2002) The sourcebook of parallel computing. Morgan Kaufmann, San Francisco. ISBN:978-1558608719
 14. Fox GC, Coddington P (2000) Parallel computers and complex systems. In: Bossomaier TRJ, Green DG (eds) Complex systems: from biology to computation. Cambridge University Press, pp 272–287. <http://surface.syr.edu/npac/61/>
 15. Ekanayake J, Gunarathne T, Qiu J, Fox G, Beason S, Choi JY, Ruan Y, Bae SH, Li H (2010) Applicability of DryadLINQ to scientific applications. Community Grids Laboratory, Indiana University, 30 Jan 2010. <http://grids.ucs.indiana.edu/ptliupages/publications/DryadReport.pdf>
 16. Gustafson JL (1988) Reevaluating Amdahl's law. *Commun ACM* 31(5):532–533. doi:10.1145/42411.42415
 17. Wikipedia (2010) Amdahl's law. Accessed 28 Dec 2010. Available from: http://en.wikipedia.org/wiki/Amdahl's_law

Computer Graphics

JOHN C. HART

University of Illinois at Urbana-Champaign, Urbana, IL, USA

Definition

The field of computer graphics is concerned with developing algorithms for visual simulation, and has both benefitted from parallel computing, especially to achieve real-time rendering rates, and contributed to parallel computing, especially through streaming architectures and programming languages.

Discussion

The field of computer graphics embodies algorithms for generating and manipulating images, representing and constructing shapes, and simulating and capturing motion. These algorithms often lend themselves naturally to parallel approaches due to the regularity of its

data, e.g., image pixels, light rays, mesh elements, or moving particles.

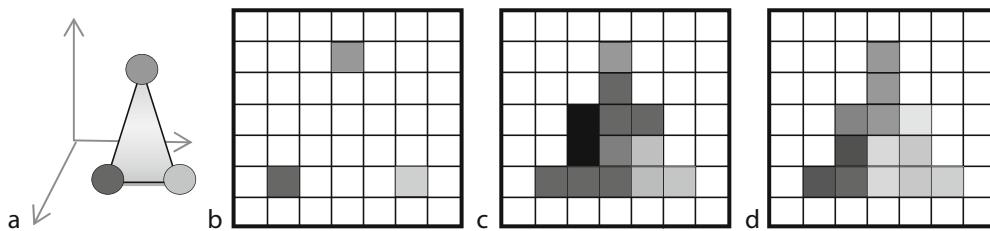
Many operations in computer graphics are specializations of more general numerical, scientific, and combinatorial algorithms that have been parallelized in more general settings. The distribution of light in a scene is a recurrent integration, often solved with Monte Carlo methods. Linear system solvers including iterative methods, conjugate gradients, and multigrid methods are used for meshing, rendering and animating 3-D objects and scenes, and a wide variety of powerful image editing operations. Combinatorial optimization algorithms like GraphCut can be used to merge texture elements. N-body solutions have been used to find a light equilibrium and for surface reconstruction from scattered point data. Hence parallel computer graphics is often simply finding the appropriate numerical or combinatorial method, usually from scientific computing, and parallelizing that algorithm for deployment on a graphics platform, sometimes capitalizing on the reduced precision and other approximations afforded to visual simulation.

Much of the remainder of parallel graphics work has focused on rendering, particularly maximizing the quality of rendering while maintaining a desired rate of image production (e.g., real-time). Rendering is the process of taking a computer representation of a scene of one or more shapes and constructing one or more images depicting that scene. Rendering can be photorealistic, based on a physical model of light, or expressive, based on a perceptual model of the viewer. Renderings can be of contrived scenes for entertainment, or of data for visualization, and can range from real time (faster than 30 images/s) to interactive (around 10 images/s) to offline (taking minutes, hours, and even days per image). In all cases, parallelism provides a toolbox for achieving greater image generation throughput.

Modern rendering falls into three categories: rasterization, ray tracing, and micropolygons, and the remainder of this discussion describes the parallelism used to accelerate these rendering algorithms to support real-time graphics.

Rasterization

Rasterization has been the primary rendering method for real-time graphics and parallelism is used to increase



Computer Graphics. Fig. 1 Vertices from a mesh polygon in a 3-D scene (a) are shaded and transformed independently in parallel into screen pixel locations (b). Screen pixels between these locations are interpolated by rasterization (c) and processed independently in parallel (d), in this example, saturating the color

the number and quality of rasterized polygons. To rasterize each polygon in a mesh, the polygon's vertices must first be shaded and transformed from scene coordinates to pixel locations. These operations are pipelined and lend themselves naturally to parallel processing since the vertices can be processed completely independently from each other (Fig. 1).

The actual rasterization process fills in the screen pixels between the pixel locations of the screen projected polygon vertices, interpolating any attributes (e.g., color) that accompany the vertices. These pixels are then further processed for texturing and fine-detail shading, which also can be trivially parallelized since the pixel computations are independent of each other.

This vertex and pixel parallelism has been implemented in a variety of graphics architectures over many decades, leading to the modern graphics processing unit (GPU). Since the computations are independent and identical over all vertices, and similarly over all pixels, single-instruction-multiple-data (SIMD) processing is utilized, and evolved into present day GPU single-process-multiple-data (SPMD) processing. For efficient processing GPU programs must still remain sensitive to SIMD performance issues, such as control flow divergence. Though we distinguish between vertex shaders and pixel shaders by their position in the graphics pipeline, the same processing array is used for both.

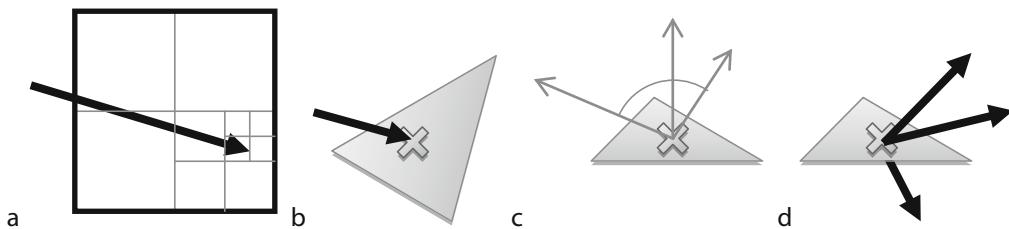
In order to support a wider variety of shading effects, these parallel vertex and pixel processes have supported custom programming, which led to the modern programmable GPU which can support programmable vertex and pixel “shaders.” For example, “skinning” is a vertex shader that smoothly deforms an elbow by interpolating at each elbow mesh vertex the rotated coordinate frame of the upper arm with the rotated

coordinate frame of the forearm. Pixel shaders were primarily motivated by the need to interpolate surface normals between vertices and compute shading per pixel, instead of computing shading per vertex and interpolating the vertex color across the polygon pixels.

Since these processes are independent and SIMD, their parallel processing is quite efficient, which made the GPU an attractive parallel computational platform for “general purpose” parallel programming, called GPGPU programming. As parallel graphics processing developed, general purpose programs (e.g., computing a Mandelbrot set or a Voronoi diagram) had to be formulated as multiple passes of polygon rendering programs to be accelerated by the GPU. Modern GPUs do not require disguising general-purpose applications as graphics rasterization tasks. They support a “compute” mode that disables rasterization and allows the SIMD processors to execute directly on input data, and can be programmed directly with general purpose parallel streaming languages, development environments, and libraries such as Nvidia’s CUDA, OpenCL, and Intel’s ArBB.

Ray Tracing

By leveraging the power of parallel processing enabled by the GPU and the multicore PC, ray tracing is becoming a viable alternative to rasterization for the real-time rendering needs of video games, virtual environments, and interactive visualization. Whereas rasterization finds for each polygon which pixels it covers, ray tracing finds for each pixel which polygons cover it (in both cases displaying for each pixel the closest polygon in the view direction). For each pixel, ray tracing constructs a line of sight from the viewer through it into the scene, and intersects that line of sight with



Computer Graphics. Fig. 2 Stages of tracing a ray. (a) Traversing a spatial data structure to find likely candidate elements for intersection. (b) Intersecting an element. (c) Shading the intersection. (d) Spawning secondary rays for reflection, refraction, and shadowing

the scene's geometry. For curved (non-polygonal) surfaces this intersection is a numerical root-finding solver. The intersection point can yield further ray tracing of shadow, reflection, refraction, and other rays to find further illumination from the light sources. This process can be trivially parallelized over the pixels for SIMD processing, but can lead to load imbalance since some pixels miss the scene entirely while others display complex illumination requiring many recursive ray intersections (Fig. 2).

Scenes are often complex, and require complex spatial data structures to efficiently determine which scene elements are intersected by which rays to avoid an expensive all-pairs ray scene intersection. Especially for meshed scenes, ray intersection becomes a spatial database query, using hierarchical or grid structures to organize the scene. When parallelized, this structure needs to be shared or distributed among the processors. When the scene is dynamic, then this structure needs to be updated or reconstructed, which itself can rely on parallel speedups.

The intersection of a pair of rays passing through neighboring pixels often share the same computation and can benefit from spatial coherence and accelerated vector processing. However, the rays they spawn to measure secondary illumination are rarely coherent and their computation can diverge precluding efficient vector processing. Some approaches seek to cache secondary rays into coherent bundles that can be intersected more efficiently, though the expense of the caching and reordering often outweighs the reward of the parallelism speedup.

The GPU, designed for parallel rasterization-based rendering, can be reprogrammed for parallel ray tracing. One approach created a pair of texture images

where each pixel corresponds to a single ray. One texture image's RGB values were the XYZ coordinates of the anchor (origin) of each ray, and a second texture image's RGB values were the XYZ coordinates of the ray directions. Then a single screen-filling quad would be rendered, whose vertices shared the same attribute data, in this case the nine values corresponding to the three coordinates of the three vertices of the triangle. As the screen-filling quad is rasterized, the attribute data is reproduced across each pixel, and when combined with the texture data provided each pixel's shader with access to the quad's triangle and the pixel's ray, allowing it to compute a ray intersection it could report back as the pixel's color and z-value. While it leverages full speed SIMD GPU computation of ray-triangle intersections, this approach requires a lot of CPU communication to coordinate spatial data structures, shading, and spawning new rays.

Ray tracing can be performed completely in SIMD on the GPU by treating the process as a state machine. The process of tracing a ray can be decomposed into four states: (1) traversing a spatial data structure to find which elements might intersect a ray, (2) intersecting the ray with elements, (3) shading the resulting intersection, and (4) spawning new rays to recursively determine illumination. Since the GPU is a SIMD processor (actually a collection of independent SIMD processors) each ray's process must wait until all of the other rays' processors finish the current state before any of them can move to the next state.

Micropolygons

Yet a third rendering approach, called micropolygon rendering, subdivides scene elements in 3-D into small "micropolygons" that each project to about the size of

a pixel and thus do not need to be rasterized. Micropolygons allow surface geometry to be displaced to more easily model embossed surface detail, and since each micropolygon projects to such a small pixel neighborhood, they can more easily be rendered in parallel.

Micropolygons led to the concept of the programmable “shader,” in an application specific little language called Renderman, that led to the vertex and pixel shaders of the modern GPU. The Renderman Shading Language succeeded because it contained a small collection of powerful high-level operators and data structures devoted specifically to the narrow task of illuminating and texturing surfaces. It inspired parallel GPU shader programming languages including Cg, GLSL, and HLSL, which are in comparison less elegant but sought to broaden the scope of GPU programming, eventually to CUDA, OpenCL, and other GPGPU programming frameworks and domain-specific languages for parallel programming.

Related Entries

- ▶ [NVIDIA GPU](#)
- ▶ [SPMD Computational Model](#)
- ▶ [Stream Programming Languages](#)

Bibliographical Notes and Further Reading

Applications of general scientific computing methods to specific problems in graphics can be found throughout the history of computer graphics literature. Some key examples of these are the Metropolis method for Monte Carlo sampling of light paths [1] and solving a symmetric positive definite matrix to find the equilibrium of light in a closed room [2]. Researchers have framed a number of graphics problems, such as seamlessly copying a portion of one image to another [3] or tiling a surface with quadrilaterals [4], as a Poisson partial differential equation, which opened a floodgate of “gradient domain” methods in graphics, solved largely interactively or in real-time by parallel matrix solvers. The GraphCut network optimization method can be used to stitch pixel swatches into an image [5]. Both radiosity [6] and radial basis function surface reconstruction [7] have been accelerated by treating them as N-body problems.

Planar Voronoi diagrams were computed with early parallel graphics hardware by depth-buffered rasterization of cones extending from the data points [8], and a Mandelbrot set could be computed by repeated image operations, called multipass programming [9].

Parallel ray tracing often suffers from memory management issues, especially when reorganizing rays into coherent bundles [10]. This plagued the ray engine which used the GPU to intersect rays and the CPU to manage ray coherence [11], while simpler geometry coherence structures, such as a grid, could be implemented on the GPU in a state-based system [12] suffered from excessive waiting that can be improved through better scheduling [13].

The micropolygon approach was pioneered by Pixar as the REYES (Render Everything You Ever Saw) architecture [14] that formed the basis for the parallel Pixar Image Computer, which could process each micropolygon independently in parallel. This lead to the development of programmable shaders and Renderman [15].

Bibliography

1. Veach E, Guibas LJ (1997) Metropolis Light Transport. Proceedings of SIGGRAPH, Los Angeles, pp 65–76
2. Goral CM, Torrance KE, Greenberg DP, Battaile B (1984) Modeling the interaction of light between diffuse surfaces. Proceedings of SIGGRAPH, Minneapolis, pp 213–222
3. Pérez P, Gangnet M, Blake A (2003) Poisson image editing. Proceedings of SIGGRAPH, San Diego, pp 313–318
4. Dong S, Bremer P-T, Garland M, Pascucci V, Hart JC (2006) Spectral surface quadrangulation. Proceedings of SIGGRAPH, pp 1057–1066
5. Kwatra V, Schödl A, Essa I, Turk G, Bobick A (2003) Graphcut textures: image and video synthesis using graph cuts. Proceedings of SIGGRAPH, San Diego, pp 277–286
6. Hanrahan P, Salzman D, Aupperle L (1991) A rapid hierarchical radiosity algorithm. Proceedings of SIGGRAPH, Las Vegas, pp 197–206
7. Carr JC, Beatson RK, Cherrie JB, Mitchell TJ, Fright WR, McCallum BC, Evans TR (2001) Reconstruction and representation of 3D objects with radial basis functions. Proceedings of SIGGRAPH, Los Angeles, pp 67–76
8. Hoff KE III, Zaferakis A, Lin M, Manocha D (2001) Fast and simple 2D geometric proximity queries using graphics hardware. Proceedings of Interactive 3D Graphics, pp 145–148
9. Peercy MS, Olano M, Airey J, Ungar PJ (2000) Interactive multipass programmable shading. Proceedings of SIGGRAPH, New Orleans, pp 425–432
10. Wald I, Slusallek P, Benthin C, Wagner M (2001) Interactive Rendering with coherent ray tracing. Proceedings of Eurographics, pp 277–288

11. Carr NA, Hall JD, Hart JC (2002) The ray engine. Proceedings of Graphics Hardware, pp 37–46
12. Purcell TJ, Buck I, Mark WR, Hanrahan P (2002) Ray tracing on programmable graphics hardware. Proceedings of SIGGRAPH, San Antonio, pp 703–712
13. Parker SG, Bigler J, Dietrich A, Friedrich H, Hoberock J, Luebke D, McAllister D, McGuire M, Morley K, Robison A, Stich M (2010) OptiX: a general purpose ray tracing engine. Proceedings of SIGGRAPH, Los Angeles, Article 66, 13 p
14. Cook RL, Carpenter L, Catmull E (1987) The Reyes image rendering architecture. Proceedings of SIGGRAPH, Anaheim, pp 95–102
15. Hanrahan P, Lawson J (1990) A language for shading and lighting calculations. Proceedings of SIGGRAPH, Dallas, pp 289–298

Computing Surface

► Meiko

Concatenation

► Allgather

Concurrency Control

► Path Expressions

Concurrent Collections Programming Model

MICHAEL G. BURKE¹, KATHLEEN KNOBE², RYAN NEWTON³, VIVEK SARKAR¹

¹Rice University, Houston, TX, USA

²Intel Corporation, Cambridge, MA, USA

³Intel Corporation, Hudson, MA, USA

Synonyms

CnC, TStreams

Definition

Concurrent Collections (CnC) is a parallel programming model, with an execution semantics that is influenced by dynamic dataflow, stream-processing, and

tuple spaces. The three main constructs in the CnC programming model are *step collections*, *data collections*, and *control collections*. A step collection corresponds to a computation, and its instances correspond to invocations of that computation that consume and produce data items. A data collection corresponds to a set of *data items*, indexed by *item tags* that can be accessed via *put* and *get* operations; once put, data items cannot be overwritten and are required to be *immutable*. A control collection corresponds to a *factory* [9] for step instances. A put operation on a control collection with a *control tag* results in the *prescription* (creation) of step instances from one or more step collections with that tag passed as an input argument. These collections and their relationships are defined statically as a CnC *graph* in which a node corresponds to a step, data or item collection, and a directed edge corresponds to a put, get, or prescribe operation.

Discussion

Introduction

Parallel computing has become firmly established since the 1980s as the primary means of achieving high performance from supercomputers. Concurrent Collections (CnC) was developed to address the need for making parallel programming accessible to nonprofessional programmers. One approach that has historically addressed this problem is the creation of *domain-specific languages* (DSLs) that hide the details of parallelism when programming for a specific application domain.

In contrast, CnC is a model for adding parallelism to any host language (which is typically serial and may be a DSL). In this approach, the parallel implementation details of the application are hidden from the domain expert, but are instead addressed separately by users (and tools) that serve the role of *tuning experts*. The basic concepts of CnC are widely applicable. Its premise is that domain experts can identify the intrinsic *data dependences* and *control dependences* in their application (irrespective of lower-level implementation choices). This identification of dependences is achieved by specifying a CnC *graph* for their application. Parallelism is implicit in a CnC graph. A CnC graph has a deterministic semantics, in that all executions are guaranteed to produce the same output state for the same

input. This deterministic semantics and the separation of concerns between the domain and tuning experts are the primary characteristics that differentiate CnC from other parallel programming models.

CnC Specification Graph

The three main constructs in a CnC specification graph are *step collections*, *data collections*, and *control collections*. These collections and their relationships are defined statically. But for each static collection, a set of dynamic *instances* is created as the program executes.

A step collection corresponds to a specific computation, and its instances correspond to invocations of that computation with different input arguments. A control collection is said to *control* a step collection – adding an instance to the control collection *prescribes* one or more step instances i.e., causes the step instances to eventually execute when their inputs become available. The invoked step may continue execution by adding instances to other control collections, and so on.

Steps also dynamically read (*get*) and write (*put*) data instances. The execution order of step instances is constrained only by their producer and consumer relationships, including control relations. A complete CnC specification is a graph where the nodes can be either step, data, or control collections, and the edges represent *producer*, *consumer*, and *control* relationships.

A whole CnC program includes the specification, the step code and the environment. Step code implements the computations within individual graph nodes, whereas the *environment* is the external user code that invokes and interacts with the CnC graph while it executes. The environment can produce data and control instances. It can consume data instances and use control instances to prescribe conditional execution.

Collections Indexed by Tags

Within each collection, control, data, and step instances are each identified by a unique *tag*. In most CnC implementations, tags may be of any data type that supports an equality test and hash function. Typically, tags have a specific meaning within the application. For example, they may be tuples of integers modeling an iteration space (i.e., the iterations of a nested loop structure). Tags can also be points in non-grid spaces – nodes in a tree, in an irregular mesh, elements of a set, etc. Collections use tags as follows:

- A step begins execution with one input argument – the tag indexing that step instance. The tag argument contains the information necessary to compute the tags of all the step's input and output data. For example, in a stencil computation a tag “*i, j*” would be used to access data at positions “*i+1, j+1*”, “*i-1, j-i*”, and so on. In a CnC specification file, a step collection is written (*foo*) and the components of its tag indices can optionally be documented, as in (*foo: row, col*).
- Putting a tag into a control collection will cause the corresponding steps (in all controlled step collections) to eventually execute when their inputs become available. A control collection *C* is denoted as *<C>* or as *<C : x, y, z>*, where *x, y, z* comprise the tag. Instances of a control collection contain no information *except* their tag, so the word “tag” is often used synonymously with “control instance.”
- A data collection is an associative container indexed by tags. The contents indexed by a tag *i*, once written, cannot be overwritten (dynamic single assignment). The immutability of entries within a data collection, along with other features, provides determinism. In a specification file a data collection is referred to with square-bracket syntax: [*x:i, j*].

Using the above syntax, together with *:* and → for denoting prescription and production/consumption relations, we can write CnC specifications that describe CnC graphs. For example, below is an example snippet of a CnC specification showing all of the syntax.

```
// control relationship: myCtrl
    prescribes instances of myStep
<myCtrl> :: (myStep);
// myStep gets items from myData, and
    puts tags in myCtrl and items in
    myData
[myData] → (myStep) → <myCtrl>,
[myData];
```

Further, in addition to describing the graph structure, we might choose to use the CnC specification to document the relationship between tag indices:

```
[myData: i] → (myStep: i) → <myCtrl:
    i+1>, [myData: i+1];
```

Model Execution

During execution, the state of a CnC program is defined by *attributes* of step, data, and control instances. (These attributes are not directly visible to the CnC programmer.) Data instances and control instances each have an attribute *Avail*, which has the value *true* if and only if a put operation has been performed on it. A data instance also has a *Value* attribute representing the value assigned to it where *Avail* is true. When the set of all data instances to be consumed by a step instance and the control instance that prescribes a step instance have *Avail* attribute value *true*, then the value of the step instance attribute *Enabled* is set to *true*. A step instance has an attribute *Done*, which has the value *true* if and only if all of its put operations have been performed.

Instances acquire attribute values monotonically during execution. For example, once an attribute assumes the value *true*, it remains *true* unless an execution error occurs, in which case all attribute values become undefined. Once the *Value* attribute of a data instance has been set to a value through a put operation, assigning it a subsequent value through another put operation produces an execution error, by the single assignment rule. The monotonic assumption of attribute values simplifies program understanding, formulating and understanding the program semantics, and is necessary for deterministic execution.

Given a complete CnC specification, the tuning expert maps the specification to a specific target architecture, creating an efficient schedule. This is quite different from the more common approach of embedding

parallel constructs within serial code. Tag functions provide a tuning expert with additional information needed to map the application to a parallel architecture, and for static analysis, they provide information needed to optimize distribution and scheduling of the application.

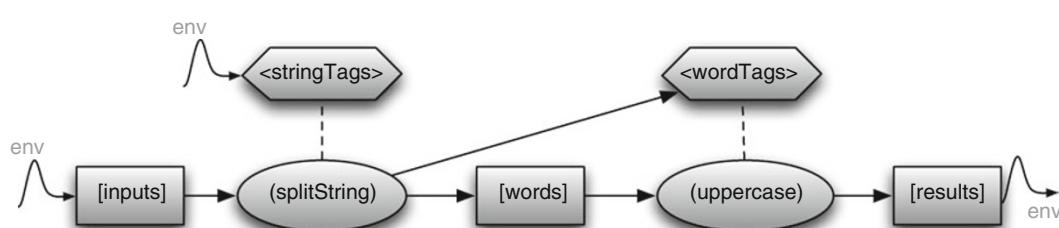
Example

The following simple example illustrates the task and data parallel capabilities of CnC. This application takes a set (or stream) of strings as input. Each string is split into words (separated by spaces). Each word then passes through a second phase of processing that, in this case, puts it in uppercase form.

```
<stringTags> :: (splitString); // step 1
<wordTags>   :: (uppercaseWord); // step 2
// The environment produces initial inputs
// and retrieves results:
env → <stringTags>, [inputs];
env ← [results];
// Here are the producer/consumer relations
// for both steps:
[inputs] → (splitString) → <wordTags>,
[words]; 
[words]   → (uppercaseWord) → [results];
```

The above text corresponds directly to the graph in Fig. 1. Note that separate strings in [input] can be processed independently (data parallelism), and, further, the (splitString) and (uppercase) steps may operate simultaneously (task parallelism).

The only keyword in the CnC specification language is env, which refers to the *environment* – the world outside CnC, for example, other threads or processes



Concurrent Collections Programming Model. Fig. 1 A CnC graph as described by a CnC specification. By convention, in the graphical notation specific shapes correspond to control, data, and step collections. Dotted edges represent prescription (control/step relations), and arrows represent production and consumption of data. Squiggly edges represent communication with the environment (the program outside of CnC)

```

// The execute method "fires" when a tag is available.
// The context c represents the CnC graph containin item and tag collections.
int splitString::execute(const int & t, partStr_context & c ) const
{
    // Get input string
    string in;
    c.input.get(t, in);

    // Use C++ standard template library to extract words:
    istringstream iss(in);
    vector<string> words;
    copy(istream_iterator<string>(iss),
          istream_iterator<string>(),
          back_inserter<vector<string> >(words));

    // Finally, put words into an output item collection:
    for(int i=0; i < words.size(); i++) {
        pair<int,int> wtag(t,i);
        c.wordTags.put( wtag );
        c.words.put( wtag, words[i] );
    }
    return CnC::CNC_Success;
}

// Convert word to upper case form:
int uppercaseWord::execute(const pair<int,int> & t, partStr_context & c ) const
{
    string word;
    c.words.get(t, word);
    strUpper(word);
    c.results.put(t, word);
    return CnC::CNC_Success;
}

int main()
{
    partStr_context c;
    for(...)

        c.input.put(t, string); // Provide initial inputs
        c.wait();               // Wait for all steps to finish
        ...                     // Print results
}

```

Concurrent Collections Programming Model. Fig. 2 C++ code implementing steps and environment. Together with a CnC specification file the above forms a complete application

written in a serial language. The strings passed into CnC from the environment are placed into [inputs] using any unique identifier as a tag. The elements of [inputs] may be provided in any order or in parallel. Each string, when split, produces an arbitrary number of words. These per-string outputs can be numbered 1 through N – a pair containing this number and the original string ID serves as a globally unique tag for all output words. Thus, in the specification we could annotate the collections with tag components indicating the

pair structure of word tags: e.g., (uppercaseWord: stringID, wordNum).

[Figure 2](#) contains C++ code implementing the steps splitString and uppercase. The step implementations, specification file, and code for the environment together make up a complete CnC application. Current implementations of CnC vary as to whether the specification file is required, can be constructed graphically, or can be conveyed in the host language code itself through an API.

Mapping to Target Platforms

There is wide latitude in mapping CnC to different platforms. For each there are several issues to be addressed: grain size, mapping data instances to memory locations, steps to processing elements, and scheduling steps within a processing element. A number of distinct implementations are possible for both distributed and shared memory target platforms, including static, dynamic, or a hybrid of static/dynamic systems with respect to the above choices.

The way an application is mapped will determine its execution time, memory, power, latency, and bandwidth utilization on a target platform. The mappings are specified as part of the static translation and compilation as well as dynamic scheduling of a CnC program. In dynamic run-time systems, the mappings are influenced through scheduling strategies, such as LIFO, FIFO, work-stealing, or priority-based.

Implementations of CnC typically provide a translator and a run-time system. The translator uses a CnC specification to generate code for a run-time system API in the target language. As of the writing of this entry, there are known CnC implementations for C++ (based on Intel's Threading Building Blocks), Java (based on Java Concurrency Utilities), .NET (based on .NET Task Parallel Library), and Haskell.

Step Execution and Data Puts and Gets: There is much leeway in CnC implementation, but in all implementations, step prescription involves creation of an internal data structure representing the step to be executed. Parallel tasks can be spawned eagerly upon prescription, or delayed until the data needed by the task is ready. The `get` operations on a data collection could be blocking (in cases when the task executing the step is to be spawned before all the inputs for the step are available) or non-blocking (the run-time system guarantees that the data is available when `get` is executed). Both the C++ and Java implementations have a rollback and replay policy, which aborts the step performing a `get` on an unavailable data item and puts the step in a separate list associated with the failed `get`. When a corresponding `put` is executed, all the steps in a list waiting on that item are restarted. The Java implementation also has a “delayed async” policy [1], which requires the user or the translator to provide a boolean `ready()` method that evaluates to *true* once all the inputs required by the step are available. Only when `ready()` for a given step

evaluates to *true* does the Java implementation spawn a task to execute the step.

Initialization and Shutdown: All implementations require some code for initialization of the CnC graph: creating step objects and a graph object, as well as performing the initial `puts` into the data and control collections.

In the C++ implementation, ensuring that all the steps in the graph have finished execution is done by calling the `run()` method on the graph object, which blocks until all runnable steps in the program have completed. In the Java implementation, ensuring that all the steps in the graph have completed is done by enclosing all the control collection `puts` from the environment in a Habanero-Java finish construct [14], which ensures that all transitively spawned tasks have completed.

Safety properties: In addition to the differences between step implementation languages, different CnC implementations enforce the CnC graph properties differently. All implementations perform run-time system checks of the single assignment rule, while the Java and .NET implementations also enforce tag immutability. Finally, CnC guarantees determinism as long as steps are themselves deterministic – a contract strictly enforceable only in Haskell.

Memory reuse: Another aspect of CnC run-time systems is garbage collection. Unless the run-time system at some point deletes the items that were put, the memory usage will continue to increase. Managed run-time systems such as Java or .NET will not solve this problem, since an item collection retains pointers to all instances. Recovering memory used by data instances is a separate problem from traditional garbage collection. There are two approaches identified thus far to determine when data instances are dead and can safely be released (without breaking determinism). First, [8] introduces a declarative *slicing annotation* for CnC that can be transformed into a reference counting procedure for memory management. Second, the C++ implementation provides a mechanism for specifying *use counts* for data instances, which are discarded after their last use. Irrespective of which of these mechanisms is used, data collections can be released after a graph has finished running. Frequently, an application uses CnC for finite computations inside a serial outer loop, thereby reclaiming all memory between iterations.



Related Work

Table 1 is used to guide the discussion in this section. This table classifies programming models according to their attributes in three dimensions: *Declarative*, *Deterministic*, and *Efficient*. A few representative examples are included for each distinct set of attributes. The reader can extrapolate this discussion to other programming models with similar attributes in these three dimensions.

A number of lower-level programming models in use today – e.g., Intel TBB [18], .Net Task Parallel Library [19], Cilk, OpenMP [3], Nvidia CUDA, Java Concurrency [17] – are non-declarative, nondeterministic, and efficient. Here a programming model is considered to be efficient if there are known implementations that deliver competitive performance for a reasonably broad set of programs. Deterministic Parallel Java [7] is an interesting variant of Java; it includes a subset that is provably deterministic, as well

as constructs that explicitly indicate when determinism cannot be guaranteed for certain code regions, which is why it contains a “hybrid” entry in the *Deterministic* column.

The next three languages in the table – High Performance Fortran (HPF) [15], X10 [6], Linda [10] – contain hybrid combinations of imperative and declarative programming in different ways. HPF combines a declarative language for data distribution and data parallelism with imperative (procedural) statements, X10 contains a functional subset that supports declarative parallelism, and Linda is a coordination language in which a thread’s interactions with the tuple space is declarative. Linda was a major influence on the CnC design. CnC shares two important properties with Linda: both are coordination languages that specify computations and communications via a tuple/tag namespace, and both create new computations by adding new tuples/tags to the namespace. However, CnC also differs from Linda in many ways. For example, an `in()` operation in Linda atomically removes the tuple from the tuple space, but a CnC `get()` operation does not remove the item from the collection. This is a key reason why Linda programs can be nondeterministic in general, and why CnC programs are provably deterministic. Further, there is no separation between tags and values in a Linda tuple; instead, the choice of tag is implicit in the use of wildcards. In CnC, there is a separation between tags and values, and control tags are first class constructs like data items.

The last four programming models in the table are both declarative and deterministic. Asynchronous Sequential Processes [2] is a recent model with a clean semantics, but without any efficient implementations. In contrast, the remaining three entries are efficient as well. StreamIt [12, 13] is representative of a modern streaming language, and LabVIEW [20] is representative of a modern dataflow language. Both streaming and dataflow languages have had major influence on the CnC design. The CnC semantic model is based on dataflow in that steps are functional and execution can proceed whenever data is ready.

However, CnC differs from dataflow in some key ways. The use of control tags elevates control to a first-class construct in CnC. In addition, item collections allow more general indexing (as in a tuple space) compared to dataflow arrays (I-structures). CnC is like

Concurrent Collections Programming

Model. Table 1 Comparison of several parallel programming models

Parallel prog. model	Declarative	Deterministic	Efficient
Intel TBB [18]	No	No	Yes
.Net Task Par. Lib.	No	No	Yes
Cilk	No	No	Yes
OpenMP [3]	No	No	Yes
CUDA	No	No	Yes
Java Concurrency [17]	No	No	Yes
Det. Parallel Java [7]	No	Hybrid	Yes
High Perf. Fortran [15]	Hybrid	No	Yes
X10 [6]	Hybrid	No	Yes
Linda [10]	Hybrid	No	Yes
Asynch. Seq. Processes [2]	Yes	Yes	No
StreamIt [12]	Yes	Yes	Yes
LabVIEW [20]	Yes	Yes	Yes
CnC	Yes	Yes	Yes

streaming in that the internals of a step are not visible from the graph that describes their connectivity, thereby establishing an isolation among steps. A producer step in a streaming model need not know its consumers; it just needs to know which buffers (collections) to perform read and write operations on. However, CnC differs from streaming in that put and get operations need not be performed in FIFO order, and (as mentioned above) control is a first-class construct in CnC. Further, CnC's dynamic put/get operations on data and control collections serves as a general model that can be used to express many kinds of applications that would not be considered to be dataflow or streaming applications.

Future Directions

Future work on the CnC model will focus on incorporating more power in the specification language (module abstraction, libraries of patterns) and integration with persistent data models.

Combinations of static/dynamic treatment of scheduling and step/data distribution will continue to be explored. Run-time strategies, such as for reducing overhead for finer-grained parallelism and for memory management, will be developed.

Static graph analysis will play a role in performance optimization in the future.

Related Entries

- [Cilk](#)
- [Dependences](#)
- [Deterministic Parallel Java](#)
- [HPF \(High Performance Fortran\)](#)
- [Linda](#)
- [OpenMP](#)

Bibliographic Notes and Further Reading

The basic principles behind the Concurrent Collections programming model are outlined in [4]. The model is built on past work done at Hewlett Packard Labs on TStreams, described in [16].

A technique is presented in [8] for memory management of data items with lifetimes that are longer than a single computation step.

[5] is a pioneering paper in dataflow languages. For a description of coordination languages and their use, see [11].

Many members of the Concurrent Collections community gathered at workshops held in 2009 (<http://habanero.rice.edu/cnc09>) and 2010 (<http://cnc10.rice.edu/cnc10>).

Bibliography

1. Budimlić Z, Burke M, Cavé V, Knobe K, Lowney G, Newton R, Palsberg J, Peixotto D, Sarkar V, Schlimbach F, Sağnak T (February 2010) Cnc programming model. Technical Report TR10-5, Rice University
2. Denis C, Ludovic H, Bernard PS (2009) Asynchronous sequential processes. *Information Comput*, 207(4):459–495
3. Chandra R, Dagum L, Kohr D, Maydan D, McDonald J, Menon R (2001) Programming in OpenMP. Academic Press, San Diego, California
4. Chandramowlishwaran A, Budimlic Z, Knobe K, Lowney G, Sarkar V, Tregiari L (2009) Multi-core implementations of the concurrent collections programming model. In: Proceedings of 14th international workshop on compilers for parallel computers (CPC). Zurich, Switzerland, Jan 2009
5. Dennis JB (1974) First version of a data flow procedure language. In: Programming symposium, proceedings colloque sur la programmation, Paris, pp 362–376
6. Charles P et al (2005) XI0: An object-oriented approach to non-uniform cluster computing. In: Proceedings of OOPSLA'05, ACM SIGPLAN conference on object-oriented programming systems, languages and applications. San Diego, California, pp 519–538
7. Bocchino RL et al (2009) A type and effect system for Deterministic Parallel Java. In: Proceedings of OOPSLA'09, ACM SIGPLAN conference on object-oriented programming systems, languages and applications. Orlando, Florida, pp 97–116
8. Budimlić Z et al (2008) Declarative aspects of memory management in the concurrent collections parallel programming model. In DAMP '09: the workshop on declarative aspects of multicore programming. Savannah, Georgia, ACM, pp 47–58
9. Gamma E, Helm R, Johnson R, Vlissides J (1994) Design patterns: elements of reusable object-oriented software. Addison-Wesley, Reading, Massachusetts
10. Gelernter D (1985) Generative communication in linda. *ACM Trans Program Lang Syst* 7(1):80–112
11. Gelernter D, Carriero N (1992) Coordination languages and their significance. *Commun ACM* 35(2):97–107
12. Gordon MI et al (2002) A stream compiler for communication-exposed architectures. In: ASPLOS-X: Proceedings of the 10th international conference on architectural support for programming languages and operating systems. ACM, New York, pp 291–303
13. Gordon MI et al (2006) Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: ASPLOS-XII:

- Proceedings of the 12th international conference on architectural support for programming languages and operating systems. ACM, New York, pp 151–162
14. Habanero multicore software research project. <http://habanero.rice.edu>.
 15. Kennedy K, Koelbel C, Zima HP (2007) The rise and fall of high performance Fortran. In: Proceedings of HOPL'07, Third ACM SIGPLAN history of programming languages conference, San Diego, California, pp 1–22
 16. Knobe K, Offner CD (2004) Tstreams: A model of parallel computation (preliminary report). Technical Report HPL-2004-78, HP Labs
 17. Peierls T, Goetz B, Bloch J, Bowbeer J, Lea D, Holmes D (2005) Java concurrency in practice. Addison-Wesley Professional, Reading, Massachusetts
 18. Reinders J (2007) Intel threading building blocks: Outfitting C++ for Multi-Core Processor Parallelism. O'Reilly Media, Sebastopol, California
 19. Toub S (2008) Parallel programming and the .NET Framework 4.0. <http://blogs.msdn.com/pfxteam/archive/2008/10/10/8994927.aspx>
 20. Travis J, Kring J (2006) LabVIEW for everyone: graphical programming made easy and fun, 3rd edn. Prentice Hall, Upper Saddle River, New Jersey

the term “CML” refers to a specific language implementation, it is also used to refer to implementations of its language primitives in other systems.

Discussion

Concurrent ML Basics

Concurrent ML was designed with the goal of supporting high-level concurrent programming. Its design was originally motivated by the idea that user-interface software should be concurrent and that message passing was the best way to construct such concurrent programs [5, 30, 38]. Its basic concurrency features were heavily influenced by earlier message-passing designs, including Hoare's *Communicating Sequential Processes* [20] and Cardelli's Amber language [6]. Applications of CML include a multithreaded GUI toolkit [18], a distributed tuple-space implementation [37], and a system for implementing partitioned applications in a distributed setting [40].

Because CML is embedded in SML, SML specifications are used to introduce CML's features and the code examples are also written in SML. For readers who are unfamiliar with SML syntax, this paragraph provides a brief description of SML specification and type syntax. Specifications in SML are declarations that appear in the signature of a module and are used to describe the types, values, etc., defined by the module. For example, the `output` function has the specification

```
val output : (outstream * string)
           -> unit
```

which specifies that `output` is a function that takes a pair of arguments, an `outstream` and a `string`, and returns the `unit` value. `Unit` is the type with only one value (written as “`()`”). The symbol “`*`” is the product type constructor and the symbol “`->`” is the function type constructor. Lastly, SML is a polymorphic language, which means that specifications can have universally qualified type variables. For example:

```
type 'a vector
val length : 'a vector -> int
```

is the specification of the abstract vector type constructor and its `length` function. The identifier “`'a`” is an SML type variable. The `vector` type constructor can

Concurrent Logic Languages

Logic Languages

Concurrent ML

JOHN REPPY

University of Chicago, Chicago, IL, USA

Synonyms

CML

Definition

Concurrent ML (CML) is a higher-order concurrent language embedded in the sequential language Standard ML (SML). CML's basic programming model consists of dynamically created threads that communicate via message passing over dynamically created channels. CML also provides *first-class synchronous operations*, called event values, which support user-defined synchronization and communication abstractions. While

be applied to other types to construct arbitrary vector types, such as “int vector” and “int vector vector” (note that type-constructor application is *postfix*). The `length` function will work on any of these vector types, since it is polymorphic.

New threads are created in CML using the `spawn` function, which has the following SML value specification:

```
val spawn : (unit -> unit)
    -> thread_id
```

Because SML is a *strict* functional language, the argument to `spawn` must be encapsulated in a unit to unit function or thunk. The `spawn` function creates a new thread to evaluate its argument and returns the new thread’s ID. CML threads communicate over typed channels that have the following signature:

```
type 'a chan
val channel : unit -> 'a chan
val recv : 'a chan -> 'a
val send : ('a chan * 'a) -> unit
```

Message passing is synchronous: both the `recv` and `send` operation are blocking. The model is distributed in that threads only interact via messages, but it is usually implemented in shared memory and is quite efficient. In addition to channels, CML provides other communication mechanisms, such as asynchronous channels and synchronous variables. The semantics of these mechanisms is defined in terms of the channel and thread operations, but their implementation is more efficient.

As an illustration of these basic primitives, the following code creates a simple producer/consumer thread pair, which communicate over a shared channel:

```
let val ch = channel ()
  fun producer () = (
    send (ch, produce ());
    producer ())
  fun consumer () = (
    consume (recv ch);
    consumer ())
in
  spawn producer;
  spawn consumer
end
```

This code illustrates a couple of idioms that CML inherits from SML. The first is the use of tail-recursion to define threads that loop for ever. The second is the use of lexical scoping to limit access to channels. In this case, the channel `ch` is visible to both the producer and consumer threads, but not outside the scope of the let binding.

First-Class Synchronous Operations

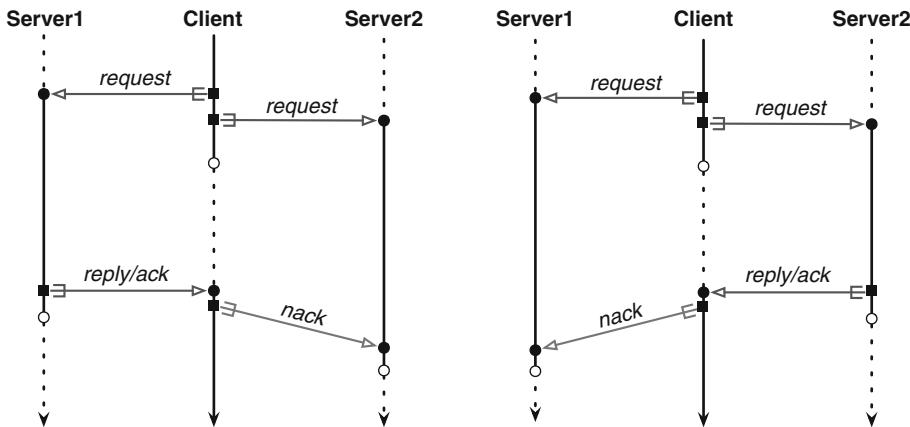
The distinguishing feature of CML is its support for *first-class synchronous operations*. Much the same way that higher-order languages support first-class function values, CML supports first-class synchronization values. This design is motivated by two observations:

1. Most interprocess interactions involve multiple messages
2. Processes need to interact with multiple partners

For example, consider a typical client–server protocol. The client sends a request message to the server and then waits for a reply. But suppose that the client wants to request a result from two different servers, accepting whichever reply it receives first? In that case, the protocol must include a *negative acknowledgment* mechanism for the case where the client wishes to abort the protocol. [Figure 1](#) illustrates the two possible outcomes of this scenario. This example assumes the existence of an asynchronous mechanism for transmitting the negative acknowledgments.

The client-side code for this interaction might look something like the following pseudo-CML code, where each server has its own request channel (`reqCh1` and `reqCh2`) and the requests consist of a triple of the request message, a private reply channel, and a signal variable for aborting the transaction:

```
let val replCh1 = channel()
  val nack1 = cvar()
  val replCh2 = channel()
  val nack2 = cvar()
in
(* send requests to the servers *)
  send (reqCh1,
    (req1, replCh1, nack1));
  send (reqCh2,
    (req2, replCh2, nack2));
(* wait for a reply from one of
 * the servers *)
```



Concurrent ML. Fig. 1 Interacting with two servers

```

selectRecv [
  (rep1Ch1, fn repl1 => (
    set nack2; act1 repl1)),
  (rep1Ch2, fn repl2 => (
    set nack1; act2 repl2))
]
end

```

This example uses two additional concurrency features that are not part of CML, but which can be easily definable in CML. The “cvars” are unit-valued write-once synchronous variables that are used to implement the asynchronous negative acknowledgements, and the `selectRecv` function implements a nondeterministic choice of reading from a list of channels paired with message handlers. As can be seen from this code, the interactions with the two servers are quite entangled. The situation would become even more complicated if a third server were added. Furthermore, the client and server sides of the protocol are split apart, which makes changes to the protocol more difficult to implement.

The traditional mechanism of procedural abstraction is not appropriate in this situation, because it hides away the synchronous aspect of the operations and does not handle the negative acknowledgments.

CML solves this problem by introducing a new abstraction mechanism called *first-class synchronous operations*. It defines a type of abstract values, called events, that represent synchronous operations, such as receiving a message on a channel or a timeout.

`type 'a event`

An event value represents a potential computation with latent communication and synchronization effects. Its type argument specifies the type of the result when a thread synchronized on the event, which is done using the following operator:

`val sync : 'a event -> 'a`

Base-event constructors create event values for the channel communication primitives.

```

val recvEvt : 'a chan -> 'a event
val sendEvt : 'a chan * 'a
      -> unit event

```

Note that these functions are *pure* – they define event values but have no side effects themselves. They also satisfy the obvious equations:

$$\begin{aligned} \text{sync} \circ \text{recvEvt} &= \text{recv} \\ \text{sync} \circ \text{sendEvt} &= \text{send} \end{aligned}$$

The power of events comes from the combinators that CML provides for building more complicated values. The first of these is the `wrap` combinator, which takes an event value and a post-synchronization action and returns a new event that will apply the action after synchronization. It has the type

```

val wrap : ('a event * ('a -> 'b))
      -> 'b event

```

and satisfies the equation

$$\text{sync}(\text{wrap}(ev, f)) = f(\text{sync } ev)$$

for any event value ev and function f that are well typed for the context.

The second combinator is `choose`, which takes a list of event values and returns a new event that represents the nondeterministic choice of the events in the list. It has the type

```
val choose : 'a event list  
      -> 'a event
```

Note that all of the events in the list must have the same result type, but one can use `wrap` to convert between types as necessary. The `sync` operator and `wrap` and `choose` combinators define the *PML subset* of CML, which is named after the first language to support first-class synchronization [34]. Full CML adds two event-generator combinators, which are used to define pre-synchronization operations. The first of these is the `guard` combinator, which takes a event-valued function and returns an event value that will evaluate the function at synchronization time and then synchronize on its result. It has the type

```
val guard : (unit -> 'a event)  
      -> 'a event
```

and satisfies the equation

$$\text{sync}(\text{guard } f) = \text{sync}(f())$$

for any function f that is well typed for the context. Typically the `guard` combinator is used to package up operations, such as initiating a transaction with a server, that should be done before synchronization. As long as the operations in the function are idempotent, `guard` is sufficient, but when they are not, some additional mechanism is required for aborting the transaction. The final event combinator provides this mechanism:

```
val withNack :  
      (unit event -> 'a event)  
      -> 'a event
```

Like `guard`, its argument is a function that is evaluated at synchronization time to produce an event value, which is then synchronized on. What is different is that the function is passed an event-valued argument, which is a negative-acknowledgment event that is used to signal when some other event in the synchronization is chosen instead.

A more modular interface to the servers in the client–server example from above can be implemented using CML’s abstraction mechanisms. Using event values, one can define an abstract interface that hides the details of the client–server protocol from the client:

```
type server  
val rpcEvt : (server * req)  
      -> repl event
```

With this interface, the client-side code has a clear separation of concerns between the interactions with the two servers.

```
sync (  
  choose [  
    wrap (rpcEvt server1,  
          fn repl1 => act1 repl1),  
    wrap (rpcEvt server2,  
          fn repl2 => act2 repl2)  
  ] )
```

The implementation of this interface is straightforward using CML’s event combinators. The server type is represented by a request channel that takes a triple of the request message, a private reply channel, and a negative-acknowledgment event that signals when the transaction should be aborted.

```
type req_msg =  
  (req * repl chan * unit event)  
type server = reqmsg chan
```

The `rpcEvt` function is implemented using the `withNack` combinator.

```
fun rpcEvt (serverCh, req) =  
  withNack (fn nackEvt => let  
    val replCh = channel ()  
    in  
      send (serverCh,  
            (req, replCh, nackEvt));  
      recvEvt replCh  
    end)
```

Modularity is supported by defining the `rpcEvt` function in the module that implements the service and only exposing the abstract interface to clients.

Event values also provide a uniform framework for incorporating other types of synchronous operations into the language. For example, synchronizing on thread termination is supported by the combinator

```
val joinEvt : thread_id
    -> unit event
```

and timeouts are supported by

```
val timeOutEvt : time -> unit event
```

System services, such as I/O and system-process management, are also modeled as event-valued operations.

Events can also be used to implement many of the high-level concurrency mechanisms that have been defined over the years, including Ada's rendezvous mechanisms [12], Multilisp's futures [19], asynchronous RPCs (or promises) [25], Linda-style distributed tuple spaces [7], and the Join-calculus communication mechanisms [17]. Many of these implementations are described in the definitive description of CML: *Concurrent Programming in ML* [37].

Semantics and Analysis

Since its earliest days, CML has been the subject of research on the semantics of concurrent languages and on program analyses for concurrent programs [3, 36].

Because CML provides a mechanism for implementing communication and synchronization abstractions, one might ask what expressiveness limits are there on these abstractions? The answer is that it depends on the underlying communication primitives. In the case of CML, which provides synchronous communication with output guards (i.e., `sendEvt`), communication mechanisms that require two-way common knowledge can be implemented, but it is not possible to define a three-way synchronization mechanism [28]. If CML were based on asynchronous message passing (i.e., non-blocking send) or did not have output guards, then even two-way common knowledge would be impossible to achieve.

The problem of program analysis for CML has also been studied by a number of researchers. Nielson and Nielson developed an effects-based analysis for detecting when programs written in a subset of CML have *finite topology* and thus can be mapped onto a finite processor network [27]. Debbabi *et al.* developed a

type-based control-flow analysis for a CML subset [10], but did not propose any applications for their analysis. Colby developed an abstract-interpretation for a subset of CML that is based on a semantics that uses control paths (i.e., an execution trace) to identify threads. Unlike using spawn points to identify threads (as in [21]), control paths distinguish multiple threads created at the same spawn point, which is a necessary condition to understand the topology of a program. Reppy and Xiao developed an analysis for CML that can be used to specialize channel primitives for better performance in a parallel implementation of CML [32]. For example, if a channel is only used for point-to-point communication, then it can be implemented using fixed storage and an atomic swap instruction.

Implementations

The idea of awarding first-class status to synchronous operations was first implemented as part of the PML language [34], which was the metalanguage for the Pegasus system designed at Bell Laboratories [38]. These ideas were then reimplemented as Concurrent ML on top of the Standard ML of New Jersey system (SML/NJ) [1] and released in the autumn of 1990. As discussed above, CML extended the PML design with two important combinators: `guard` and a precursor to `withNack` called `wrapAbort` [35]. These primitives greatly increased the expressiveness of the mechanism. CML continues to be distributed as part of SML/NJ.

CML's implementation uses SML/NJ's heap-allocated first-class continuations to implement threads [2]. This implementation strategy has three important advantages: The implementation is written in a high-level language (SML), threads are extremely lightweight in both space and time, and threads are garbage collected. CML is preemptively scheduled, but, like most of the user-space threading libraries developed in the 1980s and 1990s, it does not support multiprocessors (the main reason for this lack of multiprocessor support is because the underlying SML/NJ system does not support multiprocessors). Because the main motivation for CML was the use of concurrency as a programming idiom, and not parallelism, the lack of multiprocessor support was not a major concern. More recently, there has been an interest in parallel implementations of the

CML primitives. Reppy and Xiao developed an *optimistic concurrency* algorithm for the asymmetric subset of CML (i.e., CML without output guards) [33]. This algorithm was later extended to include support for the full range of CML primitives and has been implemented both as part of the Manticore system [16] and as a library in C# [31].

In addition to the SML/NJ implementation, the CML design has been ported to a number of other systems and languages, including other implementations of SML, other dialects of ML, other functional languages, such as Haskell and Scheme, and other high-level languages, such as Java.

- MLton is a whole-program optimizing compiler for Standard ML, which has its own implementation of the CML primitives [26]. Like the SML/NJ version, it multiplexes CML threads onto a single system thread and supports preemptive scheduling. Its implementation of threads is stack based, so they are somewhat heavier than the SML/NJ version and are not garbage collected. This implementation is being used by Jagannathan and his students to explore concurrent language design [41].
- One of the earliest re-implementations of CML was in the OCaml system, which is another dialect of ML [24].
- There have been two implementations of CML's primitives in Haskell [8, 39]; both implemented using the primitives of Concurrent Haskell [29]. These implementations support multiprocessors, since the underlying Concurrent Haskell implementation runs on multiprocessors.
- The PLT-Scheme system implements CML-style concurrency with an additional mechanism, called *kill-safe abstractions*, to support safe asynchronous termination of threads [15]. The Scheme 48 system also supports CML-style concurrency [22].
- Outside of the world of functional languages, the CML primitives have been reimplemented in Java [11] and C# [31].

Derivatives

CML has also served as the basis for further research into language mechanisms for concurrency. One of the first examples is Krumvieda's *Distributed ML* language

[9, 23], which combined the Isis model of atomic multicast [4] with CML's events.

More recently, researchers have been exploring the marriage of software transactions with CML's events. Donnelly and Fluet generalized the idea of events to allow protocols that involve multiple synchronization points [13]. They defined a new combinator

```
val thenEvt :
  'a event * ('a -> 'b event)
  -> 'b event
```

with the semantics that the expression *thenEvt evf* defines an event value, which synchronizes on *ev*, passes the result to *f*, and then synchronizes on the result of *f*. The key feature of this combinator is the synchronization is *transactional*; i.e., either all the synchronizations occur or none. Donnelly and Fluet's design was implemented in Haskell, but it has been adapted to ML by Effinger-Dean, Kehrt, and Grossman [14]. A related idea is the notion of *stabilizers*, which are a checkpointing mechanism for CML programs [41].

Bibliography

1. Appel AW, MacQueen DB (1991) Standard ML of New Jersey. In: Programming language implementation and logic programming. Lecture notes in computer science, vol 528. Springer, New York, pp 1–26
2. Appel AW (1992) Compiling with continuations. Cambridge University Press, Cambridge
3. Berry D, Milner R, Turner DN (1992) A semantics for ML concurrency primitives. In: Conference record of the 19th annual ACM symposium on principles of programming languages (POPL '92), Albuquerque, Jan 1992, pp 119–129
4. Birman KP, Joseph TA (1987) Reliable communication in the presence of failures. ACM T Comput Syst 5(1):47–76
5. Cardelli L, Pike R (1985) Squeak: a language for communicating with mice. In: SIGGRAPH'85, San Francisco, July 1985, pp 199–204
6. Cardelli L (1986) Amber. In: Combinators and functional programming languages. Lecture notes in computer science, vol 242. Springer, New York, pp 21–47, July 1986
7. Carriero N, Gelernter D (1989) Linda in context. Commun ACM 32(4):444–458
8. Chaudhuri A (2009) A Concurrent ML library in Concurrent Haskell. In: Proceedings of the 14th ACM SIGPLAN international conference on functional programming, Aug–Sept 2009. ACM, New York, pp 269–280
9. Cooper R, Krumvieda C (1994) Distributed programming with asynchronous ordered channels in Distributed ML. In: Birman KP, Renesse RV (eds) Reliable distributed computing with the Isis toolkit, pp 359–369. IEEE Computer Society Press, Los Alamitos

10. Debbabi M, Faour A, Tawbi N (1996) Efficient type-based control-flow analysis of higher order concurrent programs. In: Proceedings of the international workshop on functional and logic programming, IFL'96, Sept 1996. Lecture notes in computer science, vol 1268. Springer, New York, pp 247–266
11. Demaine ED (1997) Higher-order concurrency in java. In: Proceedings of the parallel programming and java conference (WoTUG20), Enschede, Apr 1997, pp 34–47. Available from <http://theory.csail.mit.edu/~edemaine/papers/WoTUG20/>
12. United States Department of Defense, American National Standards Institute (1983) Reference manual for the Ada programming language. Springer, New York
13. Donnelly K, Fluet M (2008) Transactional events. *J Funct Program* 18(5–6):649–706
14. Effinger-Dean L, Kehrt M, Grossman D (2008) Transactional events for ML. In: Proceedings of the 13th ACM SIGPLAN international conference on functional programming, Sept 2008. ACM, New York, pp 103–114
15. Flatt M, Findler RB (2004) Kill-safe synchronization abstractions. In: Proceedings of the SIGPLAN conference on programming language design and implementation (PLDI'04), June 2004. ACM, New York, pp 47–58
16. Fluet M, Rainey M, Reppy J, Shaw A, Xiao Y (2007) Manticore: a heterogeneous parallel language. In: Proceedings of the ACM SIGPLAN workshop on declarative aspects of multicore programming, Jan 2007. ACM, New York, pp 37–44
17. Fournet C, Gonthier G (1996) The reflexive CHAM and the join-calculus. In: Conference record of the 23rd annual ACM symposium on principles of programming languages (POPL'96), Jan 1996. ACM, New York, pp 372–385
18. Gansner ER, Reppy JH (1993) A multi-threaded higher-order user interface toolkit. *Software trends*, vol 1. Wiley, New York, pp 61–80
19. Halstead Jr RH (1985) Multilisp: a language for concurrent symbolic computation. *ACM T Program Lang Syst* 7(4):501–538
20. Hoare CAR (1978) Communicating sequential processes. *Commun ACM* 21(8):666–677
21. Jagannathan S, Weeks S (1994) Analyzing stores and references in a parallel symbolic language. In: Conference record of the 1994 ACM conference on lisp and functional programming, June 1994. ACM, New York, pp 294–305
22. Kelsey R, Rees J, Sperber M (2008) The incomplete scheme 48 reference manual. Available from www.s48.org
23. Krumvieda CD (1993) Distributed ML: abstractions for efficient and fault-tolerant programming. Ph.D. thesis, Department of Computer Science, Cornell University, Ithaca, Aug 1993. Available as Technical Report TR 93-1376
24. Leroy X (2000) The objective Caml system (release 3.00), Apr 2000. Available from <http://caml.inria.fr>
25. Liskov B, Shrira L (1988) Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In: Proceedings of the SIGPLAN'88 conference on programming language design and implementation, Atlanta, June 1988, pp 260–267
26. MLton: Concurrent ML. <http://mlton.org/ConcurrentML>
27. Nielson HR, Nielson F (1994) Higher-order concurrent programs with finite communication topology. In: Conference record of the 21st annual ACM symposium on principles of programming languages (POPL '94), Portland, Jan 1994, pp 84–97
28. Panangaden P, Reppy JH (1997) The essence of concurrent ML. In: Nielson F (ed) ML with concurrency. Springer, New York (Chap 1)
29. Peyton Jones S, Gordon A, Finne S (1996) Concurrent haskell. In: Conference record of the 23rd annual ACM symposium on principles of programming languages (POPL'96), Jan 1996. ACM, New York, pp 295–308
30. Pike R (1989) A concurrent window system. *Comput Syst* 2(2):133–153
31. Reppy J, Russo C, Xiao Y (2009) Parallel concurrent ML. In: Proceedings of the 14th ACM SIGPLAN international conference on functional programming, Aug–Sept 2009. ACM, New York, pp 257–268
32. Reppy J, Xiao Y (2007) Specialization of CML message-passing primitives. In: Conference record of the 34th annual ACM symposium on principles of programming languages (POPL '07), Jan 2007. ACM, New York, pp 315–326
33. Reppy J, Xiao Y (2008) Toward a parallel implementation of concurrent ML. In: Proceedings of the ACM SIGPLAN workshop on declarative aspects of multicore programming, Jan 2008. ACM, New York
34. Reppy JH (1988) Synchronous operations as first-class values. In: Proceedings of the SIGPLAN'88 conference on programming language design and implementation, Atlanta, June 1988, pp 250–259
35. Reppy JH (1991) CML: a higher-order concurrent language. In: Proceedings of the SIGPLAN'91 conference on programming language design and implementation, June 1991. ACM, New York, pp 293–305
36. Reppy JH (1991) An operational semantics of first-class synchronous operations. Technical Report TR 91-1232, Department of Computer Science, Cornell University, Ithaca, Aug 1991
37. Reppy JH (1999) Concurrent programming in ML. Cambridge University Press, Cambridge
38. Reppy JH, Gansner ER (1986) A foundation for programming environments. In: Proceedings of the ACM SIGSOFT/SIGPLAN software engineering symposium on practical software development environments, Palo Alto, pp 218–227, Dec 1986
39. Russell G (2001) Events in Haskell, and how to implement them. In: Proceedings of the sixth ACM SIGPLAN international conference on functional programming, Florence, Sept 2001, pp 157–168
40. Young C, Lakshman YN, Szymanski T, Reppy J, Pike R, Narlikar G, Mullender S, Grosse E (2001) Protium, an infrastructure for partitioned applications. In: Proceedings of the eighth IEEE workshop on hot topics in operating systems (HotOS), Elmau, Jan 2001, pp 41–46
41. Ziarek L, Schatz P, Jagannathan S (2006) Stabilizers: a modular checkpointing abstraction for concurrent functional programs. In: Proceedings of the eleventh ACM SIGPLAN international conference on functional programming, Sept 2006. ACM, New York, pp 136–147

Concurrent Prolog

► [Logic Languages](#)

Configurable, Highly Parallel Computer

► [Blue CHiP](#)

Congestion Control

► [Congestion Management](#)

Congestion Management

PEDRO J. GARCIA

Universidad de Castilla-La Mancha, Albacete, Spain

Synonyms

[Congestion control](#)

Definition

Congestion appears in an interconnection network when intense traffic clogs any number of internal network paths, thus slowing down traffic flowing. Congestion management refers to any strategy focused on avoiding, reducing, or eliminating network congestion and/or its negative impact on network performance.

Discussion

Introduction

Congestion is a phenomenon that may dramatically degrade network performance. Congestion situations may actually appear in both computer communication networks (like the Internet) and interconnection networks of parallel computing systems. In the former, packet dropping is allowed ("lossy" networks), thus congestion is not a critical problem as congested packets can be discarded. Nevertheless, congestion should be avoided because dropped packets have consumed

bandwidth, thus wasting it, and packet retransmissions significantly increase individual packet latency. In the latter, on the contrary, packets usually can not be discarded ("lossless" networks) due to the prohibitive delay in detection mechanisms and retransmissions for current parallel applications. Moreover, the designers of interconnection networks for modern parallel systems try to use as few network components as possible, due to their high cost and power consumption, but this increases link utilization, thus making congestion situations more likely to happen. Even when interconnection networks are overdimensioned, the actions taken by power management mechanisms tend to bring the network close to saturation, with identical consequences.

Therefore, in modern, high-performance parallel computing systems, thorough strategies should be used in order to solve the problems related to congestion situations that may arise in the network interconnecting processing or storage nodes. In fact, congestion management is one of the most important issues that high-performance interconnect designers face nowadays, always trying to guarantee a certain network performance level, even in congestion situations. The following sections offer an overview of the congestion phenomenon, its effects on network performance and the most common approaches to deal with congestion in parallel computing systems.

Congestion in Interconnection Networks

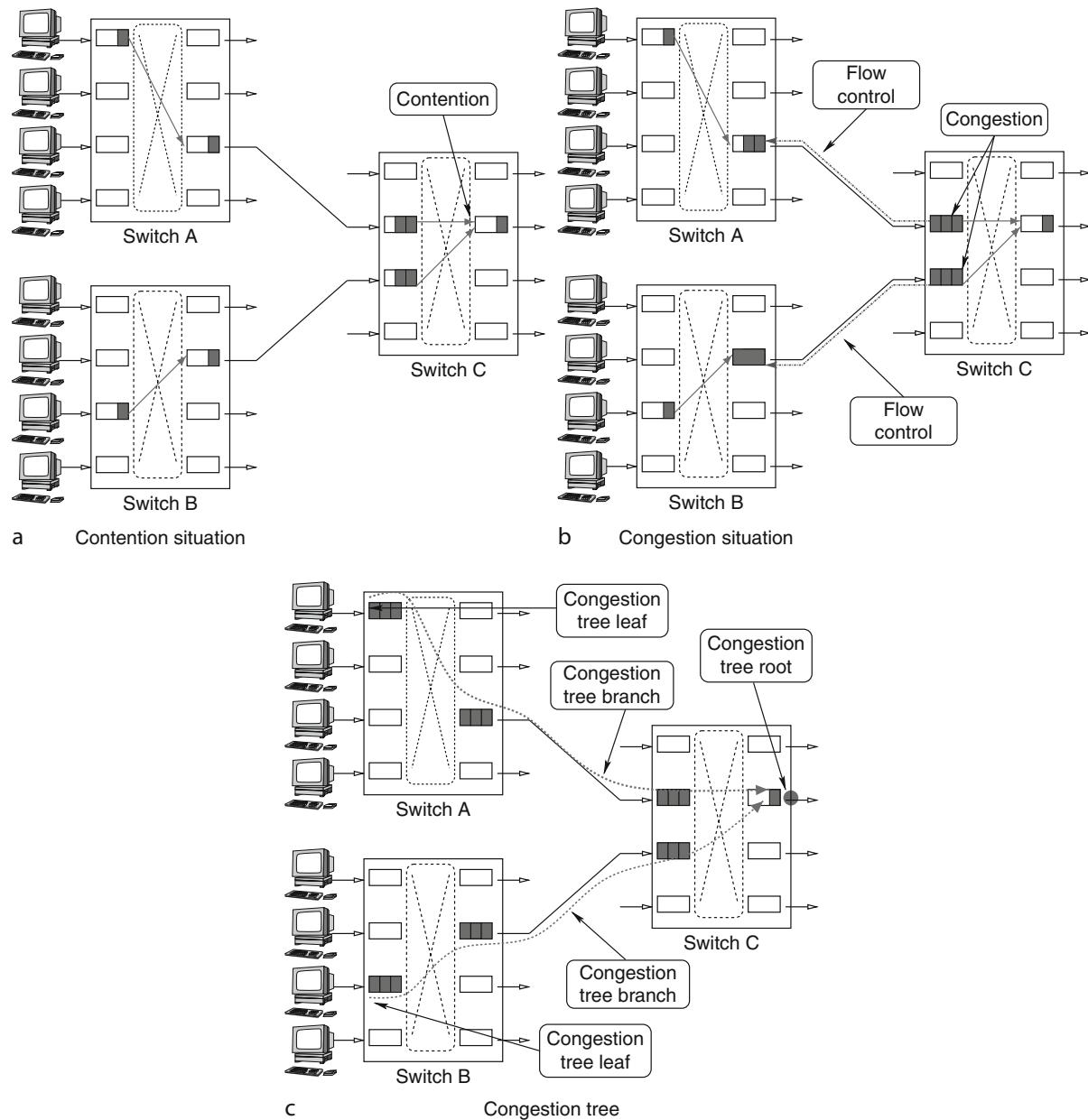
As mentioned above, congestion may appear under conditions of intense traffic in the network, when some of its internal paths, or sections of paths, become jammed. However, this idea is just the most basic definition of that phenomenon, while any study of the main congestion management strategies requires a deeper, previous analysis of both the congestion formation process and its consequences.

Congestion Basics

The immediate cause of congestion situations in interconnection networks is contention, which happens when several flows of packets crossing the network simultaneously request access to some network resource (typically, a switch output port). If the internal

speedup of switches (the maximum speed at which packets can be forwarded from input to output ports, relative to link speed) is not enough for granting several requests at the same time, the access to the requested output port will be granted to only one packet, while the rest must wait. [Figure 1a](#) shows a contention situation caused by two incoming flows requesting the same

output port at Switch C. In this example, it is assumed that switch speedup is 1. Note that the example (and the following ones) shows switches with buffers (queues) at both input and output ports, but it could be easily adapted to switches with other queue schemes. Note also that the requested output port may be connected to an end node or to another switch.



Congestion Management. Fig. 1 Contention and congestion in switches of an interconnection network

When contention persists over time, congestion appears. In these situations, the buffers containing the blocked packets will be filled and the flow control, in lossless networks, will prevent other switches (or end nodes) from sending packets to the congested ports. Although flow control is essential to avoid discarding packets, it will rapidly propagate congestion to other switches, as packets stored at some of their ports will also be blocked. [Figure 1b](#) shows a congestion situation whose origin is the contention situation shown in [Fig. 1a](#). In this example, it is assumed that the network is a lossless one (so, packets are not discarded when buffers are full), and also that the sources of the contending flows continuously inject packets into the network. These flows contributing to congestion are usually referred to as “congested flows.” Note also that packet-level flow control is assumed, but a similar situation could be easily imagined for networks using flow control at other levels (flit-level, for instance).

In this way, congestion may progressively spread through the network, even reaching the sources of congested flow packets. The whole of the network resources affected by the spreading of congestion is commonly known as a “congestion tree.” In a congestion tree, the “root” is the point where the congested flows finally meet, the “branches” are series of consecutive congested points along any path followed by congested flows, and the “leaves” are the points at the extreme of each branch. [Figure 1c](#) shows the final congestion tree that is formed from the congestion situation shown in the previous examples.

Note that the congestion tree shown in [Fig. 1c](#) is a very simple example, since congestion trees may exhibit very complex dynamics. In fact, congestion trees may actually evolve in very different ways, depending on traffic patterns, routing schemes, and switch architecture, as shown in [9]. For example, a congestion tree may grow from leaves to root and vice versa. Also, several congestion trees may grow independently and later merge, and it is even possible that some trees completely overlap while being independent.

Congestion Impact on Network Performance

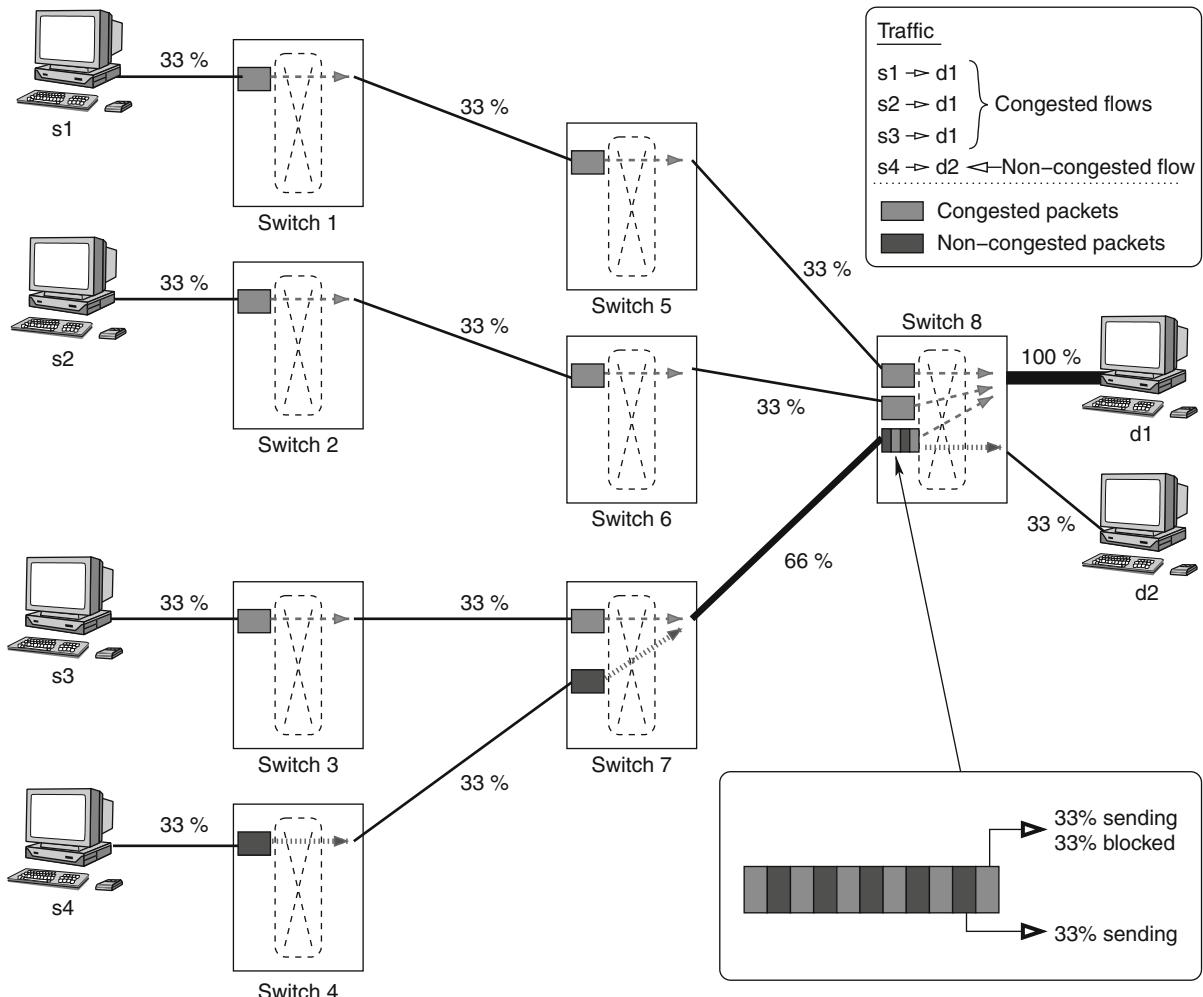
It is a well-known fact that the presence of congestion trees in a network may dramatically degrade network performance. Specifically, it has been exhaustively

reported that congestion leads to a decrease in network throughput and to an increase in packet latency. However, if the simple congestion tree described in the previous example is considered, it is not obvious that it could produce these effects in the network. Note that, in fact, the congested output port in the example could forward packets at maximum rate, and congested packets advance at the maximum possible speed for that specific traffic load. Thus, if the presence of congestion trees does not directly imply any negative effect, which is the actual cause of network performance degradation under congestion situations?

The answer to the previous question is that congested flows are actually dangerous when they share network resources (buffers, links) with other, non-congested flows. In these cases, the final effect is non-congested flows advancing at the same speed as congested ones, thereby increasing the latency of non-congested packets and decreasing overall network throughput.

[Figure 2](#) shows an example of this “real” negative effect of congestion trees. In this example, it is assumed that four sending end nodes (sources s1, s2, s3, and s4) inject packets at the maximum speed allowed by the network at any time, while there exist two receiving end nodes (destinations d1 and d2). As can be seen, three sources (s1, s2, and s3) inject packets addressed to the same destination (d1), forming a congestion tree whose root is at the output port leading to d1. Although there exists another flow in the network (from s4 to d2), it is a non-congested flow, since packets belonging to this flow do not request to cross the root of the tree. Since the three congested flows that create the tree meet at Switch 8, they must share the bandwidth of the link that connects the root point to d1. Therefore, assuming a fair arbitration from the switch scheduler, each congested flow will cross Switch 8 at a speed equal to one third (33%) of the maximum link rate. Moreover, due to the spreading of congestion by means of flow control, all the packets along the congestion tree branches will advance at the same speed, and finally even the sources of these packets will be forced to slow injection until this rate. The figure indicates link utilization percentages when the network reaches this “stable” state.

Note that the utilization of the link that connects the hot-spot destination (d1) to the network is maximum (100%), so it is not possible to improve network



Congestion Management. Fig. 2 Congestion tree causing Head-Of-Line (HOL) blocking to a non-congested flow

throughput at this point. In fact, network throughput would be the maximum if only congested flows exist. As mentioned above, this means that the mere existence of the congestion tree does not justify by itself the network throughput degradation observed in congested networks. On the contrary, note that the link leading to d_2 is used only at 33% of the maximum link rate. However, there is no contention for this link, since only one flow requests to access it. So, why is not this link used at full rate? This is because the non-congested flow requesting this link follows a path partially affected by the congestion tree, in such a way that packets belonging to the non-congested flow share a link and a queue with congested flow packets. Due to this sharing, non-congested flow packets advance at the same speed

as the congested flow ones. Note that, without this interaction between congested and non-congested packets, the link leading to d_2 would be used at maximum link rate.

Therefore, the cause of network performance degradation associated to congestion is that packets belonging to congested flows may prevent other flows from advancing at the injection rate, even if they belong to non-congested flows. Actually, this effect is a generalization of the phenomenon known as Head-Of-Line (HOL) blocking, which, in general, happens whenever a packet ahead of a First In, First Off (FIFO) queue blocks, preventing the rest of packets in the same queue from advancing. In fact, congested packets may produce HOL blocking to non-congested ones in any network point

where both types of packets share network resources. Of course, the wider the congestion tree, the greater the probability of HOL blocking caused by congested packets.

In conclusion, the performance degradation (throughput decreases, latency increases) that networks suffer in the presence of congestion trees is due to the HOL blocking that congested flows cause to non-congested ones when they share network resources. As explained in the following, some congestion management strategies are based on that principle, while others deal with congestion without explicitly considering HOL blocking.

Congestion Management Strategies

As mentioned in the introduction, congestion management is currently an essential issue in the design of the interconnection network of modern parallel computing systems. In fact, it has been a popular research topic for many years, thus existing many approaches to solve the problems related to congestion. The most relevant of these approaches are analyzed below. As several specific strategies that differ in some aspects or details may share the same basic approach, different strategies may appear in the following grouped into the same category, although independently described when necessary. Note, however, that taxonomies of congestion management techniques different than the one shown here are perfectly possible. Note also that some of the strategies described below are currently obsolete or unfeasible in modern interconnect technologies, but they have been considered in order to offer a wide overview of both past and present strategies.

Packet Dropping

This approach, as it has already been explained, consists in discarding packets when congestion occurs. Ideally, the discarded packets should be those actually contributing to congestion, i.e., packets belonging to congested flows. Obviously, if congested packets are discarded, congestion trees would disappear or even would never form, depending on the criteria for detecting congestion. Thus, theoretically, this approach would straightforwardly solve the problems congestion may produce.

However, packet dropping presents some drawbacks that make it not suitable for current high-performance

interconnection networks. First, it is quite difficult to implement this strategy without discarding packets belonging to non-congested flows. For instance, in a queue shared by congested and non-congested packets, instead of just emptying the buffer at once, it would be necessary to perform some kind of “selective discarding” in the usual FIFO queues, thus introducing significant complexity and probably an additional delay. Second, and more important, for most parallel applications packets that are discarded must be retransmitted, thus increasing final packet latency. Note that in these cases non-congested packets may experience a delay similar (or even worse) to the one they suffer in the presence of congestion trees; thus, this approach may produce the same effect that the phenomenon it tries to avoid!

Summing up, the congestion detection and packet dropping processes, and the associated re-transmissions may lead to some packets experiencing a very high latency, which may have a direct impact on application execution time in a parallel computer. In particular, parallel applications with Quality of Service (QoS) requirements would be very sensitive to this effect. For that reason, this congestion management approach is invalid for the interconnection networks of current parallel computing systems. In fact, modern high-performance interconnect technologies do not allow packet dropping (they are “lossless” networks). However, packet dropping is allowed in other environments (for instance, computer networks like the Internet) where latency requirements are not strict at all.

Network Overdimensioning

Overdimensioning the network consists in using many more network components than the minimum required for connecting all the system end nodes, with the aim of making the network to operate well below the saturation point, thus keeping low link utilization and thus reducing congestion probability. Obviously, a network offering much more bandwidth than the required by the applications is unlikely to suffer congestion situations.

This was a valid approach for parallel systems some time ago, when network utilization in parallel machines and clusters was low. However, overdimensioned networks are becoming inappropriate for current parallel systems due to cost and power consumption constraints. On the one hand, current interconnect

components are very expensive when compared to processors. Thus, the network represents a high percentage of the total system cost. On the other hand, as VLSI technology advances and link speed increases, interconnects are consuming an increasing fraction of the total system power. Moreover, current high-speed links require continuous pulse transmission in order to keep both ends synchronized, even when no data are being transmitted. As a consequence, link power consumption is almost independent of link utilization. Note that dynamic frequency/voltage scaling techniques could be used in order to reduce power consumption, but unfortunately the efficiency of these proposals is not completely satisfactory due to their slow response against traffic variations and the suboptimal frequency/voltage settings during transitions. Anyway, even if these techniques were more efficient, they would not solve the cost problem.

Therefore, a trend in current network design is to reduce the number of networks components with the aim of reducing both system cost and power consumption. Obviously, if the system computational power must be maintained, then the only ways to reduce the number of network components are to increase the number of end nodes attached to each switch (note that most current interconnect technologies allow that) or to use a more suitable network topology. However, any of these solutions lead to a higher level of link utilization, thereby driving the network to work closer to its saturation point, and thus increasing congestion probability.

Summing up, overdimensioning the network is practically unfeasible in current parallel systems, which in fact are prone to suffer congestion situations due to the low number of components used to build their interconnection networks, thus requiring specific, efficient congestion management mechanisms.

Avoidance-Based Strategies

This category includes all the techniques based on planning the use of network resources, in order to find a schedule, which guarantees that congestion never appears. Following this schedule, the network resources required by each data transmission are reserved before starting it, thus avoiding congestion situations. Some of these techniques are software-based, while others are hardware-oriented.

Note that these strategies need to know in advance both the resource requirements of each transmission and the occupancy of network resources, in order to obtain the optimal resource reservation schedule. However, this knowledge is not always available. Besides, resource reservation incurs significant overhead. Thus, this kind of technique is not suitable for general-purpose congestion management. In fact, these strategies are strongly related to QoS provision, and most of them have been proposed in that sense, thereby only implicitly addressing congestion. An example of these strategies is the one proposed in [15].

Prevention-Based Strategies

Prevention-based strategies control the traffic in such a way that congestion trees should not happen. In general, decisions are made “on the fly,” based on limiting or modifying routes (or memory accesses) with the aim of preventing the appearance of congestion situations, as proposed in [11].

Like avoidance-based strategies, prevention-based techniques need a knowledge of network status in order to obtain the optimal traffic schedule. Consequently, their drawbacks are basically the same as the aforementioned ones for the former category. Analogously, these techniques can also be software-based or hardware-oriented.

Note that the differences between avoidance-based and prevention-based strategies are quite subtle, and, in fact, in some taxonomies both types are grouped into the same category. In this case, all these strategies are usually referred to as “proactive” strategies, because they all try to solve congestion before it appears.

Reactive Techniques

Reactive (or detection-based) techniques are based on detecting the appearance of congestion, in order to activate some control mechanism that should eliminate any possible congestion tree in the network. Note that in this case, in contrast with the two previous approaches, congestion situations are actually allowed to happen, assuming that the control mechanism will finally solve the problem.

This basic approach has been followed by many proposals, which mainly differ in two aspects: the congestion detection criteria and the mechanism in

charge of eliminating congestion. Regarding congestion detection, some feedback information is usually required. For instance, some proposals locally measure the occupancy of buffers in the switches, while other techniques (especially in multiprocessor systems) monitor the amount of memory access requests.

Regarding congestion elimination mechanisms, most of them involve notifying the end nodes of the congestion so that they throttle the injection of packets or they cease (or reduce) memory access requests. Depending on which end nodes receive congestion notifications, reactive techniques can be divided into three subcategories:

1. Congestion notifications are broadcast to all the end nodes.
2. Notifications are sent only to end nodes contributing to congestion.
3. Notifications are sent only to end nodes attached to the switch where congestion is detected.

It is obvious that if the first (broadcast) notification policy is implemented, a large fraction of the offered network bandwidth is consumed for sending notifications, thus inefficiently using network resources. Another serious drawback, common to the first and second notification policies, is the lack of scalability, as in these cases reaction time is directly proportional to the distance from the point where congestion is detected to the traffic sources. Therefore, if network size and/or link bandwidth increase, the effective delay between congestion detection and reaction increases linearly. Note that during this time, a high number of packets contributing to congestion could be injected, in such a way that, by the time congestion notifications reach the sources, big congestion trees may have already been formed. All that leads to slow response and the typical oscillations that arise in closed-loop control systems with delays in the feedback loop, thus the first two notification policies may be not appropriate for many networks, especially for those with long round trip times (RTTs). The third notification policy (used, for instance in [2]), however, does not present the described scalability problems, but, unfortunately, it fails if the sources contributing to congestion are not attached to the switch where congestion is detected. As this is perfectly possible, the efficacy of the third policy would vary depending on congestion

trees' shape (which, as mentioned above, depends on several factors).

HOL Blocking Elimination Techniques

While the aforementioned strategies try to eliminate, reduce, or delay congestion itself, other techniques focus on eliminating the actual cause of performance degradation under congestion situations: The HOL blocking. In fact, note that, if the HOL blocking produced by congested packets is eliminated, congestion becomes harmless. Most of the existing HOL blocking elimination proposals are based on having different queues at each switch port, in order to separately store packets belonging to different flows, thus lowering HOL blocking probability. Note that congestion trees would not be eliminated but isolated as much as possible, in order to avoid interaction with non-congested flows. This is the common basic approach followed by several techniques which, on the other hand, differ in many aspects, basically in the number of queues required, in the policy for mapping packets to queues, and in queue management.

For instance, a well-known HOL blocking elimination technique is Virtual Output Queues (VOQs) [3], which requires at each port as many queues as destinations in the network. This scheme allows that, at each port, all the packets addressed to a specific destination are exclusively stored in the queue assigned to that destination, and never share that queue with packets addressed to other destinations, thus completely eliminating HOL blocking. However, as the number of queues per port grows with the number of destinations, this scheme does not scale with network size, becoming unfeasible in medium or large networks (note that each queue requires a minimum silicon area to be implemented).

In order to overcome this problem, a variation of this scheme uses as many queues at each port as output ports in the switch [1], and each incoming packet is stored in the queue assigned to its output port. This scheme is usually referred to as VOQ at switch level (VOQsw), in contrast to the former scheme, that may be named as VOQ at network level (VOQnet). VOQsw scales with network size, and it eliminates HOL blocking in a switch if it is directly caused by packets contending for output ports in the same switch. On the contrary, in switches affected by congestion

“spreading” from other switches, VOQsw cannot guarantee that congested packets do not share queues with non-congested packets, thus VOQsw eliminates just partially HOL blocking. The Dynamically Allocated Multi-Queues (DAMQs) technique [14] uses the same queue scheme, although in this case the size of queues may dynamically vary when required. Virtual Channels [4] may also reduce HOL blocking at the switch level but do not eliminate it.

Another scalable solution is the Destination-Based Buffer Management (DBBM) strategy [13]. This proposal also uses a reduced set of queues at each port, but in this case packets are assigned to queues according to a modulo-mapping function: a packet with destination D is stored in the queue whose number is $D \bmod N$, where N is the number of queues in a port. As a result, a set of nonconsecutive destinations are assigned to each queue, thus packets addressed to a congested destination produce HOL blocking only to packets addressed to destinations in the same set. Dynamic Switch Buffer Management (DSBM) also assigns sets of destinations to queues, but in this case it assigns depending on queue occupancy. Note that these solutions, as VOQsw, eliminate HOL blocking just partially, as congested packets are allowed to share queues with non-congested ones.

In general, for all the aforementioned HOL blocking elimination techniques, the greater the number of queues per port, the more HOL blocking is eliminated. This is because congested packets are not explicitly identified, and consequently the probability of isolating them grows with the number of queues. Moreover, note that packets are assigned to queues following “static” criteria, independently of network status.

By contrast, other solutions detect congestion in order to explicitly identify congested packets, later isolating them in dynamically allocated queues. Note that these queues would only store packets belonging to congested flows (i.e., those packets that could produce HOL blocking), and if these flows vanish, the corresponding queues could be deallocated and later reallocated for new congested flows. In this way, these techniques are able to eliminate HOL blocking without relying on an unaffordable number of queues, thereby achieving scalability. In that sense, the solution proposed in [12] for the ATLAS I assigns queues to isolate packets addressed to congested end nodes. However, in many cases congestion arises inside the network, thus in these cases

this solution would not accurately detect congestion and it would not efficiently eliminate HOL blocking. On the contrary, other proposals dynamically assign queues to isolate packets addressed to congested points, either inside the network or end nodes. This is the case for the Regional Explicit Congestion Notification (RECN) [6] and the Flow-Based Implicit Congestion Management (FBICM) [7] techniques, which probably achieve the most effective HOL blocking elimination that can be obtained with a reduced set of queues. RECN has been proposed for technologies using source routing, while FBICM for deterministic distributed routing networks. Note that both strategies require control memories at ports in order to keep track of congestion information and to manage the dynamic queues (specifically, CAM memories are used). Additionally, special control messages exchanging congestion information between switch ports are necessary. Summing up, these two strategies are very effective and scalable, but their implementation is not simple.

Other Approaches

Other strategies that may help to alleviate congestion or delay its appearance are the use of fully adaptive routing [5] or load balancing techniques [8]. In order to be actually effective against congestion, these strategies should take into account network status for making routing decisions, as the technique proposed for Networks-on-Chip (NoCs) in [10] does. Note, however, that these techniques delay the appearance of congestion, but cannot avoid network performance degradation if congestion is reached (especially under heavy traffic loads). Moreover, adaptive routing may cause out-of-order packet delivery, which is unacceptable for some parallel applications.

Future Directions

Despite many proposals for solving the problems related to congestion in interconnection networks, congestion management can be considered still an open issue. On the one hand, some solutions cannot guarantee that network performance will not suffer a significant degradation in congestion situations. On the other hand, other solutions minimize the impact of congestion but are difficult to implement because they either consume too many network resources, or require expensive additional resources, or increase switch complexity. As concerns about network components’ cost and

power consumption are not likely to disappear, cost-effective approaches will probably be the most popular ones in the future.

In that sense, as mentioned above, the strategies currently offering the best relationship among efficacy, feasibility, and scalability are probably the ones based on eliminating HOL blocking, especially those that dynamically assign queues to isolate congested packets. Therefore, it make sense that future proposals about congestion management should try, at least, to improve the efficacy and/or cost-effectiveness of such strategies, although it does not necessarily imply they should follow the same approach to solve the problems related to congestion. In particular, in some emerging technologies like NoCs, where silicon area and power consumption constraints are quite strong, most existing solutions are not suitable at all, thus new approaches should be proposed to deal with congestion in these environments.

Related Entries

- [Interconnection Networks](#)
- [Routing \(Including Deadlock Avoidance\)](#)
- [Switch Architecture](#)
- [Switching Techniques](#)

Bibliographic Notes and Further Reading

As can be drawn from the number and diversity of the aforementioned strategies, there is a huge research body in the congestion management area (although, as mentioned above, not all the strategies are suitable for the interconnection networks of modern parallel systems). Moreover, new proposals focused on or related to congestion are not currently infrequent in journals and conferences. Of course, this is a consequence of both the serious impact of congestion situations in network performance and the aforementioned difficulties for achieving a truly efficient congestion management. Taking all that into account, most of the references included below in the bibliography are just a small selection from a vast number of works proposing strategies to solve the problems related to congestion. Note, however, that there exist few studies, like the one shown in [9], devoted to the complex dynamics of the congestion phenomenon in interconnection networks.

Bibliography

1. Anderson T, Owicki S, Saxe J, Thacker C (1993) High-speed switch scheduling for local-area networks. *ACM Trans Comput Syst* 11(4):319–352
2. Baydal E, Lopez P (2003) A robust mechanism for congestion control: INC. In: Proceedings of the 9th international euro-par conference, Klagenfurt, 2003
3. Dally WJ, Carvey P, Dennison L (1998) The avici terabit switch/router. In: Proceedings of the Hot Interconnects 6, Stanford, 1998
4. Dally WJ (1992) Virtual-channel flow control. *IEEE Trans Parallel Distrib Syst* 3(2):194–205
5. Duato J (1993) A new theory of deadlock-free adaptive routing in wormhole networks. *IEEE Trans Parallel Distrib Syst* 4(12):1320–1331
6. Duato J, Johnson I, Flich J, Naven F, Garcia PJ, Nachiondo T (2005) A new scalable and cost-effective congestion management strategy for lossless multistage interconnection networks. In: Proceedings of the 11th international symposium on high-performance computer architecture (HPCA05), San Francisco, 2005
7. Escudero-Sahuillo J, Garcia PJ, Quiles FJ, Flich J, Duato J (2008) FBICM: efficient congestion management for high-performance networks using distributed deterministic routing. In: Proceedings of the 15th international conference of high performance computing, Bangalore, 2008
8. Franco D, Garces I, Luque E (1999) A new method to make communication latency uniform: distributed routing balancing. In: Proceedings of the ACM international conference on supercomputing, Rhodes, 1999
9. Garcia PJ, Flich J, Duato J, Johnson I, Quiles FJ, Naven F (2005) Dynamic evolution of congestion trees: analysis and impact on switch architecture. *Lect Notes Comput Sci* (HiPEAC-2005), 3793:266–285
10. Gratz P, Grot B, Keckler SW (2008) Regional congestion awareness for load balance in networks-on-chip. In: Proceedings of the 14th international conference on high-performance computer architecture (HPCA-14), Salt Lake City, 2008
11. Ho WS, Eager DL (1989) A novel strategy for controlling hot spot contention. In: Proceedings of the international conference parallel processing I, pp 14–18, St. Charles, 1989
12. Katevenis M, Serpanos D, Spyridakis E (1998) Credit-flow-controlled atm for mp interconnection: the atlas I single-chip atm switch. In: Proceedings of the 4th international symposium on high-performance computer architecture, Las Vegas, 1998
13. Nachiondo T, Flich J, Duato J (2005) Efficient reduction of HOL blocking in multistage networks. In: Proceedings of the 19th international parallel and distributed processing symposium (IPDPS 2005), Denver, 2005
14. Tamir Y, Frazier GL (1992) Dynamically-allocated multi-queue buffers for VLSI communication switches. *IEEE Trans Comput* 41(6):725–737
15. Yew P, Tzeng N, Lawrie DH (1987) Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Trans Comput* 36(4):388–395

Connected Components Algorithm

► Graph Algorithms

Connection Machine

GUY L. STEELE JR.
Oracle Labs, Burlington, MA, USA

Definition

The term “Connection Machine” refers to massively parallel supercomputers manufactured by Thinking Machines Corporation. The CM-1 (1985) was a purely SIMD architecture intended for artificial intelligence (AI) applications; 65,536 single-bit processors were connected by a hypercube network. The CM-2 (1987) added 32-bit-wide floating-point coprocessors and set performance records for numerical applications. The CM-5 (1991) was MIMD, using Sun Microsystems SPARC processors connected by a fat-tree network to drive proprietary SIMD floating-point coprocessors. The first TOP500 Supercomputer List (June 1993) showed that the four fastest supercomputers were CM-5 systems, and 25 of the top 100 were either CM-2 or CM-5 systems.

Discussion

Origins

Influential predecessors of the initial Connection Machine architecture include the Illiac IV, the ICL Distributed Array Processor (DAP), and the Goodyear Massively Parallel Processor (MPP). Certain other early designs also merit discussion.

While the Illiac IV [3, 7] never met its original design goals, it was for a time the world’s fastest supercomputer and used to solve important real-world scientific problems, and so it may be regarded as the first successful parallel supercomputer. Several designers and implementors who developed language compilers for the Illiac IV later developed similar compilers for the Connection Machine. The Illiac IV had a SIMD architecture, with a central control unit that issued instructions to many processing elements – 256 as originally

designed, though only one “quadrant” of 64 processors was delivered to NASA. (The first Connection Machines had a similar SIMD architecture, though with thousands of processors rather than dozens, and were likewise divided into quadrants.) The processing elements of the Illiac IV were connected into a square two-dimensional grid such that each processing element could communicate with its four nearest neighbors.

There is an engineering trade-off between the processing element size and the total number of processing elements. Each Illiac IV processing element was a full-fledged floating-point unit. In contrast, many other early parallel machine designs were SIMD architectures with much larger numbers of processing elements, each just 1 bit wide, all simultaneously executing an instruction stream broadcast by a central controller.

As early as 1958, Unger proposed a SIMD design with 1-bit processors having only a few bits of state apiece, for purposes of “spatial problems” such as visual pattern recognition. This design was never constructed, but was simulated on a (conventionally sequential) IBM 704 computer [60]. The network connecting the processors was a two-dimensional, eight-nearest-neighbors grid, but for some purposes interprocessor links could be “enabled” or “disabled” and then data could be transferred between distant processors in one step through chains of enabled links. Unger (correctly) envisioned the need for such a machine to have many hundreds of processors in order to be useful, and commented “this means thousands of memory elements and tens of thousands of gate inputs in the matrix alone. These are alarming figures ... However, progress in the components field is such that it is reasonable to hope that within a few years there may be available manufacturing processes whereby entire blocks of logical circuitry could be constructed in one unit.... While the author has specifically discussed a computer based on a rectangular matrix of modules using one-bit registers, it might be worthwhile to consider other arrangements. Possible variations include matrices in three or more dimensions, registers enlarged to handle multi-bit words, and polar co-ordinate arrays.”

The SOLOMON computer [19, 48] was designed to connect up to 2,048 processors into a 32×64 two-dimensional, four-nearest-neighbors grid though smaller sizes such as 32×32 or even 32×8 were envisioned (and apparently the only hardware actually

constructed was a demonstration system much smaller than this).

The Illiac III [35] was designed for visual pattern recognition and had multiple independent computational units, one of which was a Pattern Articulation Unit containing 1,024 SIMD processors arranged as a 32×32 two-dimensional, eight-nearest-neighbors grid.

The Staran [4] had multiple array modules of 256 processors each and a rather complex communication network [5] capable of permuting 256 bits in many ways.

The ICL DAP [39] had 4,096 processors connected into a 64×64 two-dimensional, four-nearest-neighbors grid.

The Goodyear MPP [6] had 16,384 processors connected into a 128×128 two-dimensional, four-nearest-neighbors grid.

Supercomputer projects whose gestation was roughly contemporaneous with that of the first Connection Machine designs include the NYU Ultracomputer [18], a MIMD architecture with a “fetch-and-add” network; NON-VON [17, 43], with thousands of 1-bit processors connected into a complete binary tree, intended to process massive quantities of data kept on secondary storage; and the Caltech Cosmic Cube [40], with 64 conventional microprocessors connected by a six-dimensional hypercube network (however, the notion of connecting thousands of processors using a hypercube network goes back at least to 1963 [49]).

Many of the software techniques, such as bit-serial arithmetic, developed and used on earlier massively parallel machines with 1-bit processors were directly applicable to the Connection Machine. The Connection Machine model CM-1 was distinguished from these predecessors by its inclusion of a hypercube network that provided much faster communication between distant processing elements and much greater overall network bandwidth (and by being the first massively parallel architecture to be put into commercial production, with multiple copies sold to multiple customers). The Connection Machine model CM-2 introduced hardware floating-point coprocessors that allowed it to be programmed in some ways more like the Illiac IV than like the Goodyear MPP. The Connection Machine model CM-5 represented a radical departure, with an entirely new architecture that was MIMD, used conventional microprocessors (optionally augmented with vector coprocessors), and had a network topology that

was not a hypercube; in some ways it resembled the NYU Ultracomputer.

Connection Machine Model CM-1

A full Connection Machine model CM-1 consisted of $2^{16} = 65,536$ processors [59], each having:

- An arithmetic-logic unit (ALU) and associated latches
- $2^{12} = 4K$ bits of bit-addressable local memory
- Eight 1-bit flag registers
- Router interface
- NEWS grid interface

Each ALU was an almost trivial piece of circuitry: a 3-input, 2-output logic element that could compute any two Boolean functions of three inputs. The functions were specified by two 8-bit truth tables, so in effect the ALU consisted simply of two 1-of-8 selectors controlled by the three input bits. The three inputs were any two bits from local memory and any of the eight flags; of the two outputs, one was written back to memory and one was written to a specified flag. Finally, the entire operation was conditional on yet another specified flag; if its value was 0, then the two results were not written after all.

Using this ALU, all data processing was carried out bit-serially within each processor. One of the original inspirations for the Connection Machine was Scott Fahlman’s PhD work on semantic networks and computation within such networks via marker propagation. Fahlman himself had sketched an idea for a hardware implementation [16]. W. Daniel “Danny” Hillis refined these ideas in his own PhD work at MIT [21], and in particular designed a more efficient and more easily controlled network for connecting the processors [22]. Each processor might then represent one node in a semantic network and contain the addresses of a moderate number of other processors; the network could then be used to propagate markers, each consisting of a single bit or a small number of bits, among the processors. The CM-1 ALU design was, however, adequate to implement a full adder, and so it was reasonably convenient to implement addition and subtraction in a bit-serial fashion, and then multiplication, division, and all other arithmetic operations of interest.

Consider, for example, conditional addition of two signed integers, each k bits long. The following steps

were carried out simultaneously within each hardware processor. First a bit was loaded from memory into a flag bit to control whether the results of the addition should be stored for that processor. Next a second hardware flag was cleared for use as a carry bit. Next came k iterations of an ALU operation that read one bit of each operand from memory as well as the carry bit from the hardware flag, computed the sum (a three-way exclusive OR) and carry-out (a three-input majority function), and stored the sum back into memory and the carry-out back into the hardware flag. These iterations started with the least significant bits of the operands and proceeded toward the most significant bits. The sum could replace one of the input operands or be stored into a third area of memory. The last of the k iterations stored the carry-out into a third hardware flag rather than the second one; a further computation that compared the second and third flags (the carry-outs from the last two iterations) could then determine whether integer overflow occurred.

This bit-serial strategy was not original with the Connection Machine; earlier machines that performed all arithmetic in bit-serial fashion include the Goodyear MPP supercomputer [6] and the Digital Equipment Corporation PDP-8/S minicomputer [13]. The CM-1 could perform one ALU iteration for integer addition in about half a microsecond; taking instruction decoding and initialization overheads into account, a 32-bit integer add required about 21 μ s. But with 2^{16} processors performing such additions simultaneously, this produced an aggregate rate of 2,500 MIPS (i.e., 2.5 billion 32-bit integer additions per second). By way of comparison, the earlier Cray-1 vector supercomputer (1975) had a peak speed of 160 MFLOPS when performing multiply-add operations on 64-bit floating-point operands [9], and the initial model of the Cray X-MP (1984) had a peak speed of 210 MFLOPS per CPU, where a system could have up to two (1982) or four (1984) CPUs [10, 11]. While no one expected the bit-serial arithmetic of the CM-1 to outperform Cray supercomputers on floating-point computation, it was clear that the CM-1 could be competitive for integer computations and perhaps supremely successful for marker propagation algorithms.

(It is instructive, however, to do a back-of-the-envelope calculation for the speed of floating-point arithmetic carried out on such a bit-serial processor. For

floating-point addition, the input significands must be aligned and the output sum must be normalized; for m -bit significands, the cost is roughly $m[\log_2 m]$ ALU cycles. The addition itself requires m ALU cycles. For 32-bit floating-point operands with $m = 24$, this comes to about 250 ALU cycles; for 64-bit operands with $m = 53$, it is about 700 ALU cycles. For floating-point multiplication, the cost is roughly m^2 ALU cycles, plus integer addition of the exponents, plus a 1-bit normalization step. For 32-bit floating-point operands with $m = 24$, this comes to about 650 ALU cycles; for 64-bit operands with $m = 53$, it is about 2,900 ALU cycles. If an ALU cycle is about half a microsecond, and one assumes a mix of floating-point operations that is half adds and half multiplies, then the estimated speed is roughly 275 MFLOPS for 32-bit operations and 73 MFLOPS for 64-bit operands – not quite Cray-1 class, but still fast enough to be interesting.)

The real innovation in the Connection Machine was its *router*, which passed message packets bit-serially among processors. While the network is often described as a 16-dimensional hypercube, it should be noted that the processors were packaged 16 to a chip. The $2^{12} = 4,096$ chips were connected by a 12-dimensional hypercube of actual physical wires, and additional multiplexing was done on-chip. Routing of data among chips made use of some of the same hardware flags and memory paths as the ALU operations. Each processor that needed to send a message (possibly all of them) would first load 16 bits from memory, one at a time, representing the address of a destination processor, and then n bits of data to be sent. The router on each chip could accept up to four messages at a time from processors on that chip, and would respond to each sending processor with a bit saying whether its message had been accepted; processors whose messages were not accepted on this routing cycle would try again on the next routing cycle.

Message bits traversed the hypercube in pipelined fashion; at each of 12 steps, each accepted message would have the opportunity to traverse one of the 12 hypercube dimensions. (A message would need to traverse a wire along a given dimension if and only if the address of the destination processor differed in that bit position from the address of the originating processor.) After those 12 steps had processed the 12-bit destination address in each message, data bits would follow the address bits serially along the paths that had been

established through the hypercube. As data bits arrived at a destination processor, they were then stored one at a time. For sufficiently long messages, on each routing iteration a bit would be loaded from memory to be sent out and a newly arrived bit (from an earlier position in the message) would be stored back into memory; toward the end of the process, the final steps would simply store the last arriving bits.

Two or more messages might compete for the same hypercube wire, in which case one message would get the use of the wire, and the others would not. These others would still have the opportunity to traverse other hypercube dimensions, but would not reach their destinations during the current routing cycle; instead, they would be stored into temporary buffers within the memory of whatever processors they had managed to arrive at, and then forwarded by another round of routing. A more subtle point is that, at each of the 12 dimension steps, each router had to be prepared to accept a message on the incoming dimension wire. Therefore, if the router buffers on some chip were all full and *none* of those messages needed to traverse the outgoing dimension wire, then one message was chosen and sent out on that dimension anyway – and therefore that particular message would be unable to reach its destination in the current routing cycle – just to make sure that a buffer would be free to accept an incoming message (this move was called *desperation routing*). Software would then repeat the routing process – forwarding messages that had not reached their destinations because of wire competition or desperation routing, and accepting messages that had not been previously accepted – until all messages eventually reached their destinations [20, 22].

The router allowed any processor to establish a data connection with any other processor with reasonable speed and efficiency – from this idea came the very name “Connection Machine.” Compare this to the Illiac IV [3, 7] or the Goodyear MPP [6], each of which connected processors in a two-dimensional grid, each processor able to communicate directly with only its four nearest neighbors: Transferring data to a distant processor could take quite a long time. On the other hand, it was also clear that a two-dimensional grid could be particularly attractive and efficient for certain applications such as image processing. Therefore the CM-1 had a second network, completely independent from

the hypercube, called the NEWS (North/East/West/-South) network, that allowed a processor to pass a single data bit to any of four nearest neighbors without the overhead of first loading a destination address.

Instructions were provided to the 65,536 processors from a central source through a two-step process. The CM-1 was not a stand-alone system, but a coprocessor that was always attached to a front-end computer (a Symbolics 3600 Lisp Machine). Programs running within the front-end computer would issue instructions, one at a time, to the Connection Machine. These instructions constituted an abstract architecture that supported integer and floating-point arithmetic, other operations on multi-bit operands, message-passing, reduction operations such as finding the sum or maximum of a collection of values, and most importantly *virtual processors*, by which the memory of each hardware processor was split into m regions, and each hardware processor was time-sliced so as to simulate m processors. By the time of the CM-2, this instruction set had been given a name: Paris, for PARallel Instruction Set. Paris instructions sent from the front-end processor arrived at a microcoded *sequencer* within the Connection Machine. Two networks (over and above the hypercube and the NEWS network) connected the sequencer to the Connection Machine processors: A *broadcast network* allowed SIMD *nanoinstructions* to be sent from the sequencer to the processors, and a *combining* network took a 1-bit signal from every processor and delivered the logical OR of these 65,536 signals to the sequencer. Finally, the sequencer had the ability to read or write any 32-bit word within the memory of any Connection Machine processor.

A single CM-1 nanoinstruction specified one sub-cycle of an ALU or router operation. A nanoinstruction contained a 3-bit operation code, a 3-bit flag number, a 12-bit memory address, and one 8-bit truth table for the Connection Machine ALU. As an example, the basic ALU iteration for the integer add operation described above would consist of three nanoinstructions:

- LOADA: read memory operand A, read flag operand, latch one truth table
- LOADB: read memory operand B, read conditional flag, latch other truth table
- STORE: store one result bit in memory, store other result bit in flag

The nominal clock speed of the CM-1 was 6 MHz [59], so such an ALU cycle would take about half a microsecond.

In order to allow large machines to be shared among multiple users and to allow a range of machine sizes to be offered for sale, there was actually one sequencer for every 16,384 processors. A 4×4 crossbar switch called the *nexus* allowed up to four front-end computers to be connected to a single CM-1 system, and such a system could consist of 1, 2, or 4 *quadrants*. These quadrants could be dynamically assigned to front-end processors for their use. If a front-end processor were assigned more than one quadrant, then Paris instructions issued by that front-end processor were broadcast by the nexus to multiple sequencers, and in all other ways the ganged quadrants were made to behave as if under the control of a single sequencer.

The cabinetry of the CM-1 (which was also used for the CM-2) had a striking physical design: It looked like a “cube of cubes” that was 56 inches on a side and was intended to evoke the hypercube topology of the router network (see Fig. 1). Each of the eight subcubes was about 26 by 26 by 26 in. and contained 16 vertically oriented printed circuit boards connected by a large square backplane. Each printed circuit board

held 32 processor chips of 16 processors each and had 32 red light-emitting diodes mounted on the edge opposite the backplane connectors [23, p. 110]. These lights were visible through translucent panels on the front and back of the cabinet; thus 2,048 lights were visible on the front and 2,048 on the back. These lights could be used for diagnostic reporting purposes, but also served to give the cabinet a visual “wow factor” intended to communicate to spectators that there was a lot going on inside that otherwise static-looking black cabinet [52]. (Indeed, software was eventually written specifically to produce good-looking patterns for still photos and for video recording.) The cabinet could be physically split apart into four quadrants (each consisting of two vertically stacked subcubes) for shipping, and systems were offered in three sizes, consisting of one, two, or four quadrants.

Most software for the CM-1 was coded in *Lisp, which was a package of functions that extended Symbolics Zetalisp to provide access to the Connection Machine. If the sequencer was instructed to simulate some number n of virtual processors, then *Lisp allowed the user to allocate certain data structures within the CM-1 that behaved like vectors of length n that could perform various elementwise operations simultaneously on all elements, as well as finding the largest or smallest value, the sum of all values, permuting the elements according to a vector of indices, and so on. A more elaborate dialect of Lisp called Connection Machine Lisp (CM-Lisp) was also designed and prototyped, but never became a full-fledged product. Both were aimed at supporting the programming of data parallel algorithms [24].

One example of an artificial intelligence application explored on the CM-1 was memory-based reasoning [51]. Examples of non-AI applications implemented on the CM-1 include circuit simulation [61], ray-tracing [12], and free-text search [50].

Connection Machine Model CM-2

The CM-2 was a fairly compatible revision of the CM-1 architecture and design [53, 54, 59]. A full Connection Machine model CM-2 similarly consisted of $2^{16} = 65,536$ processors, each consisting of:

- An arithmetic-logic unit (ALU) and associated latches



Connection Machine. Fig. 1 Connection Machine model CM-2 with DataVault and graphic display

- $2^{16} = 64\text{ K}$ (and later $2^{18} = 256\text{ K}$) bits of bit-addressable local memory
- Four (rather than eight) 1-bit flag registers
- Router interface
- NEWS grid interface
- I/O interface
- Floating-point accelerator interface
- Improved error-detection circuitry, including single-error correction, double-error detection (SECDED) for memory

The four biggest changes in the CM-2 hardware from the CM-1 were the floating-point accelerators, the NEWS grid implementation, in-router combining of messages, and the I/O interface. The CM-2 also had a slightly higher clock rate, 7 MHz [36].

A CM-2 could be built with or without floating-point accelerators: For every 32 Connection Machine processors (i.e., for every pair of Connection Machine processor chips) there was a proprietary custom interface chip and an off-the-shelf floating-point coprocessor chip (the Weitek WTL3132, and later the Weitek WTL3164 [36]). The main task of the custom interface chip was to transpose 32×32 bit matrices. Each of the 32 Connection Machine processors in a group would supply a 32-bit operand, one bit at a time, to the floating-point interface chip; thus, in 32 Connection Machine ALU cycles, 32 complete operands could be transferred to the interface chip, with bit j of each of 32 operands being transferred during cycle j . The interface chip would then present complete 32-bit operands, one at a time, to the floating-point coprocessor. The floating-point coprocessor had multiple registers, so operands might reside within the coprocessor while multiple floating-point operations were performed. Eventually results would be transferred from the coprocessor, one at a time, back to the interface chip, which would then transfer the results simultaneously to the 32 Connection Machine processors, one bit at a time to each. With careful microprogramming of the Connection Machine sequencer, these steps were pipelined so that operand transfer was overlapped with the floating-point operations. Later floating-point coprocessor chips also supported 64-bit floating-point operations; the process was much the same, with each operand transferred from chip to chip as two 32-bit portions. A full CM-2 system

with floating-point accelerators would contain 2,048 floating-point coprocessor chips.

The CM-1 had a separate hardware grid for its NEWS network; in the CM-2, circuitry was added to the processor chips to use the hypercube wires for NEWS connections. It was already well known that a two-dimensional grid (or, more generally, a grid of any dimension) could be embedded in a hypercube, and that all processors could be part of such a grid if the length of each axis of the grid were a power of two. This implementation strategy reduced the hardware cost of the CM-2 while increasing its flexibility to efficiently support three-dimensional grids (useful for physical simulations [42]) and four-dimensional grids (useful for QCD calculations [2]). The CM-2 NEWS interface also included features to assist the computation of parallel prefix operations along any axis of a multidimensional grid.

In the CM-2 router, as in the CM-1 router, if two or more messages were competing for use of the same hypercube wire, then one was chosen to get use of the wire. However, if any of the other competing messages had the same destination address, then the CM-2 router could *combine* them with the chosen one, rather than simply denying them use of the wire during that routing step. Messages could be combined in any of four ways: bitwise OR, choosing the largest (unsigned) value, (unsigned) integer addition, or simply choosing one arbitrarily (thereby discarding the others). (Software could also get the effect of bitwise AND or choosing the smallest value by exploiting De Morgan's laws.) The CM-1 combined messages in all the same ways, but only at the destination processor, by using its ALU as messages were received; in-router combining allowed the CM-2 to provide the same functionality with greater efficiency. The CM-2 router also had a way to fetch messages from, rather than send messages to, other processors: Request messages were routed in the usual manner, with a special kind of in-router combining operation, and then the router was run "backwards" to return data to the requesting processor – wherever in-router combining had occurred in the forward direction, the returned data was replicated in the backward direction (in effect, the forward operation constructed a set of "broadcast trees," one tree for each distinct fetch target).

The I/O interface was originally planned for the CM-1 but not implemented. In the CM-2, the I/O interface provided wide datapaths that connected directly to the Connection Machine processors, allowing high-bandwidth transfers between Connection Machine memory and peripheral devices. Each CM-2 subcube backplane provided two I/O channels, and each channel could accommodate either a framebuffer card or an I/O controller card. The 8,192 processors in each subcube were divided into two banks of 4,096 processors, that is, 256 processor chips, each having one I/O line, so I/O transfers to or from the processors consisted of 256-bit words. The graphic display framebuffer card drove a 19-inch color CRT monitor (typically $1,280 \times 1,024$ pixels), using either 8-bit or full 24-bit color, and supported data transfer from the Connection Machine processors at 40 megabytes per second. The CM I/O controller, on the other hand, multiplexed each 256-bit word into four 64-bit words for transmission on the CM I/O bus.

Two devices were initially offered by Thinking Machines Corporation for connection to the CM I/O bus: the DataVault and a VME I/O interface. The DataVault was the first commercially available RAID disk system; it supported data transfers of 25 megabytes per second, and four could be driven simultaneously by the four quadrants of a full CM-2 system. Each DataVault could hold either 42 or 84 off-the-shelf disk drives, thereby providing either 10 G bytes or 20 G bytes of data storage. Each group of 42 drives was treated as 32 for data, seven for error correction and detection, and three “hot spares.” The VME I/O interface allowed a computer with a VME bus (which might or might not also be in use as a front-end computer) to be connected to the CM I/O interface.

Each CM-2 sequencer had four times as much microcode memory as in the CM-1. The Paris instruction set, which was implemented by sequencer microcode, was greatly expanded for the CM-2. Arithmetic operations included integer and floating-point arithmetic, transcendental and trigonometric functions (including hyperbolic functions and their inverses), bitwise logical operations, interprocessor message sends with combining, multidimensional nearest-neighbor NEWS communication, global reduction operations (such as computing the logical OR of one bit from

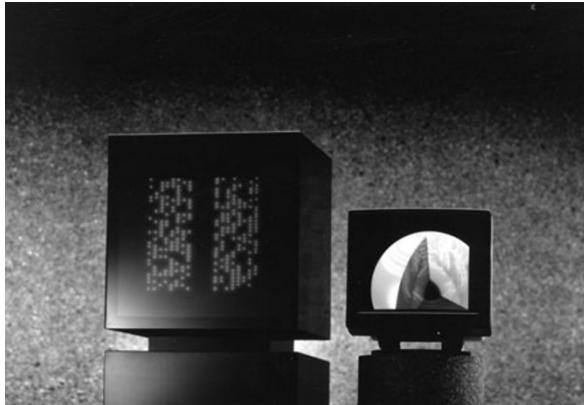
every processor, or the floating-point sum of one value from every processor), and reduction and parallel prefix operations along any axis of a multidimensional array. Every virtual processor had four flags: a carry bit, an overflow bit, a test bit (to receive the result of comparison operations), and a context bit (which if 0 suppressed storing of results for most Paris operations). Paris supported multiple virtual processor sets simultaneously within a single program; each virtual processor set could have a different NEWS configuration, and interprocessor message sends could be used to transfer data between virtual processors in different virtual processor sets. Within each virtual processor set, Paris supported dynamic memory allocation of *fields*, where a field was simply number number of consecutive bits at the same address within each virtual processor in a virtual processor set. Paris supported both stack (“push” and “pop”) and heap (analogous to “malloc” and “free”) allocation and deallocation of fields.

With the CM-2, Thinking Machines offered a choice of front-end bus interfaces, so that either a Symbolics 3600 Lisp Machine or a Digital Equipment Corporation VAX computer could be used as a front-end processor (indeed, a single CM-2 system might have both kinds of front end attached to its nexus crossbar). A VAX front end was required to execute programs written in C* or Connection Machine Fortran; either kind of front end could execute programs written in *Lisp or CM-Lisp.

The cabinetry of the CM-2 was essentially identical to that of the CM-1 (see Fig. 1). At one point, Danny Hillis investigated the possibility of making the lights blue rather than red, but discovered that while blue LEDs had just become available, their relatively high cost and short lifetime at that time made them impractical for use in the CM-2.

The DataVault cabinet was another striking example of industrial design: Rather than being rectilinear, the cabinet had a gentle curve that made it look rather like an information desk or a bartender’s station (Fig. 1). The intent was that a Connection Machine cabinet might be encircled by up to half a dozen DataVault cabinets separated by access paths.

By 1989, a version of the CM-2 called the CM-2a was offered in a smaller cabinet, approximately one eighth the size of the original CM-1/CM-2 cabinet: 26 by 26



Connection Machine. Fig. 2 Connection Machine model CM-2a with graphic display

by 43 in., consisting of a single CM-2 subcube standing on a pedestal that contained power supplies and cooling fans (see Fig. 2). The CM-2a could be populated with either 4,096 or 8,192 Connection Machine processors (and optionally with the same floating-point accelerators used in the CM-2). Three choices of front-end processor for the CM-2 and CM-2a were now available: Symbolics 3600, DEC VAX, and Sun Microsystems Sun-4/200 series. The CM-Lisp language proved to be difficult to implement in its full generality and by 1989 was no longer emphasized in Thinking Machines literature; *Lisp, C*, and Connection Machine Fortran all enjoyed continued support.

A revised version of the CM-2, with a faster clock rate and larger memory, might have become the CM-3, but was eventually shipped under the name CM-200, to emphasize continuity with the CM-2 product line. There was no CM-4 product, though that number was sometimes used to refer to a follow-on SIMD machine design that was discarded early on in favor of what became the CM-5.

The fastest CM-2 systems ever listed in a TOP500 list were two 65,536-processor machines, one at Brandeis University and one at Florida State University [58, June 1993, ranked #37 and #38]; the theoretical aggregate peak performance of each was 14 gigaflops, and the reported peak LINPACK performance was 5.2 gigaflops. The fastest CM-200 systems ever listed in a TOP500 list were two 65,536-processor machines, both at Los Alamos National Laboratory [58, June 1993, ranked #23 and #24]; the theoretical aggregate peak performance of each was 20 gigaflops,

and the reported peak LINPACK performance was 9.8 gigaflops.

The 1989 Gordon Bell prize for absolute computer performance was won by a seismic modeling application running on the CM-200 that achieved sustained performance of 6 GFLOPS [14]. The same application was further improved by use of a “stencil compiler” [8] that compiled a subroutine written in Connection Machine Fortran into application-specific microcode that could be downloaded into the sequencer; it earned a 1990 Gordon Bell Prize honorable mention for improving the performance of this application to over 14 GFLOPS [15, 36].

A variety of other scientific applications were implemented on the CM-2 [44].

Connection Machine Model CM-5

The CM-5 was designed to be upward-compatible with CM-2 software, but its hardware architecture represented a radical break in nearly every aspect [25, 56]. Rather than using proprietary 1-bit SIMD processors controlled by a proprietary sequencer, it used off-the-shelf RISC processors (SuperSPARC I chips from Sun Microsystems), thus supporting MIMD programming as well as the data-parallel style of programming. The original plan was for the CM-5 to be a purely MIMD machine, but the performance of SuperSPARC I chips turned out to be lower than predicted by the historical SPARC technology curve; in order to provide competitive numerical performance, Thinking Machines made a late design change that added proprietary floating-point vector coprocessor chips. As a result, although MIMD programming could be exploited for some purposes, the best performance on the CM-5 was attained only by using the same data-parallel programming model that had been used on the CM-1 and CM-2.

Rather than using a hypercube network to connect these processors, and separate I/O busses to connect processors to I/O devices, the CM-5 used a fat-tree network [29] to which processors and I/O devices were attached on an equal footing [31, 32, 56]. One of the (quite plausible) goals of the CM-5 architecture was to allow eventual scaling to teraflop performance [30, 32, 37], thus allowing systems to be offered in a software-compatible range of sizes spanning more than three orders of magnitude [27]. The choice of fat-tree over hypercube made this vision possible because the fat-tree, unlike the hypercube, could be made larger

without increasing the number of wires connected to each processor. The fat-tree also had the advantage that the number of processors need not be an exact power of two.

The fat-tree network was actually used to connect three type of nodes: ordinary processing nodes (perhaps having vector coprocessors), usually present in large numbers; a handful of control processor nodes, each of which ran an enhanced version of UNIX and could fulfill the role of a CM-2 front-end processor; and a modest number of I/O nodes, which were almost identical to processor nodes but had conventional I/O interfaces instead of vector coprocessor chips.

User-mode code on the SPARC could access the fat-tree network interface directly, but had to use virtual addresses to identify destinations; the CM-5 fat-tree network then provided virtual address translation. In this way the processors of a CM-5 system could be divided into groups called *partitions*, with each partition assigned to a different user task and each task protected against interference from other tasks. Privileged (supervisor) code on the SPARC could send messages using either virtual or physical (absolute) addresses to identify destinations. Typically the operating system would dedicate one control processor to manage each partition.

A high-bandwidth I/O device was typically connected to the CM-5 by striping a wide bus across a set of I/O node processors that together formed a partition; typically the operating system would dedicate one control processor to each I/O partition to manage its I/O requests. Yet another advantage of the fat-tree structure was that if partitions were allocated appropriately, then network traffic within a partition never interfered (competed for network resources) with traffic in any other partition, nor with traffic between two other partitions (thus I/O activity by one user task would not interfere with computation by another user task).

The 65,536-way logical-OR signaling network in the CM-1 and CM-2 was replaced in the CM-5 with a more comprehensive *control network*. This was a tree structure that paralleled the fat-tree *data network* but did not become fatter near the root of the tree. The CM-5 control network provided these facilities:

- Broadcasting: Any processor could inject a message into the control network, and a copy of that message

would be delivered to all processors in the partition. Each broadcast message could be from 1 to 15 32-bit words in length. There were actually three distinct broadcast facilities: user-mode, supervisor-mode, and interrupt. Supervisor-mode and interrupt broadcasts were privileged operations. An interrupt broadcast could cause every processor in the partition to receive either an interrupt or a hard-reset signal.

- Combining: Each processor could (asynchronously) inject a message into the control network; after every processor in the partition had done so, a result would be delivered to each processor. Four different combining modes were supported: global reduction, parallel prefix, parallel suffix, and router-done (which was simply a specialized logical-OR reduction intended to assist processors in cooperatively determining whether a bulk data-network transmission phase had been completed). The combining operation for the first three modes could be bitwise OR, bitwise XOR, signed maximum, signed integer addition, or unsigned integer addition. (Software could also get the effect of bitwise AND, signed minimum, unsigned maximum, or unsigned minimum by exploiting De Morgan's laws or related techniques.) Each message could be 32 to 128 bits long.
- Global bit: As in the CM-1 and CM-2, each processor could provide one bit, but the logical OR of these bits was delivered to all CM-5 processors in the partition rather than to a sequencer. In the CM-5, there were actually three global-bit interfaces: user-mode synchronous, user-mode asynchronous, and supervisor-mode asynchronous.

The CM-5 also included a third network, the diagnostic network, that was invisible to application programmers but allowed the system to self-monitor for hardware failures and to isolate failed components. This was based on then-emerging standards for on-chip testing and connection-control circuitry using a serial interface that required only a few extra pins per chip, but the CM-5 implementation was notable for using a tree-structured network to allow testing of thousands of chips in parallel. The CM-5 diagnostic network could also diagnose itself by using chips higher in the tree to test chips lower in the tree [38, 56].

A CM-5 processor node *without* vector coprocessors consisted of a 64-bit memory bus connecting three



chips: the RISC processor chip, a network interface chip that connected to the control network and data network [57], and a memory interface that could manage from one to four memory chips of 8 MB each. A CM-5 processor node *with* vector coprocessors consisted of a 64-bit memory bus connecting six chips: the RISC processor chip, a network interface chip, and four vector coprocessor chips, each of which also served as a memory interface that could manage from one to four memory chips of 8 MB each. Thus a processor node without vector coprocessors could have up to 32 MB of memory, but a processor node with vector coprocessors could have up to 128 MB of memory.

In 1991, each vector coprocessor had a peak memory bandwidth of 128 MB/s, 32 megaflops of peak 64-bit floating-point performance, and 32 mega-ops peak 64-bit integer performance [38]. (By 1993, these figures had been increased to 160 MB/s, 40 megaflops, and 40 mega-ops [56].) The RISC microprocessor could issue instructions to individual vector units or to all four at once. Each vector unit had 64 64-bit registers (which were also addressable as 128 32-bit registers), a 16-bit vector mask, and a 4-bit vector length register. Each vector instruction could process up to 16 sets of operands. Each register vector operand was specified by a 7-bit starting register number and a 7-bit stride; the first element for that vector operand was taken from the starting register; that register number was then repeatedly incremented by the given stride to produce register numbers containing succeeding elements of the vector operand. A large stride had the same effect as a small negative stride, so vector operands could be processed in reverse order. Each vector coprocessor provided addition, multiplication, memory load/store, indirect register addressing, indirect memory addressing, and population count (counting the 1-bits in a word). Each vector-unit instruction could specify at least one arithmetic operation and an independent memory operation, allowing pipelining of operands from memory and results to memory while computation was taking place internal to the coprocessor. Single-cycle multiply-add and multiply-subtract operations were supported, as well as the fairly unusual combination of an integer multiply (high or low part) followed by a bitwise Boolean operation – by choosing the high or low part of the multiply and using an appropriate power of 2 as the multiplier, the programmer could get the effect of

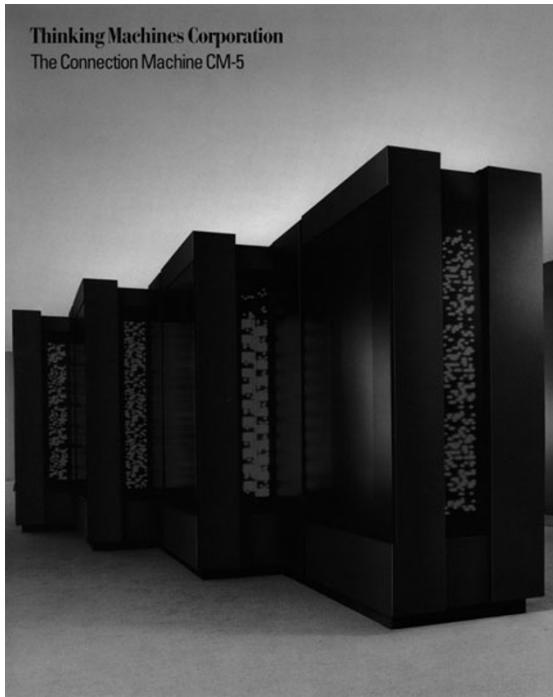
various shift-and-mask or shift-and-XOR operations in a single cycle.

The CM-5 had an option to integrate a massively parallel disk array (called SDA, or Scalable Disk Array) into the CM-5 processor cabinets: One form of CM-5 I/O node consisted of the usual RISC processor and CM-5 network interface chip, a data buffer, four SCSI-2 controllers, and eight 3.5-in. hard disk drives. Another form of CM-5 I/O node was similar but had no disk drives and just two SCSI-2 controllers; these nodes were connected to tape drives mounted in a separate cabinet (called ITS, the Integrated Tape System). A CM5-HIPPI interface allowed the CM-5 to be connected to an industry-standard HIPPI (High Performance Parallel Interface) bus; a CM-HIPPI interface was also offered for CM-2 and CM-200 systems, allowing different generations of Connection Machine systems to communicate at high speed with each other or to share I/O devices. Yet another form of CM-5 I/O node was a CM I/O bus adapter, which allowed use of CM-2 peripherals, such as the DataVault, on CM-5 systems.

The CM-5, like the CM-2 and CM-200, supported the *Lisp, C*, and Connection Machine Fortran languages, as well as CMSSL (the Connection Machine Scientific Software Library [26]) and the Prism programming environment [55, 56]. Prism was notable for using graphics-intensive visualization techniques to display debugging data for thousands of processors [45, 46] and for being one of the software products that survived the dissolution of Thinking Machines Corporation [47].

The operating system for the CM-5, which ran in the control processors, was an enhanced version of UNIX called CMOST, for “Connection Machine Operating SysTem” [28, 56]. (The more obvious acronym “CMOS” was passed over because it was, and still is, in wide use to denote the integrated-circuit technology “Complementary Metal-Oxide-Semiconductor.”) This operating system eventually included a “scalable file system” that extended supported file sizes from 32-bit integers to 64-bit integers [34].

The cabinetry of the CM-5 was quite different from that of the CM-1 and CM-2 but no less striking; multiple 7-foot-tall, oblong cabinets, each of which could hold up to 256 processors, were connected in groups of five in a staggered “lightning bolt” arrangement that allowed one end of each cabinet to display a large rectangular panel



Connection Machine. Fig. 3 Four cabinets of a Connection Machine model CM-5

of the red LEDs that had become the signature look of the Connection Machine series (see Fig. 3). These LEDs were not, however, mounted on the processor boards, and each cabinet had a panel of lights, no matter whether the cabinet contained processors, storage modules, fat-tree nodes, I/O interfaces, or a mixture. (The fact that the lights were now physically separated from the processors made it less costly to exploit their eerie look in the 1993 movie *Jurassic Park*; CM-5 cabinets – with the blinking red lights but without processors [1] – were featured in the “control room” scenes, the phrase “Connection Machine” was used in a bit of dialogue, and the Thinking Machines Corporation logo appeared prominently in the credits.)

The central cabinet in each group of five could contain fat-tree nodes sufficient to connect the other four cabinets to each other and to other lightning bolts. CM-5 systems with more than four processor cabinets were arranged on the machine room floor as parallel lightning bolts with their central cabinets connected by overhead cable bridges, each several feet wide, that supported the numerous physical fat-tree connections between the five-cabinet groups.

The fastest CM-5 system ever listed in a TOP500 list was a 1,056-processor machine at Los Alamos National Laboratory [58, June 1993, ranked #1]; each processor node was rated at 128 megaflops, for a theoretical aggregate peak performance of 135 gigaflops, and the reported peak LINPACK performance was 59.7 gigaflops.

Examples of scientific applications implemented on the CM-5 include computational fluid mechanics [41] and a solution of the Boltzmann equation [33] that was measured as running at 60.7 gigaflops.

Related Entries

- C*
- Cache-Only Memory Architecture (COMA)
- Connection Machine Fortran
- Connection Machine Lisp
- MPP
- HPF (High Performance Fortran)
- Illiac IV
- *Lisp
- MasPar

Bibliography

1. Anonymous (1993) Lights! Action! Cue the computer! Parallelogram: The international journal of high performance computing, 55 (September/October 1993). Fitzroy, London, ISSN 0953-7252, pp 10–11
2. Bailie CF, Brickner RG, Gupta R, Johnsson L (1989) QCD with dynamical fermions on the Connection Machine. In: Supercomputing '89: Proceedings 1989 ACM/IEEE conference on supercomputing. ACM, New York, pp 2–9. ISBN 0-89791-341-8. <http://doi.acm.org/10.1145/76263.76264>
3. Barnes GH, Brown RM, Kato M, Kuck DJ, Slotnick DL, Stokes RA (1968) The Illiac IV computer. IEEE Trans Comput C-17, 8 August 1968, pp 746–757. ISSN 0018-9340. <http://dx.doi.org/10.1109/TC.1968.229158>
4. Batcher KE (1974) STARAN parallel processor system hardware. In: AFIPS '74: proceedings national computer conference and exposition. ACM, New York, pp 405–410. <http://doi.acm.org/10.1145/1500175.1500260>
5. Batcher KE (1977) The multidimensional access memory in STARAN. IEEE Trans Comput 26(2):174–177. IEEE Comput Soc, Washington, DC. ISSN 0018-9340. <http://dx.doi.org/10.1109/TC.1977.5009297>
6. Batcher KE (1982) MPP: A supersystem for satellite image processing. In: AFIPS '82: proceedings, National computer conference, 7–10 June 1982. ACM, New York, pp 185–191. ISBN 0-88283-035-X. <http://doi.acm.org/10.1145/1500774.1500795>
7. Bouknight WJ, Denenberg SA, McIntyre DE, Randall JM, Sameh AH, Slotnick DL (1972) The ILLIAC IV system. Proceedings IEEE

- 60, 4 Apr 1972, pp 369–388. ISSN 0018-9219. <http://dx.doi.org/10.1109/PROC.1972.8647>
8. Bromley M, Heller S, McNerney T, Steele GL Jr. (1991) Fortran at ten gigaflops: The Connection Machine convolution compiler. In: PLDI '91: proceedings ACM SIGPLAN 1991 conference on programming language design and implementation. ACM, New York, pp 145–156. ISBN 0-89791-428-7. <http://doi.acm.org/10.1145/113445.113458>
 9. Cray Research, Inc. (1977) CRAY-1 computer system hardware reference manual 2240004. Bloomington, MN, November 1977. <http://www.bitsavers.org/pdf/cray/2240004C> CRAY-1 Hardware Reference Nov77.pdf
 10. Cray Research, Inc. (1982) CRAY X-MP series mainframe reference manual HR-0032. Mendota Heights, MN, November 1982. <http://www.bitsavers.org/pdf/cray/> R-0032 X-MP MainframeRef Nov82.pdf
 11. Cray Research, Inc (1984) CRAY X-MP series model 48 mainframe reference manual HR-0097. Mendota Heights, MN, August 1984. <http://www.bitsavers.org/pdf/cray/HR-0097> CRAY X-MP Series Model 48 Mainframe Ref Man Aug84.pdf
 12. Delany HC (1988) Ray tracing on a Connection Machine. In: ICS '88: proceedings 2nd international conference on supercomputing. ACM, New York, pp 659–667. ISBN 0-89791-272-1. <http://doi.acm.org/10.1145/55364.55429>
 13. Digital Equipment Corporation (1969) PDP-8/S maintenance manual, F-87S, fourth printing. Maynard, MA, August 1969. <http://www.bitsavers.org/pdf/dec/pdp8/pdp8s/PDP8SMaintMan.pdf>
 14. Dongarra J, Karp AH, Kennedy K, Kuck D (1990) Special report: 1989 Gordon Bell prize. Software 7(3):100–104, 110. IEEE, Los Alamitos, California. <http://dx.doi.org/10.1109/MS.1990.10021>
 15. Dongarra JJ, Karp A, Miura K, Simon HD (1991) Gordon Bell prize lectures. In: Supercomputing '91: proceedings 1991 ACM/IEEE conference on Supercomputing. ACM, New York, pp 328–337. ISBN 0-89791-459-7. <http://doi.acm.org/10.1145/125826.126011>
 16. Fahrlman SE (1980) Design sketch for a million-element NELL machine. In: AAAI-80: proceedings first national conference on artificial intelligence. Morgan-Kaufmann, Los Altos, CA, pp 249–252. <http://www.aaai.org/Papers/AAAI/1980/AAAIpenalty-\@M80-070.pdf>
 17. Gabriel RP (1986) Massively parallel computers: The Connection Machine and NON-VON. Science 231(4741):975–978. American Association for the Advancement of Science, New York. ISSN 0036-8075. <http://dx.doi.org/10.1126/science.231.4741.975>
 18. Gottlieb A, Grishman R, Kruskal CP, McAuliffe KP, Rudolph L, Snir M (1983) The NYU Ultracomputer—Designing an MIMD shared memory parallel computer. IEEE Trans Comput 32(2):175–188. IEEE Computer Society, Washington, DC. ISSN 0018-9340. <http://dx.doi.org/10.1109/TC.1983.1676201>
 19. Gregory J, McReynolds R (1963) The SOLOMON computer. IEEE Trans Electronic Comput EC-12(6):774–781. ISSN 0367-7508. <http://dx.doi.org/10.1109/PGEC.1963.263560>
 20. Hillis WD (1990) Multi-dimensional message transfer router. United States Patent 5,151,996. Filed 20 March 1990. Granted 29 September 1992
 21. Hillis WD (1981) The Connection Machine (computer architecture for the new wave). AI Memo 646, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, MA, September 1981
 22. Hillis WD (1985) The Connection Machine. MIT Press, Cambridge. ISBN 0-262-08157-1
 23. Hillis WD (1987) The Connection Machine. Scientific American 256(6):108–115. Scientific American, New York. ISSN 0036-8733. <http://dx.doi.org/10.1038/scientificamerican0687-108>
 24. Hillis WD, Steele GL Jr. (1986) Data parallel algorithms. Commun ACM 29(12):1170–1183. ACM, New York. ISSN 0001-0782. <http://doi.acm.org/10.1145/7902.7903>
 25. Hillis WD, Tucker LW (1993) The CM-5 Connection Machine: A scalable supercomputer. Commun ACM 36(11):31–40. ACM, New York. ISSN 0001-0782. <http://doi.acm.org/10.1145/163359.163361>
 26. Johnsson SL (1993) CMSSL: A scalable scientific software library. In: Proceedings 1993 conference on scalable parallel libraries. IEEE Computer Society Press, Silver Spring, pp 57–66. <http://dx.doi.org/10.1109/SPLC.1993.365582>
 27. Johnsson SL (1993) The Connection Machine systems CM-5. In: SPAA '93: proceedings fifth annual ACM symposium on parallel algorithms and architectures. ACM, New York, pp 365–366. ISBN 0-89791-599-2. <http://doi.acm.org/10.1145/165231.157377>
 28. Kahle BU, Nesheim WA, Isman M (1988) Unix and the Connection Machine operating system. In: Proceedings workshop on UNIX and supercomputers. USENIX Association, Berkeley, pp 93–107
 29. Leiserson CE (1985) Fat-trees: Universal networks for hardware-efficient supercomputing. IEEE Trans Comput 34(10):892–901. IEEE Computer Society, Washington, DC. ISSN 0018-9340
 30. Leiserson CE (1992) The networks of the Connection Machine CM-5. In: Meyer F, Monien B, Rosenberg AL (eds) Parallel architectures and their efficient use, first Heinz Nixdorf symposium, Paderborn, Germany, 11–13 November, proceedings. Lecture notes in computer science, vol 678. Springer, Berlin, pp 66–67. ISBN 3-540-56731-3. <http://dx.doi.org/10.1007/3-540-56731-3>
 31. Leiserson CE, Abuhamdeh ZS, Douglas DC, Feynman CR, Ganmukhi MN, Hill JV, Hillis WD, Kuszmaul BC, St. Pierre MA, Wells DS, Wong-Chan MC, Yang S-W, Zak R (1996) The network architecture of the Connection Machine CM-5. J Parallel Distr Comput 33(2):145–158. Elsevier. ISSN 0743-7315. <http://dx.doi.org/10.1006/jpdc.1996.0033>
 32. Leiserson CE, Abuhamdeh ZS, Douglas DC, Feynman CR, Ganmukhi MN, Hill JV, Hillis WD, Kuszmaul BC, St. Pierre MA, Wells DS, Wong MC, Yang S-W, Zak R (1992) The network architecture of the Connection Machine CM-5 (extended abstract). In: SPAA '92: proceedings fourth annual ACM symposium on parallel algorithms and architectures. ACM, New York, pp 272–285. ISBN 0-89791-483-X. <http://doi.acm.org/10.1145/140901.141883>
 33. Long LN, Myczkowski J (1993) Solving the Boltzmann equation at 61 gigaflops on a 1024-node CM-5. In: Supercomputing '93: proceedings 1993 ACM/IEEE conference on supercomputing. ACM, New York, pp 528–534. ISBN 0-8186-4340-4. <http://doi.acm.org/10.1145/169627.169795>

34. LoVerso SJ, Isman M, Nanopoulos A, Nesheim W, Milne ED, Wheeler R (1993) sfs: A parallel file system for the CM-5. In: USENIX-STC'93: proc. USENIX summer 1993 technical conference. USENIX Association, Berkeley, CA, pp 291–305
35. McCormick BH (1963) The Illinois pattern recognition computer—ILLIAC III. *IEEE Trans Electron Comput EC-12(6)*:791–813. ISSN 0367-7508. <http://dx.doi.org/10.1109/PGEC.1963.263562>
36. Myczkowski J, Steele G (1991) Seismic modeling at 14 gigaflops on the Connection Machine. In: Supercomputing '91: proceedings 1991 ACM/IEEE conference on supercomputing. ACM, New York, pp 316–326. ISBN 0-89791-459-7. <http://doi.acm.org/10.1145/125826.126004>
37. Negele JW (1993) QCD teraflops computer. Nuclear physics B—Proceedings supplements 30 (March 1993), pp 295–298. ISSN 0920-5632. [http://dx.doi.org/10.1016/0920-5632\(93\)90212-O](http://dx.doi.org/10.1016/0920-5632(93)90212-O)
38. Palmer J, Steele GL Jr. (1992) Connection Machine model CM-5 system overview. In: Proceedings fourth symposium on the frontiers of massively parallel computation. IEEE Computer Society Press, Los Alamitos, pp 474–483. ISBN 0-8186-2272-7. <http://dx.doi.org/10.1109/FMPC.1992.234877>
39. Reddaway SF (1973) DAP—A distributed array processor. In: ISCA '73: Proceedings 1st annual symposium on computer architecture. ACM, New York, pp 61–65. <http://doi.acm.org/10.1145/800123.803971>
40. Seitz CL (1985) The Cosmic Cube. *Commun ACM* 28(1):22–33. ACM, New York. ISSN 0001-0782. <http://doi.acm.org/10.1145/2465.2467>
41. Sethian JA (1993) Computational fluid mechanics and massively parallel processors. In: Supercomputing '93: proceedings 1993 ACM/IEEE conference on supercomputing. ACM, New York, pp 74–82. ISBN 0-8186-4340-4. <http://doi.acm.org/10.1145/169627.169660>
42. Sethian JA, Brunet J-P, Greenberg A, Mesirow JP (1991) Computing turbulent flow in complex geometries on a massively parallel processor. In: Supercomputing '91: proceedings 1991 ACM/IEEE conference on Supercomputing. ACM, New York, pp 230–241. ISBN 0-89791-459-7. <http://doi.acm.org/10.1145/125826.125954>
43. Shaw DE, Stolfo SJ, Ibrahim H, Hillyer B, Wiederhold G, Andrews JA (1981) The NON-VON database machine: A brief overview. *Database engineering bulletin: a quarterly bulletin of the IEEE Comput Soc Tech Committ Database Eng* 4(1):41–52. IEEE Computer Society, Washington, DC. <http://sites.computer.org/debull/8IDECCD.pdf>
44. Simon HD (ed) (1988) Proceedings conference scientific applications of the Connection Machine. World Scientific, Singapore, September 1988. ISBN 9971-50-969-5
45. Sistare S, Allen D, Bowker R, Jourdenais K, Simons J, Title R (1992) Data visualization and performance analysis in the Prism programming environment. In: Topham NP, Ibbett RN, Bemmerl T (eds) Proceedings of the IFIP WG 10.3 workshop on programming environments for parallel computing, vol. A-11 of IFIP Transactions. North-Holland Publishing, Amsterdam, April 1992, pp 37–52. ISBN 0-444-89764-X
46. Sistare S, Allen D, Bowker R, Jourdenais K, Simons J, Title R (1994) A scalable debugger for massively parallel message-passing programs. *IEEE Parallel Distri Technol* 2(2):50–56. IEEE Computer Society Press, Los Alamitos, CA. ISSN 1063-6552. <http://dx.doi.org/10.1109/88.311572>
47. Sistare S, Dorenkamp E, Nevin N, Loh E (1999) MPI support in the Prism programming environment. In: Supercomputing '99: proceedings 1999 ACM/IEEE conference on supercomputing (CDROM). ACM, New York. ISBN 1-58113-091-0. <http://doi.acm.org/10.1145/331532.331554>
48. Slotnick DL, Borck WL, McReynolds RC (1962) The SOLOMON computer. In: AFIPS '62 (Fall): proceedings fall joint computer conference. ACM, New York, December 1962, pp. 97–107. <http://doi.acm.org/10.1145/1461518.1461528>
49. Squire JS, Palais SM (1963) Programming and design considerations of a highly parallel computer. In: AFIPS '63 (Spring): proceedings spring joint computer conference. ACM, New York, May 1963, pp 395–400. <http://doi.acm.org/10.1145/1461551.1461597>
50. Stanfill C, Kahle B (1986) Parallel free-text search on the Connection Machine system. *Commun ACM* 29(12):1229–1239. ACM, New York. ISSN 0001-0782. <http://doi.acm.org/10.1145/7902.1@M7907>
51. Stanfill C, Waltz D (1986) Toward memory-based reasoning. *Commun ACM* 29(12):1213–1228. ACM, New York. ISSN 0001-0782. <http://doi.acm.org/10.1145/7902.7906>
52. Thiel T (1994) The design of the Connection Machine. *Design issues* 10(1):5–18. MIT Press, Cambridge. ISSN 0747-9360
53. Thinking Machines Corporation (1987) Connection Machine model CM-2 technical summary. Technical report HA87-4. Cambridge, April 1987
54. Thinking Machines Corporation (1989) Connection Machine technical summary, version 5.1. Cambridge, May 1989
55. Thinking Machines Corporation (1991). Prism user's guide, version 1.0. Cambridge, December 1991
56. Thinking Machines Corporation (1993) Connection Machine CM-5 technical summary, third edition. Cambridge, November 1993
57. Thinking Machines Corporation (1993) Programming the NI, version 7.1. Cambridge, February 1993
58. Top 500 supercomputer sites (1993) Semiannual lists since 1993 of the top 500 supercomputer sites in the world as measured by a LINPACK benchmark. <http://www.top500.org>. Accessed 30 March 2011
59. Tucker LW, Robertson GG (1988) Architecture and applications of the Connection Machine. *Computer* 21(8):26–38. IEEE. ISSN 0018-9162. <http://dx.doi.org/10.1109/2.74>
60. Unger SH (1958) A computer oriented toward spatial problems. In: Proceedings IRE 46(10):1744–1750. Institute of Radio Engineers/IEEE. ISSN 0096-8390. <http://dx.doi.org/10.1109/JRPROC.1958.286755>
61. Webber DM, Sangiovanni-Vincentelli A (1987) Circuit simulation on the Connection Machine. In: DAC '87: proceedings 24th ACM/IEEE design automation conference. ACM, New York, pp 108–113. ISBN 0-8186-0781-5. <http://doi.acm.org/10.1145/37888.37904>

Connection Machine Fortran

GUY L. STEELE JR.
Oracle Labs, Burlington, MA, USA

Synonyms

[CM fortran](#)

Definition

Connection Machine Fortran is a data-parallel version of Fortran developed around 1987 for Connection Machine supercomputers manufactured by Thinking Machines Corporation. It consists essentially of Fortran 77 augmented by array-processing features that had been proposed for Fortran 8x (and were eventually adopted as part of the Fortran 90 and Fortran 95 standards), additional data-parallel intrinsic functions (such as for parallel prefix operations), and data distribution directives. It was one of the parallel Fortran projects that contributed several noteworthy features to the design of High Performance Fortran.

Discussion

Connection Machine Fortran (also called CM Fortran) was developed and implemented for the CM-2 and CM-5 models of Connection Machine supercomputer. The language was specified by Thinking Machines Corporation but was initially implemented under contract by Compass, Inc. [1]. (That well-known compiler company, also known as Massachusetts Computer Associates, had previously implemented a Fortran compiler for the Goodyear MPP [2].)

Of the four programming languages (*Lisp, C*, CM Fortran, and CM-Lisp) provided by Thinking Machines Corporation for Connection Machine Systems, CM Fortran was perhaps the most conventional, adhering as closely as possible to existing Fortran standards or projected future standards, and yet also the most influential, because of its contributions to High Performance Fortran.

The version as of April 1987 was described [3] as including all of Fortran 77 as defined by ANSI standard X3.9-1978, as well as two sets of extensions: those defined by MIL-STD-1753 [4] (principally intrinsic functions for bit manipulation, the DO WHILE statement, the END DO statement, and the

IMPLICIT NONE statement), and a subset of features proposed in the draft ANSI Fortran 8x standard (draft S8, version 103), including some features described in that draft as “removed extensions.” The Fortran 8x features adopted by CM Fortran included array-valued expressions and elemental functions; other intrinsic functions such as reduction operations (such as SUM, PRODUCT, MAXVAL, MAXLOC, ANY, and ALL), dot product (DOTPROD), and matrix multiplication (MATMUL); array constructor expressions; array sections and vector-valued subscripts; masked array assignment (the WHERE statement and WHERE construct); and the FORALL statement and FORALL construct, which are somewhat like a DO loop, but possibly having more than one index variable, that executes its body (a single statement or a block of statements) in a data-parallel fashion for all possible combinations of values of its index variables simultaneously – and if a logical mask expression is also included, then only combinations of index values for which the mask expression is true are used. For example, assume that A and B are arrays of shape 100×100 ; then the FORALL construct

```
FORALL (I=1:100, J=1:100, I .NE. J)
  A(I,J) = 0.0
  B(I,J) = C(I,J)
  C(I,J) = C(J,I)
END FORALL
```

first sets all the off-diagonal elements of A to 0.0, and only then copies the off-diagonal elements of C into corresponding positions of B, and only after that transposes (the off-diagonal elements of) the array C.

It is noteworthy that the FORALL statement and construct were among the “removed extensions” in the Fortran 8x draft and were not incorporated into the Fortran 90 standard; that the FORALL statement and construct were adopted as part of High Performance Fortran [5, pp 170–184]; and that as a result of experience with High Performance Fortran, the FORALL statement and construct were included in the Fortran 95 standard. However, though the FORALL statement and construct were described as part of CM Fortran as early as April 1987 [3], they were not yet implemented in CM Fortran as of February 1990 [6, page 83] (see also [7, p 7]). They do appear to have been implemented by mid-1992 (an internal design document for implementing FORALL [8] is dated November 1991, and published

release notes dated October 1992 refer to an “enhancement” to FORALL that generates parallel code for more cases than in “previous releases” [9, p 13]). Over the next 2 years, additional enhancements and optimizations for FORALL were introduced for the both CM-5 and the CM-200 [10–13].

By September 1989, a set of compiler directives had been added to CM Fortran [14, pp 365–386]. Following the practice of Fortran compilers for other supercomputers such as the Cray series, CM Fortran directives took the form of structured comments – in this case, exploiting the fact that both Fortran comments and the abbreviation “CMF” for Connection Machine Fortran begin with the letter “C”. The two directives of greatest historical and technical interest were the LAYOUT and ALIGN directives. The LAYOUT directive specified how a Fortran array should be laid out across Connection Machine processors; for example,

```
DIMENSION A(64, 64, 64, 100)
CMF$ LAYOUT A(:SEND, :NEWS, :NEWS, :SERIAL)
```

specified that the four-dimensional array A should be laid out in such a way that elements whose indices differed only in the last axis should reside in the same virtual processor, and that the other three axes should be distributed across virtual processors in such a way that the processor numbering along the first axis corresponded to the Connection Machine router’s “send address” ordering, and the processor numbering along the second and third axes corresponded to the Connection Machine’s NEWS grid ordering (which amounted to a Gray encoding of some portion of the router address). The ALIGN directive specified that the layout of one array should be determined by the layout of another array, so as to maintain minimum communications cost between corresponding elements. For example,

```
DIMENSION A(64,64,64,100),B(64,64,64),C(64)
CMF$ ALIGN B(I,J,K) WITH A(I,J,K,1)
CMF$ ALIGN C(M) WITH B(M,M,M)
```

causes array B to be aligned with (and therefore allocated in the same virtual processors as) a slice of array A, and causes array C to be aligned with the main space diagonal of array B. Such directives allowed for explicit programmer control over array allocation decisions that had previously been made automatically by an optimization phase in the CM Fortran compiler [1, 15–18].

These layout and alignment directives were eventually adopted, with some modifications and contributions from other parallel Fortran projects [19] (notably Fortran D [20–22] and Vienna Fortran [23, 24]), into High Performance Fortran [5, pp 91–119].

As implemented for the Connection Machine model CM-2, CM Fortran did not have intrinsic functions for scan (parallel prefix and parallel suffix) operations; instead, there was a utility library that allowed CM Fortran programs to invoke individual instructions from the PARIS (Connection Machine PARallel Instruction Set), including instructions that performed scan operations. With the introduction of the model CM-5, a CM Fortran Utility Library was introduced that provided a more abstract interface to certain computational and communications facilities that could be supported on both the CM-2 and CM-5. This library included scan operations with names such as CMF_SCAN_ADD and supported segmented scan operations [25, pp 74–77]. It also included a group of operations labeled “scatters with combining,” with names such as CMF_SEND_ADD [25, pp 85–86]. These library functions were the direct predecessors of intrinsic functions such as SUM_PREFIX, SUM_SUFFIX, and SUM_SCATTER in High Performance Fortran [5, pp 301–303].

By 1993, Thinking Machines Corporation in collaboration with Applied Parallel Research had produced a parallelizing translator from Fortran 77 to CM Fortran called CMAX [26].

Figure 1 shows an early (1987) example of a CM Fortran program that identifies prime numbers less than 100,000 by the method of the Sieve of Eratosthenes, taken (with one minor correction) from [3]. This program is written in what is now entirely conventional Fortran, except that the expression [1:N] would have to be written [I, I=1, N]. Because Fortran arrays normally use 1-origin indexing (rather than the 0-origin indexing provided by such languages as C, Java, and Common Lisp), the parameter N used to specify array length is defined to be 99999 rather than 100000. The first three assignment statements set every element of array PRIME to false, every element of array CANDIDATE to true, and the first element of CANDIDATE to false. While CM Fortran did have a DO WHILE statement, this example happens to use a statement label 20 and a conditional

```

SUBROUTINE FINDPRIMES(PRIME)
PARAMETER (N = 99999)
LOGICAL PRIME(N), CANDIDATE(N)
PRIME = .FALSE.
CANDIDATE = .TRUE.
CANDIDATE(1) = .FALSE.
20 NEXTPRIME = MINLOC([1:N], CANDIDATE)
PRIME(NEXTPRIME) = .TRUE.
FORALL (I = 1:N, MOD(I,NEXTPRIME) .EQ. 0) CANDIDATE(I) = .FALSE.
IF (ANY(CANDIDATE)) GO TO 20
RETURN
END

```

Connection Machine Fortran. Fig.1 Example Connection Machine Fortran program for identifying prime numbers

GO TO statement to implement the loop. The standard Fortran intrinsic function MINLOC is used to find the index of the smallest value between 1 and N for which the array CANDIDATE has a true value. This index is then used to determine which element of PRIME to set to true. The FORALL statement processes all elements of CANDIDATE in parallel, setting to false those elements for which the expression MOD(I, NEXTPRIME) .EQ. 0 is true. The standard Fortran intrinsic function ANY returns true if any element of its array argument is true; thus the loop is repeated if any candidate remains. Note that both MINLOC and ANY are reduction operations at heart.

Related Entries

- [C*](#)
- [Connection Machine](#)
- [Connection Machine Lisp](#)
- [Fortran 90 and Its Successors](#)
- [HPP \(High Performance Fortran\)](#)
- [*Lisp](#)

Bibliography

1. Albert E, Knobe K, Lukas JD, Steele GL Jr (1998) Compiling Fortran 8x array features for the Connection Machine computer system. In: PPEALS '88: Proceedings of the ACM/SIGPLAN conference on parallel programming: Experience with applications, languages and systems, ACM, New York, pp 42–56, June 1988
2. Knobe K, Loveman DB, Marcus M, Wells I (1984) A Fortran compiler for the Massively Parallel Processor. Technical Report CADD-8402-2101, Massachusetts Computer Associates (COMPASS), Wakefield, Feb 1984
3. Thinking Machines Corporation (1987) Connection Machine model CM-2 technical summary. Technical Report HA87-4, Cambridge, MA
4. United States Department of Defense (1978) MIL-STD-1753. Military standard: FORTRAN, DOD supplement to American national standard X3.9-1978. Washington, DC, Nov 1978
5. Koelbel CH, Loveman DB, Schreiber RS, Steele GL Jr, Zosel ME (1994) The High Performance Fortran handbook. MIT Press, Cambridge, MA
6. Thinking Machines Corporation (1990) Getting started in CM Fortran. version 5.2-0.6, Cambridge, MA
7. Thinking Machines Corporation (1989) CM Fortran release notes. version 5.2-0.6, Cambridge, MA
8. Mincy J (1991) Forall design. Unpublished document, 51 pages plus title page, Nov 1991
9. Thinking Machines Corporation (1992) CM Fortran release notes. version 2.0 Beta 2, Cambridge, MA
10. Thinking Machines Corporation (1993) CM Fortran release notes: detailed. version 2.1 Beta 2.0, Cambridge, MA
11. Thinking Machines Corporation (1994) CM Fortran for the CM-200 release notes. version 2.1.1, Cambridge, MA
12. Thinking Machines Corporation (1994) CM Fortran for the CM-5 release notes. version 2.1.1, Cambridge, MA
13. Thinking Machines Corporation (1994) CM Fortran 2.2 Beta release notes. Cambridge, MA
14. Thinking Machines Corporation (1989) CM Fortran reference manual. version 5.2-0.6, Cambridge, MA
15. Knobe K, Lukas JD, Steele GL Jr (1988) Massively parallel data optimization. In: Frontiers '88: Proc. 2nd symposium on the frontiers of massively parallel computation, IEEE Computer Society Press, Washington, DC, pp 551–558, October 1988
16. Knobe K, Lukas JD, Steele GL Jr (1990) Data optimization: Allocation of arrays to reduce communication on SIMD machines. J Parallel Distribut Comput 8(2):102–118
17. Knobe K, Lukas JD, Steele GL Jr (1990) Data parallel computers and the FORALL statement. In: Frontiers '90: Proceedings of the 3rd symposium on the frontiers of massively parallel

- computation, IEEE Computer Society Press, Los Alamitos, California, pp 390–396, October 1990
18. Albert E, Lukas JD, Steele GL Jr (1991) Data parallel computers and the FORALL statement. *J Parallel Distrib Comput* 13(2):185–192
 19. Steele GL Jr (1993) High Performance Fortran: Status report. Workshop on languages, compilers, and run-time environments for distributed memory multiprocessors. *SIGPLAN Notices* 28(1):1–4
 20. Fox G, Hiranandani S, Kennedy K, Koelbel C, Kremer U, Tseng CW, Wu MY (1990) Fortran D language specifacaton. Tech. Rep. CRPC-TR 90079, Center for Research on Parallel Computation, Rice University, Houston, Texas, December 1990
 21. Hiranandani S, Kennedy K, Koelbel C, Kremer U, Tseng CW (1991) An overview of the Fortran D programming system. Tech. Rep. CRPC-TR 91121, Center for Research on Parallel Computation, Rice University, Houston, Texas, March 1991
 22. Hiranandani S, Kennedy K, Tseng CW (August 1992) Compiling Fortran D for MIMD distributed-memory machines. *Commun ACM* 35(8):66–80
 23. Chapman B, Mehrotra P, Zima H (1992) Programming in Vienna Fortran. *Sci Program* 1(1):31–50
 24. Chapman B, Moritsch H, Mehrotra P, Zima H (1993) Dynamic data distributions in Vienna Fortran. In: Supercomputing '93: proceedings of the 1993 ACM/IEEE conference on supercomputing, ACM, New York, NY, pp 284–293
 25. Thinking Machines Corporation (1992) CM Fortran user's guide for the CM-5. version 1.1.3, Cambridge, MA
 26. Sabot G, Wholey S (1993) Cmax: A Fortran translator for the Connection Machine system. In: ICS '93: Proc. 7th international conference on supercomputing, ACM, New York, pp 147–156

from S-expressions to S-expressions, and special syntax for performing elementwise operations, reductions, and permutations on these data structures.

Discussion

Of the four programming languages (*Lisp, C*, CM Fortran, and CM-Lisp) provided by Thinking Machines Corporation for Connection Machine Systems, CM-Lisp was the most radical in design (requiring the use of non-ASCII characters in its notation, and introducing an associative data structure indexed by non-numeric values) and the most difficult to implement (requiring automatic garbage collection of parallel data structures). Danny Hillis used it in his book about the Connection Machine [4] to explain how he imagined programming a massively parallel supercomputer intended for nonnumerical applications in a data parallel style [5]. While it was featured along with the other three languages in the earliest *Connection Machine Technical Summary* [10], it was dropped from all later versions [11, 12], and Thinking Machines never offered a complete parallel implementation as a commercial product.

CM-Lisp consists of Common Lisp augmented with an extra data structure, the *xapping*, essentially an unordered set of ordered index-value pairs suitable for parallel processing, where each element of each pair may be any CM-Lisp data structure and the index of each pair must be unique within that xapping. Here is the CM-Lisp notation for a xapping that maps names of colors to numbers:

```
{red→14 green→3 purple→93 blue→3.5}
```

For the common and useful special case that the indices are consecutive integers starting from zero, the xapping is also called a *xector*, and a square-bracket notation may be used:

```
[banana apple pear kumquat]
```

is simply an alternate notation for

```
{0→banana 1→apple 2→pear 3→kumquat}
```

A distinctive feature of CM-Lisp is that xappings could be conceptually infinite by having a *default value* considered to be associated with every index not explicitly mentioned:

```
{red→14 green→3 purple→93 blue→3.5 →0}
```

Connection Machine Lisp

GUY L. STEELE JR.

Oracle Labs, Burlington, MA, USA

Synonyms

[CM-lisp](#)

Definition

Connection Machine Lisp (CM-Lisp) is a data-parallel version of Lisp developed around 1987 for Connection Machine supercomputers manufactured by Thinking Machines Corporation. Unlike the *Lisp language, it drew no sharp distinction between front-end data and parallel data, and provided for parallel processing of S-expressions, not just numbers and bit fields. CM-Lisp introduced an aggregate data type called a *xapping*, which was essentially a (not necessarily finite) map

is a xapping that has a pair $x \rightarrow 0$ for every possible CM-Lisp data structure x other than red, green, purple, and blue.

CM-Lisp also adds three notations for parallel processing. The α notation allows elementwise parallel processing; if two or more xappings are processed by an α -expression, then corresponding values are combined by matching up their indices (a process that can be regarded as a simple form of database join operation – simple because no input xapping will have multiple pairs with the same index, and therefore no output xapping will have multiple pairs with the same index). As a simple example, if x names the xapping

```
{red→14 green→3 purple→93 blue→3.5}
```

and y names the xapping

```
{red→4 yellow→5 green→6}
```

then the expression $\alpha (+ \cdot x (* 2 \cdot y))$ produces the value

```
{red→22 green→15}
```

The character “ α ” indicates that the following expression is to be executed “in parallel, as many copies as needed”; the character “ \cdot ” in effect says, “but not this part – just evaluate it once and use the value” (i.e., the following expression should be evaluated just once and is expected to produce a xapping). Thus, in the expression $\alpha (+ \cdot x (* 2 \cdot y))$, there will be many copies of the addition operation $+$ and the multiplication operation $*$, and also of the constant 2, but x and y already have parallel values. For every possible index, the result xapping will have a pair with that index if and only if *all* input xappings have a pair with that index. This use of the α and \cdot characters is syntactically reminiscent of (and intentionally modeled on) the backquote notation of Common Lisp [8, 9], in which the backquote character “`” indicates that a copy should be made of the following data structure, and the comma character “,” in effect says, “but not here – just evaluate it and use the value.”

The second notation uses the character β as a reduction functional: If f is a function of two arguments, then βf is a function that, when given a xapping x as an argument, will use f to combine the values of all the pairs in x pairwise, repeatedly, until a single value results. Thus

$\beta +$ is a function that will sum all the values in a xapping, and $\beta \max$ will return the largest value in a xapping.

The third notation uses the same character β (in an overloaded fashion) as a permutation functional that, in an abstract sense, describes what the Connection Machine router does when passing messages from processor to processor. If f is a function of two arguments, then βf is a function that, when given two xappings x and y as arguments, matches up pairs by their indices just as the α notation does; if x contains a pair $p \rightarrow q$ and y contains a pair $p \rightarrow r$, then the result will contain a pair $q \rightarrow r$. If one regards xapping indices as naming Connection processors (or, to turn it around, if one thinks of a xapping as having a Connection Machine processor associated with each pair), then this describes the action of processor p sending to processor q a message containing the value r . If x happens to contain more than one pair whose value is q , then multiple processors will send messages to the same processor q , in which case the function f is used to combine those values into a single value so as to produce a single pair in the result with index q . (This is why the same character β is used to notate two different operations: In the most general case, they each involve reduction of multiple values to a single value).

The design of CM-Lisp attempts to be completely general in giving meaning to every possible Common Lisp operation when α is applied to it. This leads to a rather intricate theory of control flow and function application that can be described either by a metacircular interpreter or algebraic relationships between α and such Lisp constructs as lambda and if [7].

Figure 1 shows an early (1987) example of a Connection Machine Lisp program that identifies prime numbers less than 100,000 by the method of the Sieve of Eratosthenes, taken (with minor alterations) from [10]. The function `find-primes` takes an argument n indicating the number of integers to be tested; for this example, it should be called as `(find-primes 100000)`. The function `make-xector` makes a `xector` (a xapping whose indices happen to be consecutive integers starting from 0); thus the local variable `candidate` is bound to a xector of length n whose elements are all initially `t` (true), and the local variable `primes` is bound to a xector of length n whose elements are all initially `nil` (false). The function `iota`

```
(defun find-primes (n)
  (let ((candidate (make-xector n :initial-element t))
        (primes (make-xector n :initial-element nil)))
    (value (iota n)))
  (asetf candidate ' [nil nil])
  (do ((next-prime (position t candidate) (position t candidate)))
      ((null next-prime) primes)
    (setf (xref primes next-prime) t)
    α(setf •candidate
          (and •candidate
               (not (zerop (mod •value next-prime)))))))
```

Connection Machine Lisp. Fig. 1 Example Connection Machine Lisp program for identifying prime numbers

creates a xector that maps integers to themselves; thus the local variable `value` is bound to a xector such that element k has value k , for $0 \leq k < n$. The construction `asetf` indicates that `setf` is to be used as many times as needed; the second argument is a xector `[nil nil]` of length 2, and so at most two `setf` operations will be performed (fewer if n is 0 or 1). The effect is to set elements 0 and 1 (if they exist) of the xector in variable `primes` to `nil`. The `do` loop is a conventional Common Lisp `do` loop; it binds, initializes, and steps the variable `next-prime` repeatedly, executing the body in between, until the test expression `(null next-prime)` is true, at which point the result expression `primes` is evaluated and its result is returned. The function `position` is the conventional Common Lisp sequence function `position`, overloaded to process xectors (and implemented using parallel techniques); it returns the index of the leftmost element (i.e., the element of smallest index) whose value matches the given value (in this case `t`), but returns `nil` if no element of the xector matches. Each time that `next-prime` is not `nil`, the two body expressions are executed. The function `xref` is analogous to Common Lisp `aref`, but indexes a xapping rather than an array; thus the expression `(setf (xref primes next-prime) t)` sets one element of the xector `primes`, namely the one selected by the index `next-prime`, to `t`. (One could select many elements in parallel, thereby producing a new xapping or updating many elements of an existing xapping, by using `α` with `xref` in an expression such as `α (xref primes •indices)`). With elements of a xapping spread across

many Connection Machine virtual processors, this construction would perform interprocessor communication.) The second expression in the body of the `do` loop updates elements of the `candidate` xector; because the operands `candidate` and `value` are each preceded by a bullet character “`•`,” they are treated as already parallel; on the other hand, the operand `next-prime` has no bullet in front of it and therefore is automatically replicated (broadcast to all virtual processors). This expression uses the conventional (and idiomatic) Common Lisp special form `and`, but it could (equally idiomatically) have been written using `when`:

```
α (when •candidate
      (setf •candidate
            (not (zerop (mod •value
                           next-prime)))))
```

Either version has the effect of inactivating virtual processors for which `candidate` has the value `nil`, then executing the expression `(not (zerop (mod •value next-prime)))` on whatever virtual processors remain active, then reactivating virtual processors disabled in the first step.

By 1987 there was a working implementation of CM-Lisp that included a garbage collector and a compiler, but it restricted the values of xappings stored within Connection Machine processors to be integers, floating-point numbers, or characters, and did not yet support the execution of nested `α` expressions in parallel [13]. Programming languages whose designs were influenced by experience with CM-Lisp include Paralation Lisp [6], which was by design less abstract than CM-Lisp so

as to admit an efficient compiled implementation [1], and NESL [2], which successfully implemented nested parallel execution of data-parallel operations on nested vectors of differing size [3].

Related Entries

- [C*](#)
- [Connection Machine](#)
- [Connection Machine Fortran](#)
- [*Lisp](#)
- [Nesl](#)

Bibliography

1. Blelloch GE (1990) Vector models for data-parallel computing. MIT, Cambridge
2. Blelloch GE (1996) Programming parallel algorithms. Commun ACM 39(3):85–97
3. Blelloch GE, Hardwick JC, Chatterjee S, Sipelstein J, Zagha M (1993) Implementation of a portable nested data-parallel language. In: PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on principles and practice of parallel programming, ACM, New York, pp 102–111
4. Hillis WD (1985) The Connection Machine. MIT, Cambridge
5. Hillis WD, Steele GL Jr (1986) Data parallel algorithms. Commun ACM 29(12):1170–1183
6. Sabot GW (1988) The paralation model: Architecture-independent parallel programming. MIT, Cambridge
7. Steele GL Jr, Hillis WD (1986) Connection Machine Lisp: Fine-grained parallel symbolic processing. In: LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming, ACM SIGPLAN/SIGACT/SIGART, New York, August 1986, pp 279–297
8. Steele GL Jr, Fahlman SE, Gabriel RP, Moon DA, Weinreb DL (1984) Common Lisp: The language. Digital, Burlington
9. Steele GL Jr, Fahlman SE, Gabriel RP, Moon DA, Weinreb DL, Bobrow DG, DeMichiel LG, Keene SE, Kiczales G, Perdue C, Pitman KM, Waters RC, White JL (1990) Common Lisp: The language, 2nd edn. Digital, Bedford
10. Thinking Machines Corporation (1987) Connection Machine model CM-2 technical summary, technical report HA87-4. Cambridge
11. Thinking Machines Corporation (1989) Connection Machine technical summary, version 5.1. Cambridge
12. Thinking Machines Corporation (1993) Connection Machine CM-5 technical summary, 3rd edn. Cambridge
13. Wholey S, Steele GL Jr (1987) Connection Machine Lisp: A dialect of common lisp for data parallel programming. In: Kartashev LP, Kartashev SI (eds) Proceedings of the second international conference on supercomputing, vol III, International Supercomputing Institute, Santa Clara, pp 45–54

Consistent Hashing

- [Peer-to-Peer](#)

Control Data 6600

JOHN SWENSEN
CPU Technology, Pleasanton, CA, USA

Synonyms

- [CDC 6600](#)

Definition

The Control Data 6600 computer, regarded by many as the world's first supercomputer, was designed by Seymour Cray and James Thornton, introduced in 1964, and featured an instruction issue rate of 10 MHz, with overlapped, out-of-order instruction execution in multiple functional units and interleaved memory banks. A unique "scoreboard" unit controlled instruction issue and execution so as to match the behavior of traditional, sequential execution.

Discussion

Introduction

The state of the art in scientific computing, immediately prior to the Control Data 6600, was represented by the Univac LARC, introduced in 1960, with 20K–96K 48-bit words of core memory, a 4-μs floating-point add time, an 8-μs floating-point multiply time, and a 28-μs floating-point divide time. The LARC was closely followed, in 1961, by the IBM 7030 (also called Stretch), with 16 K–128 K 64-bit words of core memory, a 1.4-μs floating-point add time, a 2.5-μs floating-point multiply time, and a 9-μs floating-point divide time. The IBM 7030 executed 1.2 million instructions per second (MIPS), on average, and occupied 2,000 ft² of floor space.

Three years later, the Control Data 6600 computer offered 64K–128K 60-bit words of core memory, a 0.4-μs floating-point add time, a 0.8-μs floating-point multiply time, and a 2.9-μs floating-point divide time. Its execution rate averaged 3 MIPS, 2.5 times faster than the IBM 7030, but occupied only 200 ft² of floor space.

Architecture

The CDC 6600 instruction-set architecture had several features common to Cray's machines, including:

- A load-store architecture
- Several register types
- An absence of condition-codes
- Three-address instruction formats
- Word-only access to memory
- Non-standard arithmetic
- Delegation of input-output and overhead functions to peripheral processors

These architectural features were chosen to allow Cray's characteristically aggressive clocking, with multiple cycles required, even for simple instructions.

Load-Store Architecture

Load-store architectures require operands to be loaded into registers before they can be operated on, and results must be explicitly stored to memory. With explicit load instructions, support for overlapped instruction execution, and with a sufficient number of registers, it is often possible to schedule long-latency memory loads of operands in advance of their use, so that a computation is not delayed by the access latency. With explicit store instructions, memory bandwidth is only used to save desired intermediate results.

The CDC 6600 was a load-store architecture, but it had no pure load or store instructions. Rather, loads were initiated when one of a subset of address registers was written, and stores were initiated when one of a different subset of address registers was written.

Register Sets

The registers of the CDC 6600 included a set of eight address registers (A registers), a set of eight index registers (B registers, with B0 holding the constant zero), and a set of computation registers (X registers). The A and B registers were 18 bits in length, large enough to address the entire, 128K word memory space. Loop indices and other housekeeping variables could, also, be represented with 18 bits of precision. The X registers were 60 bits in length, and were used to hold fixed-point and floating-point operands.

Loading and storing of registers was through an unusual mechanism; an instruction that wrote to one of registers A1 through A5 caused the corresponding register X1 through X5 to be loaded from memory at the address in the corresponding A register. An instruction that wrote to either A6 or A7 caused the contents of either X6 or X7 to be stored at the address in the corresponding A register. Registers A0 and X0 did not have this behavior.

In addition to the general-purpose A, B, and X registers, the CDC 6600 had an 18-bit program counter, an 18-bit base register for all memory accesses, an 18-bit field-length register for all memory accesses, an exit-enable register, as well as a pair of base and field-length registers for Extended Core Storage accesses.

The program counter could only point to a 60-bit location in memory, even though up to four instructions could be packed into a memory location. This necessitated that every branch target was to the instruction encoded in the high 15 or 30 bits of the destination word.

The base and field-length registers allowed program and data relocation within the memory space of the CDC 6600, supporting efficient sharing of the computer by multiple programs. Other than this linear relocation, no other address translation was performed by the CDC 6600.

The exit-enable register encoded the conditions that could cause a program to exit, and included address-out-of-range, operand-out-of-range, and indefinite-operand conditions. Upon program exit, this register field held the actual conditions at the time of program exit.

All of the registers of the CDC 6600 were saved (and restored) in a 16-word block of memory by the Exchange Jump operation.

Branch Condition Encoding

Conditional branches in the CDC 6600 were based on the states of specified registers (or pairs of registers), rather than the state of a single, condition-code register. Branch conditions include the zero, negative, in-range, and indefinite status of any X register, as well as equality and inequality relations between arbitrary pairs of B registers.

In common computer architectures, a single condition-code register is usually implicitly set by the last arithmetic or logical operation, and sometimes by the last load operation, as well. Typically, it encodes if the last result was zero, was negative, caused a carry-out, or caused an arithmetic overflow. For an implementation with strictly sequential execution, condition codes cause few performance penalties, although determination of a zero result often requires an additional cycle after an arithmetic result is available.

When instruction execution is overlapped, however, the implicitly set, single-condition-code register requires that the condition-setting instruction immediately precede its conditional branch. If branches are not based on the state of a single condition code, useful instructions can often be inserted between a long-executing instruction and the branch depending on its result.

Other approaches to branch conditions include conditional writing the of the condition code (for example, in the Sun SPARC architectures), or multiple condition codes (for example, in the IBM Power and PowerPC architectures). These approaches both require additional opcode space to specify the condition-writing behavior.

Three-Address Instruction Format

Most of the CDC 6600 instructions specified three registers, two source registers, and a destination register. Although requiring a larger instruction word than a two-address instruction format (where the destination register is, also, one of the source registers), a three-address instruction format does not require an extra instruction to move a result. This is particularly helpful when registers are not completely symmetrical, as was the case with the CDC 6600's loadable registers X1-X5 and storable registers X6 and X7.

The opcode field implicitly encoded the type of registers addressed by the instruction, so that each register specifier field was only three bits in length, for a total of nine bits for the three operands, leaving six bits for an opcode in the 15-bit instruction format. A longer, 30-bit instruction format included a 15-bit constant field that was combined with one 3-bit register-specifier field to form an 18-bit constant for branch targets or immediate operands. Note that the 18-bit branch target addressed

a 60-bit word; branches were always to the most significant 15 or 30 bits of the destination addresses. The 15-bit and 30-bit instruction formats are shown below.

Memory Access Sizes

Loads and stores in the CDC 6600 always moved 60 bits; there was no direct architectural support for accessing smaller operands. Furthermore, loads only targeted registers X1-X5, and only registers X6 and X7 could be stored. Loading an operand into an A or B register required a load to an X register, followed by a move to the desired register. If other than the low 18 bits of the memory word were desired, the X register could be shifted, with a separate instruction, before moving the new, low 18 bits of its contents to the desired A or B register.

The advantages to such inconvenient restrictions were that the distribution networks between memory and the registers could be highly optimized and that fewer load and store opcodes were required. These advantages were minor, however, and the architectural trade-off reflects Cray's philosophy that ease of programming was far less important than speed of execution.

Number Representations

Integer arithmetic in the CDC 6600 was performed using a 1's-complement representation, in which arithmetic negation, like logical negation, required inverting each bit of the input operation. Two disadvantages of the 1's-complement representation were that additions and subtractions required that the carry-out from the sum be added into the least-significant bit of the sum (an end-around carry), and that there were two representations for zero (in octal, for 18-bit values, $+0 = 000000$, $-0 = 777777$), although the CDC 6600 fixed-point add units never produced a negative-zero result.

Floating-point numbers in the CDC 6600 had a 1-bit sign, an 11-bit binary exponent (biased by 1,024), and a 48-bit coefficient with the radix point to the right of the least significant bit. Both the exponent and the coefficient were represented as 1's complement numbers. The radix point on the right allowed easy conversion of integers to floating-point values. In addition to finite values, the floating point format encoded positive and

negative infinity (that might be generated via arithmetic overflow or division by zero), as well as indefinite values (that might be generated by dividing zero by zero, or by adding positive infinity to negative infinity). These exceptional results could be tested by branch instructions.

The CDC 6600's floating point representation differed from all other computer families, most notably differing from that of the IBM 360, with its hexadecimal exponent, and sign-and-magnitude fraction. At the time of its introduction, this mattered little, but, over time, the ubiquity of the 360 and 370 architectures meant that most of a growing collection of carefully crafted numerical routines could not be used by programmers of the CDC 6600 and its successor machines.

Peripheral Processors

The central processor of the CDC 6600 was dedicated to computation; input-output operations, job scheduling, and other "overhead" operations were delegated to a set of ten peripheral processors, or PPUs. Although implemented in the same technology as the central processor, and housed within the same cabinet, each of the PPUs executed at a fraction of the central processor's speed and had a very different instruction set architecture, along with its own, private memory.

These PPUs were connected to the computer system's input-output devices and controlled them directly. The PPUs interfaced to the central processor via the system memory. The PPUs could interrupt the central processor by using the Exchange Jump operation, which simultaneously saved the current central processor program context to memory and loaded a new context from memory. In one sense, the central processor was a compute-server slave to the collection of PPUs. In a different sense, the central processor was the star performer, freed from mundane responsibilities by its entourage of support processors.

Instruction Set Listing

The instruction set of the CDC 6600 central processor is listed below. Octal notation is used to represent the six-bit opcodes; where three-digit opcodes are specified, they indicate that the i-field of the instruction further expand the opcode. The value jk is the 6-bit constant formed by the j- and k-fields.

00	Stop
01	Return Jump to K
013	Central Processor-initiated Exchange Jump (special hardware option)
02	Goto K + Bi
030	Goto K if Xj is zero
031	Goto K if Xj is not zero
032	Goto K if Xj is positive
033	Goto K if Xj is negative
034	Goto K if Xj is in range (not infinite)
035	Goto K if Xj is out of range (infinite)
036	Goto K if Xj is definite
037	Goto K if Xj is indefinite (similar to IEEE NaN)
04	Goto K if Bi == Bj
05	Goto K if Bi != Bj
06	Goto K if Bi >= Bj
07	Goto K if Bi < Bj
10	Transmit Xj to Xi
11	Logical Product (AND) of Xj and Xk to Xi
12	Logical Sum (OR) of Xj and Xk to Xi
13	Logical Difference (XOR) of Xj and Xk to Xi
14	Transmit not Xk to Xi
15	Logical Product of Xj and not Xk to Xi
16	Logical Sum of Xj and not Xk to Xi
17	Logical Difference of Xj and not Xk to Xi
20	Rotate Xi left jk places
21	Shift Xi right jk places (with sign-extension)
22	Shift Xi nominally left Bj places (right if Bj negative)
23	Shift Xi nominally right Bj places (left if Bj negative)
24	Normalize Xk to Xi and Bj
25	Round and normalize Xk to Xi and Bj
26	Unpack Xk to Xi and Bj
27	Pack Xk and Bj to Xi
30	Floating point sum of Xj and Xk to Xi
31	Floating point difference of Xj and Xk to Xi
32	Floating double sum of Xj and Xk to Xi
33	Floating double difference of Xj and Xk to Xi
34	Rounded floating point sum of Xj and Xk to Xi
35	Rounded floating point difference of Xj and Xk to Xi
36	Integer sum of Xj and Xk to Xi
37	Integer difference of Xj and Xk to Xi
40	Floating point product of Xj and Xk to Xi
41	Rounded floating point product of Xj and Xk to Xi

42	Floating double product of Xj and Xk to Xi
43	Form jk mask in Xi
44	Floating point divide Xj by Xk to Xi
45	Rounded floating point divide Xj by Xk to Xi
46	Nop
47	Sum of 1's (population count) in Xk to Xi
50	Sum of Aj and K to Ai (load or store Xi if i != 0)
51	Sum of Bj and K to Ai (load or store Xi if i != 0)
52	Sum of Xj and K to Ai (load or store Xi if i != 0)
53	Sum of Xj and Bk to Ai (load or store Xi if i != 0)
54	Sum of Aj and Bk to Ai (load or store Xi if i != 0)
55	Difference of Aj and Bk to Ai (load or store Xi if i != 0)
56	Sum of Bj and Bk to Ai (load or store Xi if i != 0)
57	Difference of Bj and Bk to Ai (load or store Xi if i != 0)
60	Sum of Aj and K to Bi
61	Sum of Bj and K to Bi
62	Sum of Xj and K to Bi
63	Sum of Xj and Bk to Bi
64	Sum of Aj and Bk to Bi
65	Difference of Aj and Bk to Bi
66	Sum of Bj and Bk to Bi
67	Difference of Bj and Bk to Bi
70	Sum of Aj and K to Xi
71	Sum of Bj and K to Xi
72	Sum of Xj and K to Xi
73	Sum of Xj and Bk to Xi
74	Sum of Aj and Bk to Xi
75	Difference of Aj and Bk to Xi
76	Sum of Bj and Bk to Xi
77	Difference of Bj and Bk to Xi

Implementation Technology

The resistor-transistor logic gates of the CDC 6600 were packaged in “cordwood” modules, approximately $2.5'' \times 2.5'' \times 0.8''$ in size, in which pairs of circuit boards were connected by resistors between the boards. With the silicon transistors on the inside, wiring traces on the outside, off-module wiring on one edge, power and mechanical connections to a cooling plate on the opposite edge, each module implemented a high-density, serviceable logical element.

The core memory used by the CDC 6600 was packaged in blocks, approximately $6'' \times 6'' \times 2.5''$, each

holding 4,096, 12-bit subwords, with five memory modules making up a bank of 4,096, 60-bit words. A fully populated CDC 6600 had 32 interleaved banks of memory; each bank had an access time of 800 ns and cycle time of 1,000 ns, and a different bank could be accessed every 100 ns.

Memory reads to core memory were destructive, so that, following a read, the data had to be written back or it would be lost. This characteristic was exploited by the Exchange Jump operation, which saved the current 16-word program context, while simultaneously loading a new program context.

Implementation Features

The major implementation features of the CDC 6600 were its:

- Ten functional units capable of overlapped execution
- An Instruction Stack for caching instructions in loops
- A Scoreboard unit to manage the overlapped and out-of-order instruction execution

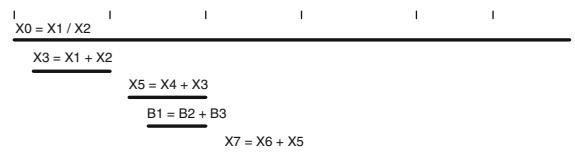
Functional Units

Functional units of the CDC 6600 included:

- A 60-bit Boolean unit (3 cycles)
- A 60-bit shift unit (3 cycles)
- A 60-bit fixed (integer) add unit (3 cycles)
- A 60-bit floating point add unit (4 cycles)
- Two 60-bit floating point multiply units (10 cycles)
- A 60-bit floating point divide unit (29 cycles or 8 cycles)
- Two 18-bit increment (short integer) units (3–11 cycles)
- An 18-bit branch unit (8–15 cycles)

These functional units all required multiple cycles to execute, as indicated, but they were not pipelined; they could only process one set of operands at a time. Two multiply units were provided, because multiplies are relatively common in scientific programming, but slow to execute. Two increment units were, also, provided, because their operations, which included load and store operations, were very common.

One clock cycle was the minimum time required to issue a new instruction, so it was possible, with care, for a program to keep multiple functional units busy simultaneously. Furthermore, differing execution times could cause instructions to complete out of program order. For example, a divide, followed by a series of adds could finish after all of the following adds finished. In the figure below, the divide issues and begins execution in cycle 0, completing in cycle 29. The following floating-point add (to X3) issues and begins execution in cycle 1, completing in cycle 5. The next add (to X5) cannot issue until cycle 6 because the add unit is busy; its input operand is available, so it executes in cycle 6 and completes in cycle 10. The next instruction, an 18-bit add (to B1), issues and begins execution in cycle 7, completing in cycle 10. The final add (to X7) issues and executes in cycle 11, when the add unit is not busy, completing in cycle 15.



The floating point add and multiply units supported both single and double precision arithmetic, but required two instructions to return the more and less significant parts of a double-precision result.

Conversions between integer and floating point operands used the shift unit, sometimes requiring two instructions to complete a conversion.

The divide unit executed floating point divides, as well as a population-count instruction (which counted the number of 1-bits in a word) and a no-operation (nop) instruction. Because of the branch-target restrictions, nop instructions were very common in CDC 6600 programs.

Most branch instructions, besides requiring the branch unit, also required the use of an increment unit or the fixed-point add unit for operand comparison or testing.

Instruction Stack

Instructions for the CDC 6600 were cached in an 8-word instruction buffer (called the Instruction Stack), so up to 32 instructions could be held in fast-access

storage, although loops were limited to, at most, 26 instructions, plus one branch, if they were to remain in the Instruction Stack.

Scoreboard Unit

The Scoreboard Unit enforced sequential execution semantics, while allowing instructions to execute and complete out of program order. For this discussion, an instruction *issues* when it is decoded and sent to a functional unit for eventual execution, an instruction *executes* when it has received all of its operands and begins the specified operation, and it *completes* when it writes its final result(s) to a memory location or to one or more registers. Instructions in the CDC 6600 always issued in program order, although they often executed and completed out of order.

Because the CDC 6600 functional units were not pipelined, a functional unit remained busy from the time an instruction was issued to it until one cycle after that instruction completed. A result register remained busy from the time its instruction issued until it completed.

During the issue process, each instruction was decoded and its required functional unit(s) and destination register(s) were determined. If a required functional unit was unavailable, or if the destination register was busy, issuing stalled until all were available. These were called first-order conflicts by the designers of the CDC 6600. Once the first-order conflicts disappeared, the instruction was sent to the required functional unit(s), along with the source and destination register specifiers. This enabled the issue process for the following instruction to begin.

If all required input registers were available, execution began; otherwise, the functional unit waited for them to become available. Unavailability of input registers was called a second-order conflict; this tied up a functional unit, but allowed other instructions to continue executing.

Execution proceeded and, one cycle before the result was ready, the functional unit signaled that it was ready to release its result to the destination register. If all release datapaths were busy, or if one or more previously issued instructions had not yet used the older value in the destination register, completion stalled. Inability to write a result was called a third-order conflict; like a second-order conflict, it tied up the functional unit, but

did not affect the execution of instructions that did not depend on the uncompleted instruction.

The Scoreboard Unit operated on state information associated with the functional units and the registers. With each functional unit (FU) was associated:

- F_m – function to be performed by the unit (e.g., integer add, integer subtract)
- F_i – the destination register for the result
- F_j – the first source register for the function
- F_k – the second source register for the function
- Q_j – the functional unit producing the result in F_j
- Q_k – the functional unit producing the result in F_k
- RF_j – a bit indicating that F_j was ready
- RF_k – a bit indicating that F_k was ready
- XS – a bit indicating that the FU had begun execution
- RQ – a signal indicating that the FU was requesting to release its result
- RL – a signal indicating that the FU was releasing its result
- AC[] – a set of bits indicating that registers F_j and F_k had not been read

Also, copies of the current input operands were kept within each functional unit.

With each register r in {A0-A7, B0-B7, X0-X7}, was associated QR, the functional unit or memory read storage channel that would, eventually, write its data, or an indication that the register had its data. With each register r was associated AC, or All Clear, indicating that no functional unit reading that register had started execution.

The issue process can be described by the following algorithm, operating on each instruction p in succession:

1. Decode instruction p to determine the functional unit FUp, F_m, F_i, F_j, and F_k.
2. Use F_j and F_k to look up values for Q_j, Q_k in QR[].
3. While FUp is busy or register F_i is busy, stall.
4. Send F_m, F_i, F_j, F_k, Q_j, Q_k to FUp.
5. If F_j is not waiting for a FU result, set RF_j and transmit register F_j to FUp, otherwise clear RF_j.
6. If F_k is not waiting for a FU result, set RF_k and transmit register F_k to FUp, otherwise clear RF_k.
7. Clear AC[FUp][F_j] and AC[FUp][F_k].
8. Set QR[F_i] to FUp.

Notes on the issue process:

- Steps 2 and 3 depend on step 1, but can execute during the same processor cycle as step 1.
- Step 3 could repeat for many cycles, given a long chain of instruction dependencies.
- Once Step 3 is past, Steps 4, 5, 6, 7, and 8 can proceed in parallel.

The execute process can be described by the following algorithm, operating in each functional unit in parallel:

1. If RF_j is clear and RL[Q_j] is active, load operand j with the result on the release datapath and set RF_j.
2. If RF_k is clear and RL[Q_k] is active, load operand k with the result on the release datapath and set RF_k.
3. If RF_j is set and RF_k is set, start execution, set AC[F_j] and AC[F_k], and set XS.

Notes on the execute process:

- Steps 1 and 2 execute in parallel.
- The release datapaths for operand j and operand k are either independent, or RL[Q_j] and RL[Q_k] are asserted on different cycles, so no release datapath conflicts can occur.

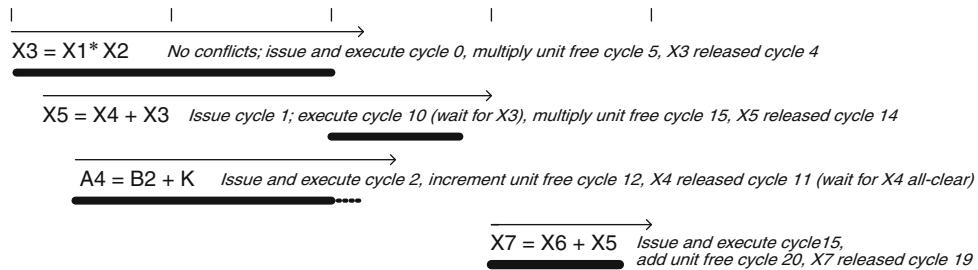
The result release process can be described by the following algorithm:

1. One cycle before completing its operation, a functional unit asserts RQ.
2. If any AC[*][F_i], for all relevant functional units, is clear, delay the grant.
3. If a functional unit of higher priority for the release datapath is asserting RQ, delay the grant.
4. Otherwise, grant the release request.
5. The following cycle, the functional unit asserts RL and writes its result onto the release datapath to the appropriate register.

6	3	3	3
Opcode	i	j	k

6	3	3	18
Opcode	i	j	K

Control Data 6600. Fig. 1 Issue and execute timing example for dependent and independent instructions



Control Data 6600. Fig. 2

Notes on the release process:

- Because the RQ signal is sent one cycle early, there is time to detect release conflicts and, if there are none, the release can be granted without delaying the functional unit.
- Any other functional units waiting for the result can select it from the release datapath, saving a register read.

A simple example illustrates some of the scoreboard functions. Assume that all four instructions have been loaded into the Instruction Stack. The multiply to X3 has no conflicts, so it issues and begins execution in cycle 0. The multiplier remains busy through cycle 11 (indicated by the light arrow), and execution runs from cycle 0 through cycle 10 (indicated by the bold line please see Fig. 2).

The add to X5 issues in cycle 1 because both X5 and the adder are free. However, the operand X3 is not available until cycle 10, when the add begins to execute. The adder releases X5 in cycle 14 and the unit becomes free in cycle 15.

The 18-bit add to A4 (causing a load to X4) issues in cycle 2 because X4, while an input operand to a previous, uncompleted instruction, was copied into the add unit at cycle 1. The release of X4 is delayed until after X4 receives the all-clear signal (in cycle 10, when the add started execution). X4 is released in cycle 11 and the increment unit becomes free in cycle 12.

Issuing of the add to X7 stalls from cycle 4 until cycle 15, when the add unit becomes free. X5 is available by then so execution also begins in cycle 15, releasing X7 in cycle 19, and freeing the adder in cycle 20.

The scoreboard unit enforced the serial dependency of X3 -> X5 -> X7, allowed the load to X4 proceed in parallel, but prevented the load to X4 from completing

before the add to X5 began. Sequential execution would have required an additional eight cycles to complete.

For all its apparent complexity, the Scoreboard Unit required fewer gates than the average functional unit in the CDC 6600 [5]; nevertheless, this was the only computer to use such a unit. A more complex algorithm, the Tomasulo algorithm, used in the IBM 360 Model 91, provided faster execution for poorly scheduled and register-allocated code [6]. A simpler algorithm, which only tracked pending writes to registers, was used in Cray's later, pipelined machines, allowed faster instruction issuing and, with good instruction scheduling and register allocation, allowed instruction execution as fast as did the Scoreboard Unit of the CDC 6600.

Derivative Machines

The Control Data 6400 employed the same technology as the CDC 6600, but only supported sequential execution of instructions in a single, non-pipelined execution unit. In the Control Data 6500, two 6400 central processors shared a single, interleaved memory system.

In 1969, the Control Data 7600 succeeded the CDC 6600 as the fastest computer in the world. It executed the same instruction set in multiple, fully pipelined, functional units, with an instruction issue rate of 36 MHz.

Bibliography

- Control Data Corporation (1965) Control data 6400/6500/6600 computer systems reference manual, Publication No. 60100000. St. Paul, Minnesota
- Control Data Corporation (1966) 6600 central processor, vol 2: functional units, publication No. 60239700 (formerly Publication No. 091466). St. Paul, Minnesota
- Control Data Corporation (1967) 6600 central processor, vol 1: control and memory, publication No. 010167. St. Paul, Minnesota

4. Thornton JE (1964) Parallel operation in the Control Data 6600. In: AFIPS Proc. FJCC, pt 2 vol 26, pp 33–40
5. Thornton JE (1970) Design of a computer the Control Data 6600. Scott, Foresman, and Company, Glenview (out of print, but available, online, at http://www.bitsavers.org/pdf/cdc/cyber/books/DesignOfAComputer_CDC6600.pdf)
6. Tomasulo RM (1967) An efficient algorithm for exploiting multiple arithmetic units. IBM J 11:25–33

Coordination

- Path Expressions

Copy

- Broadcast

Core2-Duo / Core2-Quad Processors

- Intel Core Microarchitecture, x86 Processor Family

COW

- Clusters

Crash Simulation

M'HAMED SOULI¹, TIMOTHY PRINCE², JASON WANG³

¹Université des Sciences et Technologies de Lille,
Villeneuve d'Ascq cédex, France

²Intel Corporation, Santa Clara, CA, USA

³LSTC, Livermore, CA, USA

Definition

During the creation and conception of a new car model, safety engineers from automotive industry perform crash simulation, in order to evaluate the level of safety of a car and its occupants. During a car crash, the kinetic energy of the car, $E = 1/2M.v^2$, that the vehicle has

before impact, is transformed into deformation energy. Crash simulation indicates the capability of the material used for the design, to absorb this energy and protect the occupant. Most important results are the deceleration felt by the occupants, which must fall below threshold values fixed in legal car safety regulations [1]. To ensure driver safety during a car crash, and meet the regulations in order to get the official approval and homologation of a new car model for road services, car manufacturers need to perform crash tests. Race cars need also to meet requirements specified by the FIA, “Fédération Internationale de l’Automobile” [2]. Safety regulations require several physical crash tests that must be performed on a new model. These tests are extremely expensive; to reduce the number of physical crash tests, and also the product development lead time and costs, engineers need to carry out a range of virtual tests, or crash simulations, using crash codes. These codes are based on explicit finite element method well suited for analyzing nonlinear dynamic response of structures. The ability of crash codes is to effectively handle material nonlinearity, and nonlinear behavior such as contact. To perform a crash simulation in a reasonable time scale, safety engineers use High-Performance Computing (HPC) that allows to perform several crash simulation, during a day, from frontal crash to components impact simulation. Nowadays, automotive industry relies heavily on simulations; by using simulation, the number of real test crash is reduced, but computer simulations cannot fully replace the crash test, since a crash test is required in the final stage of development to validate numerical results. Simulations are compared to test data using high-speed camera for visualization, and accelerometers that are placed at different locations on the dummy. Before the first prototype is built, a new car model goes through thousands of computer simulations, crash and components system impact simulations. When engineers conduct the physical crash tests, the model has already achieved a high standard performance through computer simulations. In automotive industry, car crash simulation is the most computer time-consuming task, thus the need to use parallel computing. With adoption of HPC technology, new systems can perform crash simulations using clusters, an assembly of several computers running in parallel, “parallel computing,” that can achieve a speed that is proportional to the number of CPU's in the parallel

system. In a crash simulation, the most critical part of the parallel simulation is the contact handling, new contact search algorithms based on bucket sort have been developed that lead to better scalability. A detailed description of the performance of contact algorithm is described in [3].

Discussion

To ensure driver safety during a car crash, impact structures are designed and optimized to absorb the kinetic energy during the crash, and limit decelerations acting on parts of human body like knees and necks. Decelerations values have to meet safety requirements specified by government regulations. To meet product development schedule of new car models, it is necessary to use parallel processing for crash simulation. Vector processing has been in use from the beginning of practical crash simulation. Vector processing which permits the performance of several calculations simultaneously was introduced as an extension to serial processor [4], to reduce computer time for crash simulation. Vector processing techniques, first developed for supercomputers for high-performance applications, were commonly used in crash simulation in the 1980s and 1990s.

HPC remained expensive and limited because of single CPU system. In recent years, multiple CPUs and multiple cores have offered increase of multiple performance and lower cost for crash simulation and other engineering simulation tasks. Later, in order to extend execution of data parallel processing constructs as DO loop, IF loop analysis, and nested loops in the Fortran language, SMP (Shared Memory Processing) version of crash code has been developed.

In the SMP technology, multiple CPUs share same memory. Since CPUs are accessing shared memory simultaneously, data supplied from memory to CPUs is likely to be slow, when the number of CPUs increases, which limits the number of CPUs that can be used efficiently. However, it has been observed that the scalability of the SMP version of crash codes was observed to stop at eight processors [3]. To overcome this problem, and solve the limitation of shared memory processing, the MPP (Massive Parallel Processing) version of crash codes has been developed, which uses distributed memory. In MPP, each small group of CPUs has its own memory, and CPUs are interconnected via a high-speed

networking system. MPP version of crash codes as LS-DYNA and other crash codes has been performing with scalability up to hundred and even thousands of processors; however, it has been observed that interconnected speed is a significant limiting factor in the scalability of MPP performance [3].

When using the MPP version, prior to computer processing, domain decomposition needs to be performed to divide the problem. In domain decomposition, the model is decomposed into subdomains; each subdomain is processed by a CPU and uses the memory dedicated to that CPU.

Current compilers are not yet capable of automatically translating an SMP version of a crash code that runs on shared memory into an MPP version that runs efficiently on distributed memory. MPP requires a certain amount of development that needs to be performed jointly by developers from computer and software development companies.

Several computer companies, such as Fujitsu in Japan which has been involved from the beginning in the vectorization and parallelization of crash simulation codes, first started developing SMP version of crash codes using OpenMP language, and then moved to MPP using MPI library (Message Passing Interface). Today, most engineering simulations are performed on Quad core or more powerful CPUs.

Few years ago, simple simulation events, such as a small model using a few dozen elements and representing an event that lasts 100 ms, required a day or more on a vector supercomputer to complete. Today, by using low-cost parallel computing, safety engineers can model large-scale crash car to car simulation with airbags and occupants in just a few hours. Nowadays, all car companies have incorporated parallel computing into their car design.

Historical Performance Trends

The primary goal of crash simulation is to improve safety for human occupants and evaluate design improvements to enable vehicles to score well on crash tests performed by regulatory and insurance agencies. In order to improve the accuracy of crash test safety analysis and to represent vehicle geometry more accurately, the number of finite elements in analysis models has been continuously increasing.

Crash safety numerical simulation is performed primarily by explicit time marching, in which the entire evolution of a crash from initial impact to final resting state, about a tenth of a second, is calculated in steps of about 1ms. During crash simulation, different parts of the car get in contact with themselves and each other, this can be handled by using contact algorithms. Contact analysis is an important feature in crash simulation, and continuing research for efficient parallelization of contact algorithms is still ongoing. A detail description of the performance of contact algorithm is described in [5]. Crash simulation is performed using finite difference method for time integration, and Finite element method FEM, for space discretization [6]. Deformations, velocities, and forces on all relevant parts of the vehicle and of dummies simulating the danger of injury to human occupants are calculated at each time step. To simulate a crash and occupant safety, crash analysis software, including LS-DYNA, PAM-CRASH, and other commercial and academic codes, must be able to handle large deformation, material nonlinearity, and complex contact conditions among multiple components [7]. The software must be able to simulate different type of car crash events, as frontal, side, and rear impact.

In the early 1990s, crash simulation would typically be performed on Symmetrical Multi-Processor supercomputers, with 36,000 elements considered a large model. In year 2001, several car manufacturers had adopted MPI cluster computing [8] for crash simulation, using CPUs such as AMD Athlon K7 [9], Intel Xeon DP, or Itanium. At that time, high-speed

Interconnect system did not support more than 24 CPUs efficiently, thus a simulation with 500,000 elements would be the most detailed, suitable for overnight completion.

Also, by that year, the detail and computational cost of a useful practical analysis far exceeded the Dodge Neon benchmark (535,000 elements), the only benchmark quoted the first year. This detail of the car crash model of the benchmark is fully described in the next section, (elements number, nodes number, etc.). The benchmark has been completed on several commercial analysis codes of similar nature, all available for parallel computing. For this car model, we compare the performance of high-end dual CPU technical workstations quoted in 2003 against a consumer product single CPU quad core desktop of 2009. From Tables 1 and 2, we observe that not only the performance has been improved by a factor of six, the computer system in Table 1 is five times more expensive than the computer used in Table 2. Consequently, cost and performance of minimum parallel computing platforms for crash simulation each improved by more than a factor of 6 over 6 years time.

These quotations are for the SMP (OpenMP) single node threaded analysis, at least in the case where that information was provided in the submission.

Tables 3 and 4 compare MPP performance cluster results quoted in 2003 and 2009, using IBM and Intel CPUs; Table 3 represents the highest performance reported in that year. From these tables, we can conclude that for clusters, performance has been increasing at least by a factor of 4 between the years 2003 and 2009.

Crash Simulation. Table 1 Computing Time for HP and IBM with 2 CPU

Vendor	Computer	Processor	Total CPU	Time (s)	Benchmark	Date
HP	HP-UX Itanium2 Cluster	1.5 Ghz Itanium2 Rx2600	2	9,397	Neon	11/12/2003
IBM	IBM p655	1.7 Ghz Power4+	2	9,701	Neon	11/13/2003

Crash Simulation. Table 2 Computing Time for ARD with 4 CPU

Vendor	Computer	Processor	Total CPU	Time (s)	Benchmark	Date
ARD	CA92121i	1.7 Ghz Power4+	4	1,565	Neon	02/12/2009

Crash Simulation. Table 3 Computing Time for IBM with 32 and 64 CPU and HP 32 CPU

Vendor	Computer	Processor	Total CPU	Time (s)	Benchmark	Date
IBM	IBM p655	1.7 Ghz Power4+	32	885	Neon	11/13/2003
IBM	IBM p655	1.7 Ghz Power4+	64	667	Neon	11/13/2003
HP	HP-UX Itanium2 Cluster	1.5 Ghz Itanium2 RX2600	32	1,041	Neon	11/13/2003

Crash Simulation. Table 4 Computing CPU Time for SGI and INTEL with 64 CPU

Vendor	Computer	Processor	Total CPU	Time (s)	Benchmark	Date
SGI	Altix ICE8200EX	INTEL XEON Quad Core X5570 2.93 Ghz	64	157	Neon	03/31/2009
INTEL	Supermicro X8DTN	INTEL XEON Quad Core X5560 X5560	64	168	Neon	03/27/2009

Crash Simulation. Table 5 Computing Time for INTEL and CRAY with 64 CPU

Vendor	Computer	Processor	Total CPU	Time (s)	Benchmark	Date
INTEL	S5000PAL	INTEL XEON Dual Core X5560	64	31,213	Car2car	05/31/2006
CRAY	CRAY XT3	AMD Single Core 2.4 Ghz	64	40,935	Car2car	04/27/2006

Crash Simulation. Table 6 Computing Time for SGI and CRAY with 64 CPU

Vendor	Computer	Processor	Total CPU	Time (s)	Benchmark	Date
SGI	Altix ICE8200EX	INTEL XEON Quad Core X5570 2.93 Ghz	64	15,720	Car2car	03/31/2009
CRAY	CX1	INTEL XEON Quad Core X5570 2.93 Ghz	64	16,325	Car2car	06/31/2009

In 2009, much larger clusters could be purchased for the price of clusters quoted in 2003.

As reported in TopCrunch [10], a project initiated to track the aggregate performance trends of high-performance computer systems and engineering software [10], InfiniBand™ adapters and switches had proven cost-effective for cluster communication, and substantial improvements in simulation cost and

performance are demonstrated. In the TopCrunch report, the car2car benchmark model using close to six million elements (5,800,000 elements) was introduced in 2006 to represent better the detail of simulation required in practice at that time.

In Tables 5 and 6, we compare the performance of realistic size models for clusters installed in 2006 and 2009. In these two tables, it is shown that in 3 years time,

computational performance doubled, while cutting the number of cluster nodes in half, using the same total number of CPU cores.

In the performance quotations referred in [Tables 1–6](#), an increasing number of cores has accounted for a major part of enhanced performance. This trend may be expected to continue.

Recent development in parallel computing combines both SMP and MPP technologies, called hybrid OpenMP/MP. The hybrid model may see increased usage as the number of cores within shared memory nodes increases, as it should moderate growth of memory size and communications load with increasing number of threads.

Practical Benefit of Improved Parallel Computation

In 2003, only a few passenger vehicles, aimed toward the safety conscious market segments, had been designed by the aid of crash simulation, and achieved good ratings on various governmental and insurance industry tests. By 2009, all vehicles marketed internationally had published crash survival ratings, most of them substantially improved, while the required tests had become more comprehensive. Cost-effective parallel computing enabled this degree of success in product introductions.

CPU vendors like to tell how, if the automotive industry had made progress equivalent to the computing industry, we would all be driving vehicles faster and safer than bullet trains at less than the price of a single train ride. In fact, the safety record of current vehicles has improved by several times in less than a decade, due directly to the success of their manufacturers in computational design and product improvement, with no adverse impact on consumer acceptability.

For the simulations, the models have been improved and details taken into consideration by increasing the number of elements in the model.

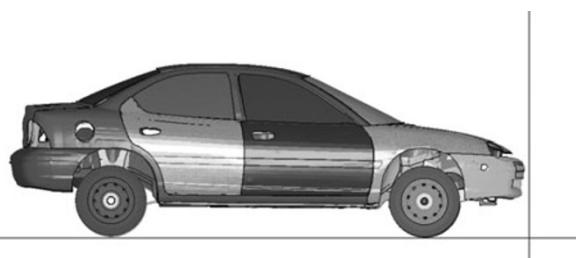
- 1986: First model had 3,439 elements
- 1990: 1.5×10^4 – 2.00×10^4 elements
- 1995: 5.0×10^4 – 10.0×10^4 elements
- 2000: 10×10^4 – 25.0×10^4 elements
- 2005: 1.0×10^6 – 1.50×10^6 elements

In the near future, in order to have an accurate model, the number of elements in a full scale crash will

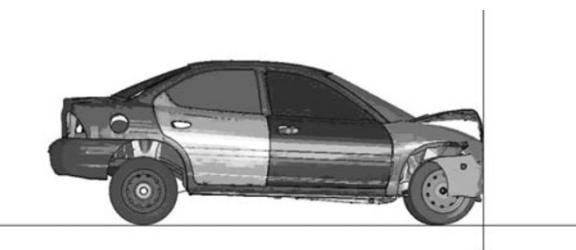
exceed ten million elements. All current simulations performed on clusters.

Description of the Dodge Neon Benchmark

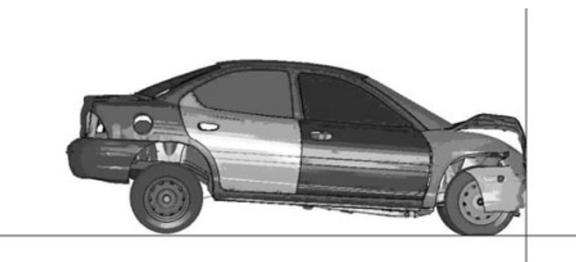
The first test case illustrated here is a vehicle crash-worthiness application of vehicle impacting into a rigid barrier. The digitized vehicle model, a Dodge Neon, is developed by the National Crash Analysis Center at the George Washington University. The model is developed through reverse engineering process that involves systematic disassembly of the physical vehicle to finite element model of each and every part in the vehicle. The geometry data for each part is converted to an FE (finite element) mesh and carefully reassembled with the appropriate consideration of connections and constraints between elements to create a full vehicle FE model. A series of material level characterization tests,



Crash Simulation. Fig. 1 Car deformation at time $t = 0$ ms



Crash Simulation. Fig. 2 Car deformation at time $t = 40$ ms



Crash Simulation. Fig. 3 Car deformation at time $t = 80$ ms

on coupons extracted from various locations of the vehicle, are performed to gather the required input for the material models in FE program. The fully assembled model of the vehicle consisting of 535,070 elements is shown below with model statistics listed.

Number of components	324
Number of nodes	291,206
Number of shells	532,077
Number of beams	73
Number of solids	2,920
Number of elements	535,070

Run have been performed using LS-DYNA MPP version. The following pictures show the car deformation at time $t = 0$, $t = 40$ ms, and $t = 80$ ms (Figs. 1–3).

Bibliography

1. Zienkiewicz OC, Taylor RL (1967) The finite element method for solid and structural mechanics. McGraw Hill, New York. ISBN 978-0-7506-6321-2
2. Heimbs S, Strobl F, Middendorf P, Gardner S, Eddington B, Key J (2009) Crash simulation of an F1 racing car front impact structure. In: 7th European LS-DYNA Conference, Salzburg, Austria, May 2009
3. LS-DYNA theoretical manual (1998) Livermore Software Technology, Livermore, CA
4. Kondo K, Makino M (2008) Crash simulation of large number of elements by LS-DYNA on highly parallel computers. Fujitsu Sci Tech J 44(4):467–474
5. Yih-Yih L, Wang J (2009) Performance of the hybrid LS-DYNA on crash simulation with multiple core architecture. In: 7th Europena LS-DYNA Conference, Salzburg, Austria
6. Li-Xin G, Gong J, Jin-Li L (2010) Three dimensional finite element modeling and front crash process analysis of car bodywork. Appl Mech Mater 44–47:920–923
7. Haug E, Scharnhorst T, Du Bois P (1986) FEM-Crash, Berechnung eines Fahrzeugfrontalaufprall. VDI Berichte 613:479–505
8. <http://www.mpi-forum.org>
9. <http://en.wikipedia.org/wiki/Athlon>
10. <http://topcrunch.org/>

Cray MTA

- [Tera MTA](#)

Cray Red Storm

- [Cray XT4 and Seastar 3-D Torus Interconnect](#)
- [Cray XT3 and Cray XT Series of Supercomputers](#)

Cray SeaStar Interconnect

- [Cray XT3 and Cray XT Series of Supercomputers](#)

CRAY T3E

MICHAEL DUNGWORTH, JAMES HARRELL, MICHAEL LEVINE, STEPHEN NELSON, STEVEN OBERLIN,
STEVEN P. REINHARDT

Synonyms

OS jitter

Definition

This entry describes the hardware and software architecture of the CRAY T3E Massively Parallel Processor (MPP), a landmark supercomputer system that became the first commercially successful MPP, and the first to be used in production data centers around the world. It discusses the historical context leading to the development of the T3E and its predecessor system, the CRAY T3D, and the significance of the T3E to the computational science and engineering community.

Overview

The Transition from Vector Systems to Massively Parallel Systems

In 1976, the first CRAY-1 supercomputer was shipped to Los Alamos National Laboratory, setting a new benchmark for general purpose supercomputing. Although Cray Research was a very small company with less than 100 employees at the time, Seymour Cray and a small team had already started down what would turn out to be a rather difficult path to design a successful follow-on product, the CRAY-2. The success of the single-processor CRAY-1 came from a combination of high-performance chip and packaging technology together with an effective streaming vector architecture. Operations achieved a large parallel speedup, provided that long sequences of data could stream through highly segmented “vector pipes” consisting of special vector pipeline registers feeding highly segmented functional units. This architecture was a natural fit for many



problems that simulate extensive n-dimensional computational space. However, successful speed-ups did require management of an extra layer of architectural complexity – the efficient loading and unloading of the vector registers from, and to, main memory.

Seymour Cray always saw simplicity as a mark of architectural elegance. So for the new project, he naturally began to think about alternatives to the vector registers. Perhaps these registers could be eliminated and performance could be more than made up by a large number of CPUs sharing a common memory space. (In current parlance multiple integrated CPUs are now usually called “cores.”) So it was that the first iteration of the CRAY-2 design would have up to 64 scalar “M” processors, each handing off intermediate 64-bit results to similar processors throughout a shared-memory architecture. I/O work would be managed by a cluster of 16-bit “A” processors also connected to this shared-memory system. Significant architectural and detailed logic designs for the “M” and “A” processors were completed, and both an experimental operating system and a Fortran compiler were running in a simulated environment.

Then one day this entire phase of the CRAY-2 work came to an abrupt halt. It had become apparent after consultation with users of the CRAY-1 that there were huge challenges to effectively program the proposed CRAY-2 for anything but the most regularly organized problems. Memory management and data coherence issues were unfamiliar and intimidating. Effective compiler technology even for the vector-based CRAY-1 was only at an immature state at the time. The CRAY-2 project was in very real danger of having operating hardware available but without nearly adequate system software structures and with no clear timely path to discovering them. For a small new company in need of a strong successor story to help sell its first products, this was not a promising formula for success. There would be more starts and restarts of the CRAY-2 project before it could successfully fill a new application space using a moderate number of vector processors but with a competitively larger memory subsystem that was able to exploit dense dynamic memory chip technology. In fact, the first production.

Cray designers were sensitive to the notion that there are CPUs that run “fast” and then there are computers that solve problems *fast*, and it was becoming clear that fast CPUs would be single chip

microprocessors with on-chip multi-level cache. The personal computer and workstation markets embraced this solution, which drove down production costs by orders of magnitude, but there were serious difficulties for supercomputer designers trying to exploit this opportunity. High production volumes precluded any alterations to the internal design of the CPU for the very high performance market. Due to a very limited pin count, the balance of functional unit performance with the delivery of data from outside the device was seriously tilted in the wrong direction. If that were not enough, there was one more hurdle: The single chip processors were getting high performance in no small measure from onboard cache subsystem. This created a nightmare for managing coherence of data across a large multiprocessor system. Yet the microprocessors were becoming impossible to ignore as internal clock rates were becoming incredibly fast and at ever lower costs as generation after generation followed Moore’s Law. From the supercomputer designers’ perspective these upstarts were extremely frustrating!

The CRAY T3D was the first serious attempt to have at least some success in influencing the I/O properties of a commercial 64-bit microprocessor so that large numbers of processors could cooperate efficiently. Additional special purpose hardware around each CPU to fill in the gaps. CRAY T3E came incrementally closer to the mark. Eventually and inevitably, microprocessors for desktop computers hit some of the same fundamental obstacles that supercomputer designers had faced decades earlier. Desktop (and even notebook) CPUs became multi-core. Seymour Cray’s early hardware vision was at last shared, and its challenges faced, by a much broader spectrum of computer systems designers.

Equally critical to the evolution of massively parallel systems were significant advances in operating system, compiler and applications software technology. The success of the CRAY T3E was due in large measure to a unique collaboration of hardware, system software, and user applications designers from academia, government laboratories, and private industry. The programming obstacles encountered by Seymour Cray’s first CRAY-2 project had finally been overcome.

The Cray T3E

The Cray T3E, which first shipped in 1996 to the Pittsburgh Supercomputing Center, was Cray Research,

Inc's second-generation MPP supercomputer, the follow-on product to the first-generation Cray T3D. The T3D and T3E were the first Cray Research supercomputers to use commodity processors instead of in-house-designed custom CPUs. The T3E was also a departure from the T3D because it was self-hosted, had a very different interconnect architecture, and an entirely new I/O subsystem.

In a technology world driven by Moore's Law, the T3E enjoyed a relatively long service life: It remained in production for over 5 years with some systems still in use in 2010. It was popular with users primarily because of its good communication characteristics and its high ratio of bandwidth between processors, relative to the performance of the processors when compared to the ratios achieved on simple clusters. This made it much easier for users to attain high efficiency and good scaling performance on parallel applications. In addition, the large distributed memory proved very valuable to users. The T3E was also valued by data center managers for the efficiency of its operating system and the high utilization that could be sustained on high-performance computing (HPC) production workloads.

The T3E was "self-hosted," running a distributed version of Cray's Unix operating system, UNICOS/mk. UNICOS was the Unix port to Cray hardware that had been in use since the early 1980s on the CRAY-1, CRAY-2, and all later models of Cray parallel-vector systems. The/mk designation was taken from Chorus, a company that produced a Unix microkernel, which was used as the basis for the Unicos microkernel and server structure. UNICOS/mk was unique because it incorporated some of the latest operating system ideas from both microkernels and servers and also maintained the UNICOS user interface across a distributed memory. Essentially, the system was standard UNICOS reorganized into servers so as to support different varieties of Cray hardware, different scales of system size, and all the features needed to meet existing requirements for usability, performance, and manageability.

The distributed nature of both the operating system and the hardware allowed UNICOS/mk to provide improvements in resiliency. Distribution of OS services simplified the software that ran on any particular processing element (PE) and made it easier to isolate problems. The resultant simplification also made the software more robust. Concerns about the reliability

impact of the significant increase in the number of hardware components necessitated by the T3E architecture had generated a requirement that the T3E should be able to ride through PE failures. But, because UNICOS/mk was conceived as a distributed system, a compute PE failure was easily managed.

The T3E software architecture had two main components: "system" PEs and "application" PEs, whose sole purpose was to run user applications code. There were three specialized types of system PEs: OS PEs, Command PEs, and I/O PEs. Command PEs were used for user login and job launch. The user interface on a command PE was the standard UNIX interface. The operating system ran only system PEs distributed throughout the machine at a ratio of 1 system PE for every 16 application PEs. System PEs could also flexibly be substituted for failed user PEs as necessary to maintain a full complement of application PE's. The ability to map in replacement PEs and eventually repair, reboot, and reintegrate failed PEs, without the need to reboot the whole system, was a tribute to the effectiveness of the distributed hardware and software architecture.

The CRAY T3E was available in two versions that varied by cooling technology and system scale. The liquid-cooled (LC) version scaled to 2048 user processing elements (PEs) while air-cooled (AC) systems scaled to 128 user PEs. The first system was introduced using the 300 MHz version of the DEC Alpha EV5 microprocessor. This first release was later dubbed the T3E-600 (referring to the peak MFLOPS performance of each Alpha processor). Subsequent product versions tracked Moore's Law improvements in the performance of the microprocessor and the EV5 was supplanted by the advanced-process EV56 (21164A) resulting in increased clock speeds of 450 MHz (T3E 900), 600 MHz (T3E 1200), and 675 MHz (T3E 1350). Memory per PE could be configured in sizes ranging from 64 Mbytes to 2 Gbytes, for a maximum aggregate system memory of 4TBytes.

In traditional Cray style, the T3E was designed to be a balanced system and, courtesy of the CRAY Model F I/O subsystem, could be configured with prodigious I/O throughput that was commensurate with its computational capability. This distributed I/O system used Cray's GigaRing network channel to string together chains of I/O PEs hosting channel adapters attached to external disks, tapes, and communications networks in

fault-tolerant ring topologies. Each GigaRing channel was capable of sustained data transfer rates of over 900 Mbytes/s and the system could employ 1 GigaRing for every 4 PEs in the system.

In 1998, a 1280 PE T3E-1200 became the first computer to sustain more than 1 TFLOPS (Trillion Floating Point Operations Per Second) on a real application code, LSMS (Locally Self-Consistent Multiple Scattering), a first-principles metallic magnetism simulation code written by Oak Ridge National Laboratory and optimized by Pittsburgh Supercomputer Center for the T3E. An earlier MPP system, known as ASCI Red, was already installed at Sandia National Laboratory, and in June 1997 was the first system to demonstrate a *peak* performance of 1.3 TFLOPS, but the T3E was able to deliver *sustained* performance in excess of 1 TFLOPS from a system with a peak performance of 1.54 TFLOPS.

Architecture

Processor Choice

Initial work on the T3E architecture and design started in 1992, more than a year before the first shipment of its predecessor, the T3D. Although the T3D used the DEC EV4 microprocessor and Cray was well aware of DEC's next-generation, more powerful, EV5 design they nonetheless investigated alternative processors for the T3E and initially favored a design being proposed by the start-up company MicroUnity. That processor, code-named Terpsichore, would have been implemented in MicroUnity's own Bi-CMOS technology and was designed to use several advanced single instruction multiple data (SIMD) architectural features to enable very high performance on regular vectors of data. Terpsichore was connected to memory and peripheral chips using multiple high-speed ring networks derived from the emerging Scalable Coherent Interface (SCI) channel standard. If successful, it would have run at a then unheard-of 1 GHz clock frequency and would have provided the very large address space and latency-hiding features necessary for large-scale distributed systems. Unfortunately manufacturing process problems, excessively high power dissipation in test chips, and pressure from majority investors to target wider, more-easily attained markets, such as television set-top media processors, caused delays and ultimately abandonment of the Terpsichore chip. Consequently, Cray quickly

started over with the DEC EV5, but some concepts from the MicroUnity experience, notably the aggressiveness of the SCI channel signaling and ring architecture, significantly influenced Cray's designers and were incorporated into the GigaRing I/O channel and the design of the T3E communications link.

The DEC Alpha EV5 superscalar processor was a large step over its predecessor. Many of its performance characteristics that were important to the T3E were a result of improvements to its memory interface, including support for up to two outstanding 64 byte cache line fills and the addition of a 96-Kbyte on-chip second-level cache. Since the T3D and T3E both emphasized bandwidth improvement over latency reduction, no board-level caches were provided in either system.

Global Memory

At the time there was considerable concern about memory management and memory use in MPP systems, and Cray focused considerable attention on issues such as memory mapping, translation look-aside buffer (TLB) misses, communication performance between PEs and I/O. The features that are described here provided a set of tools for both system and applications software that made the new environment much more easily used than expected, and in some ways better than on more conventional systems.

The T3E followed in the design footsteps of the T3D and implemented a distributed memory architecture, but with a shared global address space (GAS) that allowed fine-grain communications between processors without hardware-imposed cache coherency for remote memory. Conventional distributed memory computers have isolated local memories in each PE that can only be directly referenced by the local processor(s), with data communications between PEs accomplished using an I/O-like mechanism. These designs incur relatively large transfer overhead and start-up penalties that dramatically reduce both efficiency and performance on small or nonuniform access patterns. For example, the IBM SP-2 MPI ping-pong latency was greater than 100 μ s, compared to about 1 μ s for the T3E. The T3D and T3E were designed to support direct access of other PEs' memories using loads and stores (or intermediate "puts" and "gets"), enabling higher payload-to-overhead ratios on the sparse communications patterns that are important to many scientific algorithms.

The EV5 (aka, 21164), the second-generation “Alpha” 64-bit architecture processor from Digital Equipment Corporation, was the world’s most powerful microprocessor in its day but, like all microprocessors intended for desktops and conventional servers, its memory interface was designed to operate in an environment where the memory was physically close (on the same printed circuit board), small (a few Gbytes at most), and accessed using address patterns that made profitable use of large caches as a latency-avoidance mechanism. Unfortunately, none of these characteristics are present in large-scale distributed global memory architectures like the T3E, so a major architectural challenge was to provide external extensions to the address space, the memory management, and the latency-hiding capability. A maximum scale T3E might have over 4 Tbytes of addressable memory, beyond the addressable range of the EV5; conventional memory management translation look-aside buffers (TLBs) could only map a very small fraction of the T3E global address space, making significant performance delays due to TLB misses on every memory reference a near-certainty; and, latency hiding was required because the memory of a remote PE might be thousands of clock periods away, thus stalling a processor that was waiting on a data load.

The Global Segments (Global Memory Segments, GSEGs) capability was provided to help with remote memory access. The operating system was able to convert a user process I/O buffer address to a GSEG and pass it through the service PEs to the GigaRing device processors directly. The GigaRing devices were then able to use the address and do a zero-copy RDMA to or from the user’s buffer directly to a device. This was a mechanism that was as simple and high performance as the previous I/O subsystems used by Cray vector mainframes.

Although the T3E team had to freeze the design very early in the life of the T3D, much was learned from the predecessor system and the accumulated experience of compiler, library, and application developers as they struggled to generate efficient and scalable code for a distributed memory architecture. As a first-generation design, the T3D had provided programmers a variety of mechanisms to extend the addressing and memory interface limitations of the Alpha microprocessor and enable a processor to read or write data from a remote PE’s memory. Address extension for direct loads and

stores to remote memory used an external address segment register set called the annex. Latency hiding was accomplished by using either a prefetch queue (PFQ) (a set of first-in, first-out registers that could be the targets of “special” loads), or an external block transfer engine (BLT) that could move memory autonomously, but imposed significant overhead penalties. T3D memory management also required that all PEs participating in a computation have the same local mapping of memory, causing a page miss in one processor to require identical fix-ups in all PEs. While helpful, these mechanisms could be confusing and poor choices were punished by poor performance. Overall, achieving high efficiency was difficult unless the programmer deeply understood and carefully planned around the performance quirks.

E-Registers

The T3E architecture simultaneously simplified the way global shared memory was supported and greatly increased performance through the introduction of a large set of registers external to the microprocessor, called E-registers. This set of 512 user-level registers plus 128 system registers was mapped to a section of the EV5’s memory space normally reserved for memory-mapped I/O devices. Data could be moved efficiently between E-registers and processor-internal registers using standard loads and stores. The EV5 automatically merged sequential loads or stores to I/O space into four 64-bit word transfers that maximized the external chip data bandwidth.

A meta-instruction set controlling E-register operations was defined that could be issued by the processor by writing to additional special memory-mapped addresses. The particular address used represented the appropriate op-code, and instruction arguments could be passed in the data.

Possible E-register operations included “gets,” which would accomplish a data load from local or remote memory into an E-register, and “puts,” which would store data from an E-register into memory. To fetch data from a remote PE’s memory, the programmer would issue a “get” by storing the desired virtual address in the encoded memory address. The destination E-register would be implicitly specified in the op-code. Circuitry external to the EV5 then translated the global virtual address to determine the physical PE containing the

desired data and forwarded the request over the network. The returning data would land in the reserved E-register, to be subsequently loaded by the processor from its memory-mapped location. As many “gets” could be outstanding as there were available E-registers to act as destinations for the returning data, thus providing an efficiently pipelined latency-hiding mechanism. E-registers also supported advanced memory-based synchronization primitives including atomic memory operations (such as fetch and increment, compare and swap, and masked swap) and user-space messaging queues.

Using E-registers enabled the T3E to sustain network-saturating data transfer speeds even on global memory reference patterns with large non-sequential address “strides.”

Memory Management

The T3E implemented a unique global memory management and virtual address translation scheme that supported implicit data distributions provided by languages like Fortran-D and Cray’s CRAFT, as well as allowing independent and different mappings of each PE’s local memory, even as it is shared and viewed as globally contiguous. This was accomplished by using a hardware “centrifuge,” which could separate PE-designating bits and PE-oriented address bits from the virtual address, and remote translation of the virtual address to the physical address within the remote PE’s memory. The remote translation mechanism also implemented an “atomic memory mover” that allowed relocation of memory pages even while the page was being accessed, preventing common memory management tasks from disrupting computation.

Local Memory

The T3E provided no board-level cache and the EV5 was only permitted to cache local memory. Global memory references changing data in local memory were kept coherent by a back-map that automatically invalidated any cached copy on the processor. Instead of board-level cache, the T3E provided stream buffers that detected and pre-fetched sequential streams of data (as are common in bandwidth-intensive scientific applications), resulting in sustainable streamed data rates over twice

those achieved in server designs using the same processor but with conventional board-level caches (470 Mbytes/s compared to 190 Mbytes/s).

Synchronization

E-registers enabled advanced memory-based synchronization primitives including atomic memory operations (such as fetch and increment, compare and swap, and masked swap) and user-space messaging queues. Synchronization was further enhanced using a virtual barrier network that allowed arbitrary membership, and had the ability to send messages over embedded virtual spanning trees in the data interconnect network.

Network

Perhaps the most remarkable and visible attribute of the T3E was its 3D torus interconnect network. In this scheme PEs are physically connected to their nearest neighbors in three dimensions and the faces of the resulting logical cube are “wrapped around.” The router in each PE consisted of a fully adaptive router switch steering traffic among seven pairs of high-speed (600 Mbytes/s) channels: carrying data simultaneously in both directions of three dimensions, plus a connection to the PE. Traffic was routed according to predetermined routing tables in direction order: +x, followed by +y, followed by +z, followed by -x, -y, and -z. Five virtual channels were utilized: Four to prevent deadlock and one to enable adaptive routing. The routing algorithm was designed to send each packet along the shortest possible route, but adaptive routing allowed packets traveling in the network to route around broken or congested paths by selecting an alternative route. The T3E’s “minimal adaptive routing” strategy allowed packets to change the order in which they traversed the various x, y, and z hops, but the actual path length was not changed. One advantage of a 3D torus interconnect topology is the ability to route a packet “the other way” along any dimension and, in some failure cases, “long way around” routing was needed to route around faults, but this was accomplished by changing the routing tables.

The T3E network and communications infrastructure was optimized for pipelined delivery of fine-grain messages. Packets traveling through the network could have payloads as small as one 64-bit word, making the



T3E very efficient at random communications supporting sparse matrix algorithms.

The router internal clock ran at 75 MHz, however data was transmitted on the communications link at five times the internal frequency (375 MHz). The links were 14 bits wide, for a raw bandwidth of 656 Mbytes/s. Peak data payload was around 500 Mbytes/s after protocol overheads. The network of the T3E provided a bisection bandwidth of over 150 GBytes/s at 512 PEs and nearly flat global bandwidth of over 600 Mbytes/PE over scale to that size system.

Technology

The T3E was implemented using a combination of DEC Alpha EV5 processors, commodity DRAM memory chips, and Cray-designed CMOS application-specific integrated circuits (ASICs) blended in a custom architecture using the exotic packaging, power, and cooling technologies that were a hallmark of Cray supercomputers to achieve high-performance in a dense form factor.

The custom ASICs were sourced from LSI Logic and used a 0.8 micron process and three metal layers to provide a maximum of approximately one million gates per chip.

A single large printed circuit board (PCB) with more than 20 layers formed the motherboard for a T3E module. Each PE consisted of a processor chip, a C-chip that connected the processor to both the local memory and the interconnect network router, and a router or R-chip that completed the implementation of a single PE in the T3E's 3D torus network. Four PEs were laid out on each motherboard, and motherboards were attached to a "cold plate" for cooling. In the LC configuration, the cold plate was an aluminum heat exchanger through which a chilled inert fluorocarbon liquid (3M's "Fluorinert") was circulated. The cold plate had a series of machined "plateaus" on the surface that protruded through holes in the PCB beneath each chip, thus making direct contact with the chip. Heat was conducted through the plateau into the cold plate and removed through an external chilled water heat exchanger.

Fluorinert, though quite expensive, was used for the closed-loop in-chassis cooling cycle because it has the property of being electrically nonconductive and noncorrosive, so no damage to the densely packed electronics would occur in the event of a leak.

The LC cold plate was approximately a half-inch thick and was configured as a sandwich between two motherboards supporting a total of 8 PEs per module. AC cold plates supported a motherboard on one side and were machined with conventional air-cooling fins on the other, so they only supported 4 PEs operating at a somewhat elevated temperature, compared to the LC design.

Standard-package DRAM memory chips were mounted separately from the motherboard on four daughter cards per PE. Each daughter card also held an M-chip, which drove a bank of memory and interfaced between the DRAM and the C-chip. Pairs of daughter cards were sandwiched on two thin aluminum plates, which were then plugged as mezzanine assemblies over each PE. The memory aluminum cool plates contacted cold plate plateaus, providing a thermal conduction path for the heat of the DRAM to the cooling fluid or air, depending on the LC or AC configuration. No cooling fans were used in the LC design.

The 8-PE LC module, including memory mezzanine memory cards, was approximately 14 by 22 in. and a mere 2 in. thick. Connectors for the router channel wiring cables ran along both long edges of each motherboard, and used a unique zero-insertion-force connector design that required insertion of a tapered metal wedge, after seating the module in the chassis, to slide individual pin shuttles into contact with the sockets on the boards. Each LC module had a blue and red hose (hot/cold) with quick-disconnect inlet and outlet connectors to plug into Fluorinert manifolds mounted on each side of the rack.

The AC chassis could support up to 16 four-PE modules per chassis, and two chassis could be connected together to create a max-scale 128 user PE system. The LC system had considerably higher density, with 34 module slots to accommodate up to 256 user PEs per chassis. Power supplies, pumps, and Fluorinert/chilled water heat exchangers occupied perhaps two thirds of the chassis, with a single stack of modules in the front.

Flat cables, one per router channel, were strung between the modules in the chassis (the design of the torus network was "folded" so no cable ever had to cross over to the opposite module side) on the sides of the chassis. In larger configurations (greater than 256 PEs), multiple chassis would be arrayed in an alternating line, with every-other chassis facing the opposite



direction and overlapping by the depth of the module stack, “cheek to cheek.” This allowed torus network connections between chassis with the shortest possible cable connections as well as enabling side access to the pumps, power supplies, and heat exchangers in the rest of the chassis. From overhead, a large T3E looked like a very large zipper.

With no fans, the sound level of a full-scale LC T3E was considerably less than the AC version, or any contemporary conventional server racks.

Operating System Software

The development of a distributed operating system for the T3E was a significant effort for a small company like Cray. The transition from UNICOS on vector systems to UNICOS/mk on massively parallel systems was a major effort for the Cray software group. In order to meet the schedule, average of 100 people worked on the system for more than 4 years, while a skeleton crew of 15 senior people was assigned to manage and support UNICOS on Cray’s entire customer base. This was an extremely high-risk approach. In effect, the Software Group was betting on the extremely high reliability of the installed UNICOS systems, which proved to be a sound decision.

The competing requirements of supporting a new hardware architecture while also carrying forward an existing customer base presented significant challenges. Some of the new ideas for the Cray Parallel Programming Environment were very different from the interfaces and functionality that had been in use on vector mainframes. All of this contributed to the uncertainty experienced by the software system designers as they debated how to prepare for a new machine that was expected to run a whole new set of applications.

Historical Perspective

By the early 1990s, there was an ongoing sea change in supercomputing system software. The first wave, in the mid-1980s, had been the transition from vendor-proprietary operating systems to UNIX. Despite early skepticism, the Unix operating system and environment had successfully replaced the traditional operating systems of the mainstream supercomputer vendors. Furthermore, as supercomputers moved from research labs to production engineering groups, a new expectation of higher reliability was sweeping through the industry.

At Cray, the view had always been that the very highest performance systems, pushing the limits of technology, would never be as reliable as more common computers. Cray was designing at the leading edge of technology, and reliability was inevitably compromised in favor of higher performance, in the same way that specialist high-performance cars are often less reliable than family automobiles. Traditionally, Cray customers had an “early-adopter” culture whereby they were willing to accept a degree of risk in return for the earliest possible access to new, more powerful, systems. However, their users were now raising the bar. The new expectation was that systems had to be for what had previously seemed like long periods of time – now measured in months rather than weeks. This trend had been in progress for several years, but in the 1990s the market requirement was rising sharply. Hardware was becoming more reliable at the component level, but massively parallel systems contained many more components, and the pressure was on software to find methods that would make the overall system more reliable.

The advent of UNIX had brought a change to the way programmers thought about systems. The use of a high-level implementation language (C) and standardized interfaces was a very different approach from the assembler-programming techniques that most companies and laboratories had used in the 1970s. However, the actual evolution of UNIX was surprising to its originators: Rather than collaborate on an industry standard, each vendor produced its own flavor of UNIX. All the companies that adopted UNIX were under pressure to differentiate their products, and porting features from a proprietary OS that their customers had come to depend on was one way to differentiate. But this approach led to divergence of UNIX implementations, and at this point, in the early 1990s, there was no Linux with its common kernel. There were some industry standards but they did not cover as many interfaces as were in common use, and they did not cover the machine-dependent variations.

However, in many cases these vendor-specific additions to UNIX constituted useful improvements. For example, the original UNIX did not have the performance-oriented features needed for supercomputers, such as a fast file system or a tunable process scheduler. UNIX also lacked a number of features, such as job accounting, that were necessary for managing

large shared computational resources. Operating system developers were quick to resolve these problems by adding features to the kernel. Even though the kernel was not really the place where much of this code belonged, expediency and the pressure to deliver performance and resiliency were the rationale for making this entire code kernel based.

The consequence of all this feature addition was to take a lightweight kernel, roughly 20,000 lines of code in AT&T System V circa 1984, and create multimillion line kernels by the early 1990s.

The Evolution of Microkernels

At Cray, by the early 1990s, the UNICOS kernel had grown to more than 500,000 lines of code. This was far too much for even the best developers to keep in their heads, and ongoing software reliability was a concern. So the development team began to look at microkernels with the goal of separating the kernel components and creating a set of “firewalled” functions with defined interfaces. Microkernels were a new idea at that time. The Defense Advanced Projects Research Agency (DARPA) was actively supporting a microkernel called MACH, developed at Carnegie Mellon University, but there was another microkernel, Chorus, that had originated at the French national research institute, INRIA, and had been adopted by AT&T. There was much discussion at Cray about the relative merits of the two microkernels and much speculation about the potential performance loss that was expected from using either of them. The performance loss was certain because this architecture required multiple additional context switches between user, microkernel, and operating system services. The question was whether the performance loss would be an acceptable price to pay for the benefits.

When the T3D project started at Cray, the Software Group was reorganized into multiple OS development teams, based on the T3D, the T3E and future Vector Systems, each with its own views of the evolving software world. The T3D developers chose to use a modified version of the MACH microkernel for the back-end MPP, and were successful in producing a viable MPP. The Vector team elected to build a prototype version of UNICOS (Cray's proprietary UNIX operating system) on top of the Chorus microkernel. Chorus did not require virtual memory (unsupported by Cray vector system)

as MACH did, so it was possible to port Chorus and UNICOS to a Cray vector system. The experiences of the T3D team with the difficulties of MACH and the decision to make T3E self-hosted made Chorus the obvious choice. The final problem with MACH that caused Cray to abandon it as a potential microkernel for self-hosted systems was a set of fundamental security issues that were also being experienced by the developers of the Open Software Foundation's OSF/AD, which was also based on MACH.

As a result of its vector and T3D experience, Cray then began to look at a combination of Chorus and UNICOS as not just a potential OS for future MPP systems, but also as a next-generation implementation of UNICOS on vector systems. The big difference between the T3D, which comprised an MPP running MACH front-ended by a vector machine running UNICOS, and the T3E was that the T3E MPP would run as a complete single system without front-end support. The T3E would be “self-hosted”; simply running a microkernel on all the MPP PEs would not suffice. In addition, a primary objective of the T3E software design was to present a single system image, defined as a single Process Identifier space (pid space) and single file-name space, to the user, and to achieve this goal the operating system would have to be distributed across multiple MPP PEs. Thus the microkernel was conceived to be the low-level PE OS responsible for providing communication between all the PEs. But, the big departure was to take the rich set of UNICOS services and distribute them in “stacks” across MPP PEs depending on the software-defined PE type and the need for system services. These ideas were speculative at the time. No previous high-performance computing (HPC) company had delivered a distributed system, and there were a number of valid concerns starting with the potential performance of a single HPC application distributed across many PEs and the performance impact of PE-to-PE communication on a mixed scientific workload.

The Cray software group as a whole was divided almost into thirds in their views of the wisdom of this direction. A little more than a third of them wanted to go forward with the microkernel-based UNICOS plan; a little less than a third were convinced that this technological approach would fail; and about a third thought the whole MPP concept was doomed anyway.

The people at DARPA who had been funding MPPs and the development of MACH were also extremely unhappy that Cray had decided not to continue with MACH.

Distributed Software Architecture

There were serious technical issues in dividing UNICOS into servers (serverization). The organization of the Unix system was built around the concept of files, and separating the process management services from the file system, while also creating a simple interface, proved difficult. The idea of firewalling the microkernel from the servers was dropped because of performance issues in the context switching. Without the firewall the concept of being able to reboot individual services instead of the whole OS (on a node) was not a reality. This made some of the resiliency hopes less possible. Still, the idea of rebooting individual PEs, instead of the whole system, was possible and was an important step toward a resilient HPC system.

Besides resiliency, there were other compelling reasons to go forward with a distributed UNICOS operating system on the T3E. First, the back-end T3D-style of MPP had been difficult for programmers to run and debug applications. A back-end system, by definition, is always more opaque than a system with which the user interacts directly when testing a code, and the limitations of the T3D in this respect were a concern for Cray customers. Second, UNICOS had evolved on vector systems to become a very stable, high-performing, full-featured HPC Unix system. The customers liked it and the idea of a UNICOS version running on the T3E was well received. In addition, the performance of a microkernel-based UNICOS running on a single Cray vector CPU had proven to be much closer to the performance of native UNICOS than most people had thought possible. Finally, there were not a lot of alternatives. This technological direction was undoubtedly risky, but the whole T3E program was risky. So the company decided to go forward, and carefully manage the risk by establishing well-defined milestones and deliverables. As is often the case, there was a protracted debate over a less critical issue, the OS name, but UNICOS/mk stuck. It was naming similar to the Chorus-enabled version of Unix from AT&T and it preserved the UNICOS brand.

Serverization

The following diagram shows the structure of the OS stack and all the possible servers ([Fig. 1](#)).

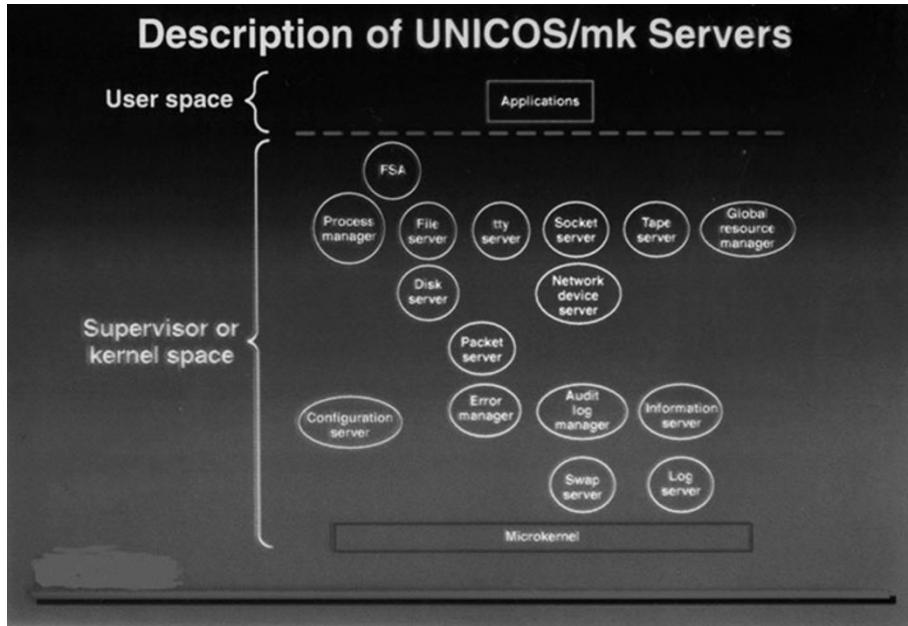
This simple example of server stacks shows compute PEs with minimal services and OS PEs with complete stacks of services ([Fig. 2](#)).

A useful way of demonstrating the contents of both service and applications PEs was to use the Cray 3D viewer. Here are three examples of different views of two support PEs and a command PE in the torus, and the processes running on those PE's ([Figs. 3–5](#)).

The system PEs did not allow user processes to run. Users logging into the system were automatically directed to a command PE where the user interface appeared to be standard UNICOS, but the underlying system was actually distributing the user's processes across multiple command PEs as needed. Compute PEs were only able to execute applications that had been submitted to the resource manager/schedulers.

The distribution of system services meant that some set of PEs became "system PEs." This was hotly debated because the market expected that a T3E with "N" PE's would allow a single application to run on "N" PEs simultaneously: Cray customers had become accustomed to the idea that a user application could access all the user-programmable hardware resources of the machine in a single execution. Thus system PEs could not be included in the advertised processor count and needed to be "extras." The fact that the system PEs could not be used by applications was irritating to some users, who saw them as pure overhead, and the cost of these extra PEs also had to be considered.

However, reality was that operating systems services are essential to the running of any application, but the cost of such services on conventional mainframes had traditionally been accrued as system overhead. Also, on mainframes, this overhead was a variable number that depended on application characteristics, whereas the T3E serverization approach essentially allocated a fixed proportion of total system computational resource to system services. The resolution was to peg the number of system PEs, as a percentage of total PE's, to the overhead of a non-distributed OS on a shared-memory vector system. As a result Cray decided, based on the measured performance of the installed base of CRAY C90 vector systems, to use no more than one in 16 T3E PEs



CRAY T3E. Fig. 1 The Unicos/mk Software Stack – The microkernel and all the OS servers

(6.25%) for dedicated system purposes, and to include these PE's as "free" PEs that were not included in the MPP processor count. Thus a 1280 PE T3E would actually comprise 1280 computational PE's and up to 80 PE's dedicated to OS services, for a maximum of 1360 PE's.

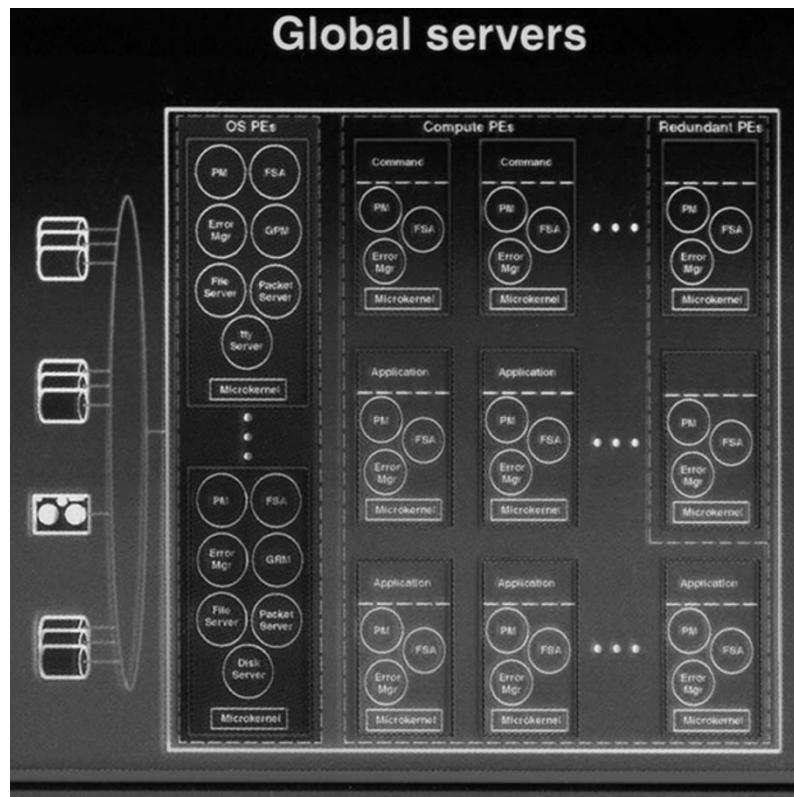
All system services ran on the service PEs, including PEs used by users logging into the system on "login" PEs. A few extra PEs were also kept available as replacements for failed compute PEs, which might be lost between planned maintenance outages. The operator could repurpose the replacements and, later, a PE reboot feature was added that allowed operators to restart failed compute PEs. The T3E cabinet to support this system organization held a maximum of 272 PEs: 256 user PEs; 16 system and/or redundant PEs.

As part of scaling, and also of distribution, the stack of OS services on an OS PE was differentiated. The most obvious difference was "Command" PEs that were used for user logins. But other OS services were separated or replicated onto OS PEs. Depending on the system size and configuration, there could be different numbers of these specialized OS PEs. OS PEs could be allocated to services based on the utilization of the service within the system. I/O device support was often configured on I/O channel-connected PEs with no other services. This

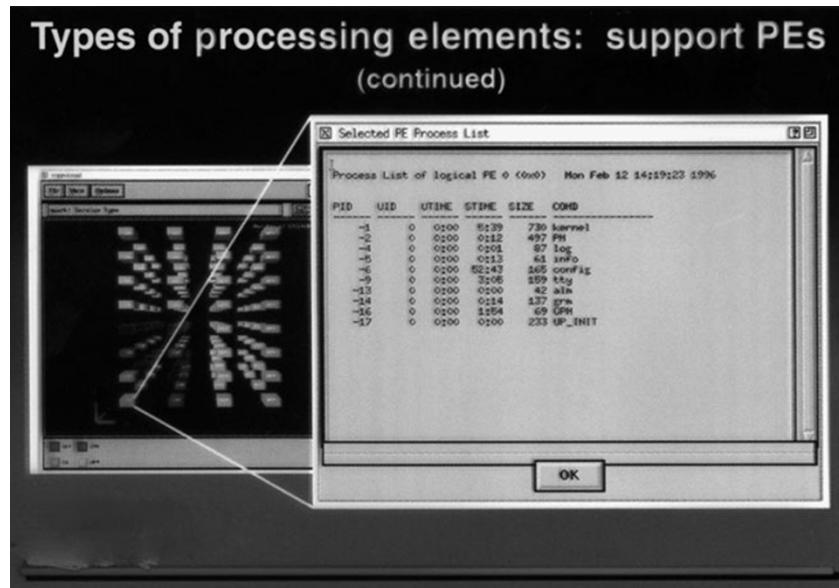
reduced the number of different activity types on the OS PEs, and thereby reduced the number of interactions between different services, which had been the source of many failures on vector systems. Focusing the tasks allocated to any given PE and simplifying the services made the system inherently more resilient and easier to debug.

The use of OS PEs also allowed for the development of Global and Local service divisions. This was used extensively in process management. The local process manager (PM) could respond on its own to a `getpid()`, but would forward a `kill()` to the Global Process Manager (GPM). This global/local split was popular because it improved local performance and reduced the amount of communication.

Another example of local versus global servers was the development of the File System Assistant (FSA). File system requests are dominated by read and write requests. The extra requests to OS servers from compute PEs and the copying of data across multiple PEs was something to avoid. The FSA communicated the user's `open()` request to the file servers. In the file server's reply to the FSA open request there was a disk server address and a set of disk blocks for the file. Depending on the request type there could be multiple disk servers and multiple block allocations. Read and write requests

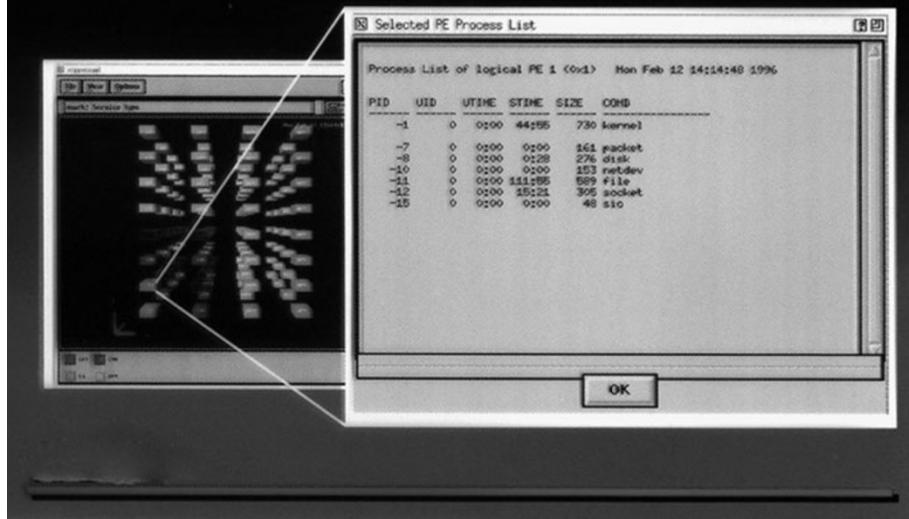


CRAY T3E. Fig. 2 The Unicos/mk OS deployed on a system showing the bulk of the OS servers on OS PEs and a light weight set of servers on the Compute PEs



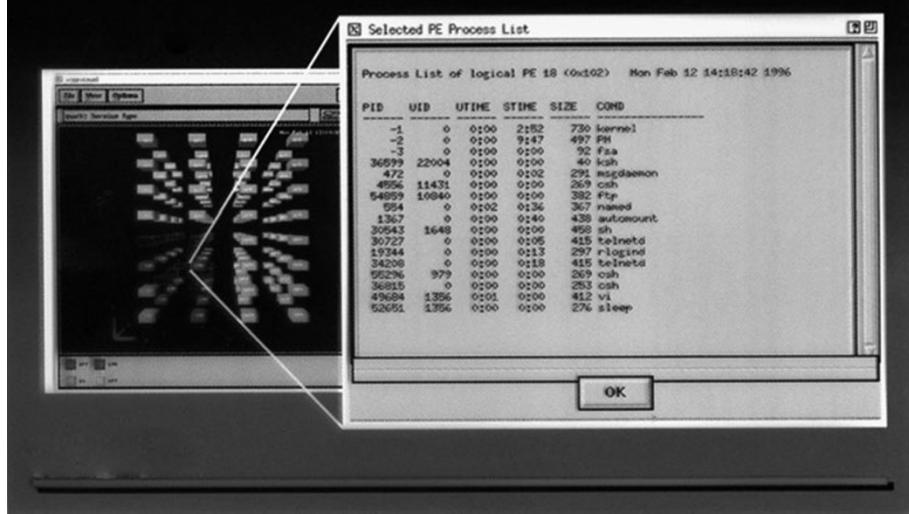
CRAY T3E. Fig. 3 A snapshot of a system in the left-hand window and the actual OS servers in the right-hand window. This shows the main OS servers on one of the OS PEs

Types of processing elements: support PEs (continued)



CRAY T3E. Fig. 4 A snapshot of a system in the left-hand window and the actual OS servers in the right-hand window. This shows the file system, disk, and other I/O OS servers on one of the OS PEs

Types of processing elements: command PEs (continued)



CRAY T3E. Fig. 5 A snapshot of a system in the left-hand window and the actual OS servers and user processes in the right-hand window. This shows the OS servers and user processes on one of the Command PEs

from the user could now bypass the file server and be sent from the FSA directly to the disk servers. The file server was responsible for managing and guaranteeing coherency.

There were a number of key optimizations of the T3E serverized system. The distribution of global process management and file I/O was primarily via system-call forwarding from either compute or service PEs to the process-management and file-server PEs respectively. The code for these calls was optimized to do as few data transfers as possible. The goal was to keep the PE interconnect as free of service requests as possible. The term “piggybacking” was used to describe these optimizations. Substantial changes were made to the IPC mechanism to improve performance including the creation of interrupt threads to reduce the latency of thread switches for forwarded requests. Another optimization that was important in process management was the separation of `fork` from `exec`. In small Unix systems, often a `fork/exec` would be the same program replicated, like a shell. In HPC, the `exec` on a compute PE was guaranteed to be an application code not the shell that forked it. This optimization eliminated the unnecessary copying of process images across the interconnect network. High-performance application workloads were a performance benefit for the operating system too. These applications were invariably compute-bound and made few system calls, which helped with reducing overhead and non-application communications.

The amount of available memory on compute PEs was a critical resource. The OS was allowed to use all the memory on service PEs, but had to be using the absolute minimum on compute PEs. The target was set initially at 10% of a 128MB PE and this turned out to be a good target. Later, other MPP operating systems were found to be using 90% or more of compute-PE memory and this became a competitive marketing advantage for Cray.

Schedulers

Scheduling had been an important part of HPC operating systems on vector machines, but the MPP required a new scheduler to manage the compute PEs. This scheduler became important in providing overall system utilization. There was a performance advantage in locating processes on PEs that were “close” to the other PEs running processes for the same application. The scheduling

needed to be fast, accurate, and ensure that no compute PEs were loaded with more or less than a single runnable process. A simple algorithm was developed to make a serial array of the three dimensions of the compute PEs in the T3E torus. This made allocation and management much simpler and defied earlier predictions of poor placement. UNICOS/mk supported space-sharing from the outset; that is, the ability to split the system into multiple disjoint partitions and run a different job in each one. An interesting additional feature was coarse-granularity time-sharing with gang-scheduling within a set of PEs. This was the ability to have a partition load processes from multiple different applications, and then run simultaneously all the processes from a specific application across the whole partition. This would idle the other applications that happened to share PEs with an active application and, where there was a discontinuity in the number of PEs a process used, even more PEs might be idled. However, combined with scheduling constraints at the job level, the process/memory scheduler was tuned well enough that it was able to overlay applications across PEs and achieve a remarkably good overall utilization of the compute PEs.

Checkpoint/Restart

The T3E was one of the last systems to have a completely usable checkpoint/restart capability. At the outset of the program, many developers considered this feature to be impossible to implement successfully. The number of processes, the size of the checkpoint image, and the difficulty of coordinating the saving and restoring of process state all presented daunting challenges. However, the checkpoint feature was an important requirement of several large customers and Cray decided to invest in the best implementation possible. This project took a long time to complete and was hard to debug completely, but utilization charts subsequently produced by the National Energy Research Scientific Computer Center (NERSC), which reported system utilization greater than 99%, vindicated the effort put into this feature.

The interesting technical lesson learned from this project is that a process checkpoint can be viewed as a primitive that, once in place, allows the development of other features such as process migration. The migration feature had never been considered in the original design process, but the checkpoint capability had essentially created the potential for the system to package an

active process and restart it at a different location in the MPP array, which is the definition of process migration. The availability of process migration is of significant benefit to a datacenter operation that wants to run an interactive program development environment during the daytime hours and a batch processing environment overnight. Many of the big production batch jobs may need to run for several days, and thanks to process migration they can be automatically suspended, saved, and rescheduled in a flexible manner. It also provides an effective mechanism for unscheduled high-priority jobs to preempt an existing workload on some or all of the compute PEs. This workload scheduling capability was fundamental to the outstanding results achieved by NERSC.

System Scaling

Despite concerns that the operating system would not be ready in time to support the early hardware deliveries, the OS software was actually delivered more or less as planned. Because there was market demand for T3E from a number of T3D users and a constrained initial production capacity, Cray decided to allocate smaller numbers of PEs to each of several early adopters to be followed by upgrade deliveries, rather than allocate the whole initial production to a single large customer. In practice, as the installed systems transitioned from small to larger scale, the software team was able to deal more easily with the inevitable scaling issues than if they had to deal with a single full-size machine from day 1.

As a general rule of thumb, it seemed that every factor-of-2 increment in system size unearthed new scaling problems. The distribution of process management and process running between service and compute PEs scaled pretty easily. There was not a lot of stress on the process management PEs except when they were subjected to specific stress tests. The compute PEs were sufficiently balanced that most applications could run without generating much OS overhead and communication to the process services, even though almost all system calls were forwarded. So, distribution worked without a lot of hierarchical scaling, which was a relief. I/O and networking had been predicted to scale quickly from one file system PE to many, but this happened more slowly than expected. There were probably a lot of reasons for this, but in practice it was not a major

issue as application I/O usage definitely lagged behind the system's ability to scale.

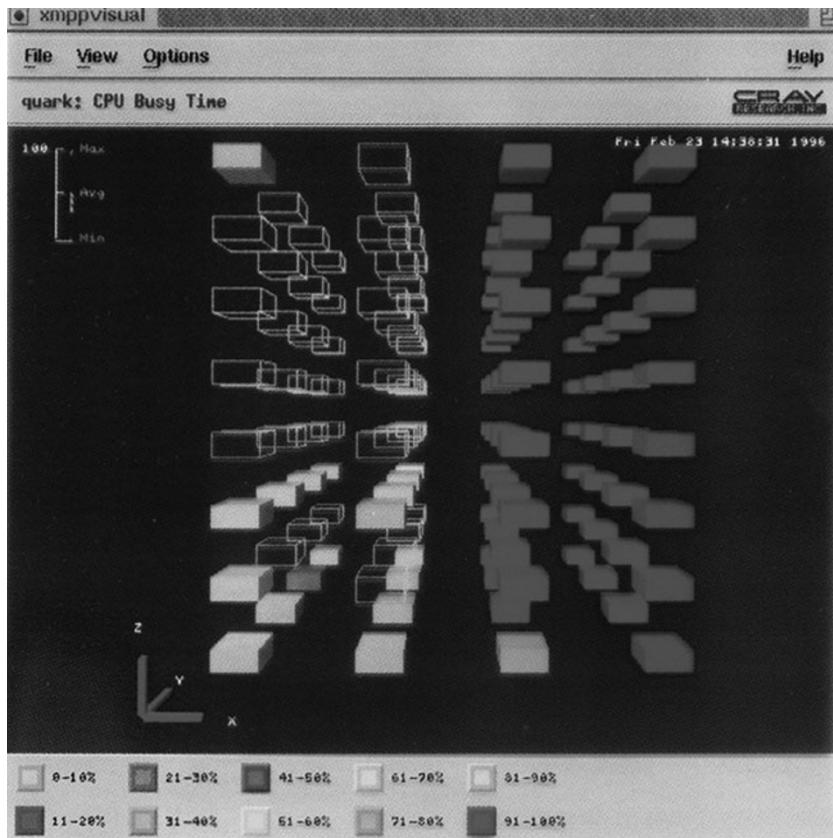
Scaling I/O is a good example of the work done to scale the T3E system and user software. The system could improve scaling by splitting the software stack onto different PEs. This was "vertical scaling." As discussed previously, this was separating drivers onto channel-attached PEs, and file and network services onto other OS PEs. Each group of drivers or file server PEs could be scaled independently, horizontally. I/O scaling was also improved by shortcircuiting the global communication between servers whenever possible. Finally, in another effort to group requests, users were given new interfaces that allowed lists of I/O requests across multiple destination PEs.

Overall, the development team was somewhat surprised at how easily the system worked. Features that they were concerned about mostly seemed to scale without problems. The hardware was actually helping the OS in that the interconnect network was so fast and able to communicate so many small messages quickly that running the inter-service communication mechanism was less of a problem than had been expected. The user interface looked like a conventional shared-memory vector UNICOS system and was easy to demonstrate to customers. Explaining the system organization had been difficult because the concepts were new, but the development of a graphical 3D display of the PEs and PE status finally gave a view of the system to users that could be readily understood. Internally, the microkernel architecture had transformed much of the original UNICOS kernel into special processes or threads, thus allowing the developers to use standard debuggers on many of the kernel processes. It had made productive programmers out of the majority of the Operating Systems Group.

Here are a few examples of the 3D viewer measuring processor utilization on all the PEs in a system. Note that not all the PEs are running applications in these examples. This allowed us to demonstrate activity in the system and how it changed as applications started and stopped (Figs. 6–8).

OS Jitter

One of the clear lessons from the T3D related to the necessity of coping with what later became known as OS jitter; that is, the effect that typical general-purpose



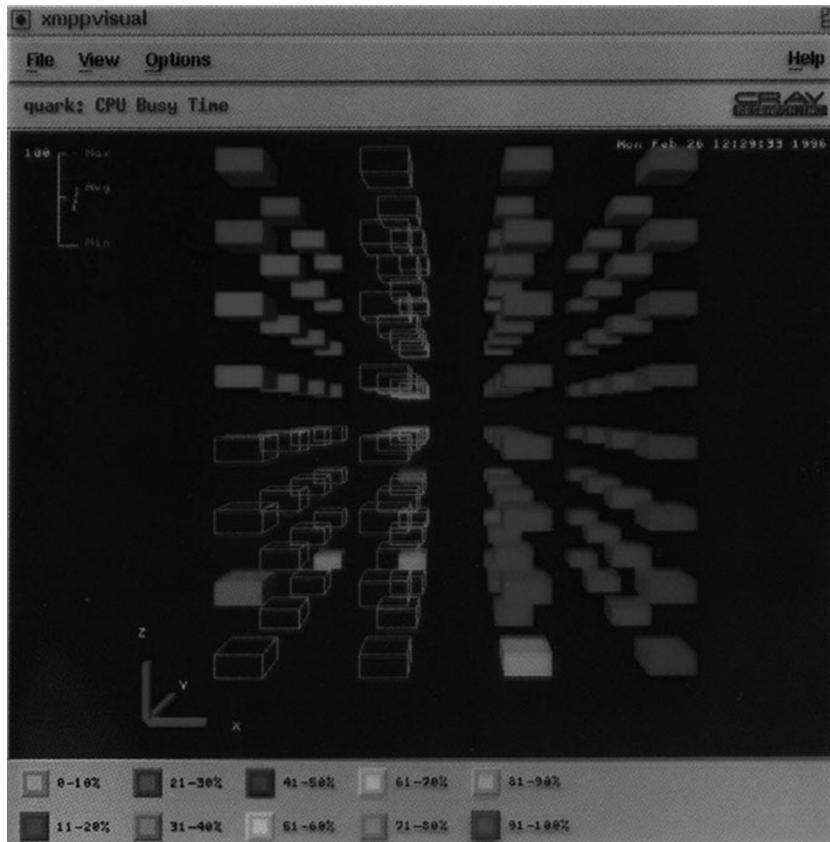
CRAY T3E. Fig. 6 A snapshot of a system showing processor activity on different PEs. This system is running several applications. The snapshot shows application placement and available compute PEs

OS services could have in a thousand-processor collective execution. Because the T3D and T3E were targeted at tightly coupled problems, the frequency of inter-processor synchronization in applications was high, often in the range of every 50 to a few hundred microseconds. At each of these synchronizations, if even one processor was delayed due to some other OS service, all the remaining processors would wait for the lagging processor to catch up. Even in the T3D's constrained back-end configuration and accompanying microkernel OS architecture the effect of timer interrupts (used for a variety of purposes, such as dead-man timeouts) that were unsynchronized across the whole system often cost a factor of 2 or more in performance.

The major changes to address OS jitter on the T3D were to drastically reduce the actual frequency of timer interrupts on the compute PEs, in effect

coarsening the response time, and to synchronize these coarse timer interrupts across all the PEs in the system. This was made possible by the global clock in the T3D/T3E, which is no longer typical of high-end parallel systems. These changes addressed the issue sufficiently for the T3D, but UNICOS/mk had a decidedly richer set of system services that were used by daemons, subsystems, etc., and further steps were necessary.

The T3E compute-PE OS stack was limited to the microkernel and a process manager. This reduction in local services ensured that the PE itself would be "quiet." This is more important than synchronization. Synchronizing ensures "lockstep," but reducing the services gives cycles to the user application and reduces the need for synchronization. The global clock provided synchronization, similar to the T3D. This kept time-related "housekeeping" in parallel, but communications



CRAY T3E. Fig. 7 A snapshot of a system showing multiple applications running and available compute PEs

between services, such as “heartbeat” checks had to be carefully examined and reduced as the system scale increased. There had always been an ethic at Cray related to ensuring the services did not reduce the time and space available for applications. This new environment was another lesson in monitoring system overheads and developing an awareness of their communications impact.

Programming Environment

Programming Models

Cray learned from the T3D that there were a variety of programming approaches for MPP systems, and no single approach was likely to satisfy everyone, although message-passing was emerging as the most common. Both the T3D and T3E provided fertile substrates for programming model development and exploration, as the hardware global address space (GAS) and fast

interconnect and synchronization meant the building blocks for a wide variety of approaches could be made to run fast.

Most of the T3D/T3E development team (both hardware and software) came from the legacy of Cray parallel-vector machines, which had a global address space and shared memory (though not hardware cache-coherence, as there were no caches, and the software managed the “cache” implemented by the vector registers). Many at Cray believed that the global address space provided a significant benefit in both ease of programming and performance, and wanted to find ways to expose that appropriately through the prevailing languages of the day, Fortran and C. This put Cray at odds with many other MPP pioneers, who had adopted the message-passing approach as being simple and less dependent on specific hardware, compiler, and run-time capabilities that might not be available on all systems.



CRAY T3E. Fig. 8 A snapshot of a system showing applications in different states of starting and completing

Message-Passing

Cray had implemented the Parallel Virtual Machine (PVM) interface on the T3D in 1992–1993, partly because of the need for a mechanism not only to communicate among the distributed processes on the T3D but also to communicate between the parallel-vector host system (a CRAY Y-MP or CRAY C90) and the distributed processes. Since then the message-passing community had coalesced and the Message-Passing Interface (MPI) standard was under development. While PVM was still supported for backward compatibility with T3D customers, most of the T3E focus for message-passing development was on MPI.

The E-register mechanism in the T3E hardware was a much better infrastructure for message-passing implementation than the block transfer engine (BLT) and prefetch queue (PFQ) of the T3D, and the resulting performance was very good. The MPI libraries could drive the hardware interconnect at 260 MB/s for bigger

transfers (100 KB or more), and the ping-pong latency was good (14 us), but not yet to the 1 us level achieved in later GAS machines.

SHMEM

SHMEM was a direct result of the fertile hardware communication and synchronization infrastructure provided by the T3D. One of Cray's MPP performance specialists realized that the T3D prefetch queue (PFQ) could be used to overlap communication with computation, and wrote a small prototype library that exposed this capability. He used it in a variety of codes and realized its advantages for fine-grained communication. When it came time to implement libraries for the T3E, the team realized that this interface, now known as SHMEM, was valuable and should be carried forward. The T3E's E-registers were a major expansion of the capabilities of the PFQ, enabling much more data to be in flight between remote processes and the

requesting process, and so SHMEM was consequently a much more potent performance enhancer. The T3E SHMEM implementation exploited the larger number of outstanding requests, each for 8 64-bit words of data (compared to a single 64-bit word for each PFQ request) to deliver the full bandwidth of the T3E interconnect (350 MB/s) for even small transfers of 1–4 KB and delivering unprecedented 1- μ s latency for single-64-bit-word requests. The much better performance of SHMEM reflected in part the close match with the underlying hardware and partly Cray's minimal experience with optimizing MPI protocols onto GAS hardware.

Language-Based Global-Address-Space Models

Researchers and developers had proposed and implemented a variety of language approaches to parallelism. OpenMP was becoming a standard by this time, targeted at cache-coherent systems of modest scalability. Vienna Fortran and Fortran D had both proposed and implemented early interfaces from Fortran to distributed memory systems, and Cray had implemented its own version, CRAFT, on the T3D. These languages had explored different approaches, but had in common the provision of global variables and a means of specifying their distribution across the PEs of the system and the use of those variables. Cray's experience with CRAFT on the T3D was mixed: A few developers had excellent success in both speed of development and performance, but the CRAFT implementation was incomplete in some respects (e.g., there were no collective operations) and not deeply optimized. In addition, its portability was limited to Cray systems, and many of the most avid user proponents of GAS, especially those whose workloads were characterized by the GUPS benchmark, had migrated to C as their base language of choice, not Fortran. As Cray was deciding what to implement for the T3E, the High-Performance Fortran (HPF) group was gaining critical mass, and it was clear that implementing a GAS Fortran language different from HPF made no sense. Unfortunately, interactions between the HPF team and the Cray software team were not the best, so HPF did not incorporate all the CRAFT lessons, notably the difficulty of optimizing even less-general distributions than were supported by HPF and the magnitude of effort required to provide a fast comprehensive run-time and libraries. For its part,

Cray was so constrained by its own financial difficulties that it did not implement HPF in its own compilers, instead relying on third-party compilers from The Portland Group and others. While HPF's progress took several years to unfold, ultimately it was unable to provide the performance that most MPP users expected, and consequently withered away.

However, Cray's mainstream compiler development was not the only source of language-based ideas for using the T3E. A T3E performance specialist, with help from the Fortran front-end group, integrated his ideas for a simple “PE-oriented” approach, first embodied in SHMEM, into Fortran in an interface first known as “F--”(F minus minus, a parody of C++) and later, at the request of for a less negative spin, formally named Co-Array Fortran (a contraction of covariant array, a mathematical term). The key innovation was to extend indexing operations with another set of square brackets that indicated the PE on which data resided; indexing without this extra information defaulted to the local PE. This work was eventually released in the late 1990s as part of the Cray Fortran90 compiler, though not fully documented and supported, and was later incorporated into the Fortran standard in 2005.

Similar ideas had been percolating in groups that focused on C more than Fortran. A team from the Institute for Defense Analyses Center for Communications and Computing had developed the ac compiler on the T3D, exploiting its PFQ similar to the way SHMEM did, though integrated with the C language. A team from Lawrence Livermore National Laboratory had implemented the Parallel C Preprocessor (PCP) first on the BBN TC2000 system and later the T3D, and a team at UC Berkeley had implemented the Split-C compiler. These languages attempted to extend C's close-to-the-hardware nature with minimal mechanisms supporting distributed memory. The ac compiler was ported to the T3E, but the bigger story was that these three efforts merged into the Unified Parallel C (UPC) effort, which later delivered compilers that work on a variety of parallel systems.

The fertility of the T3E infrastructure was proven by one last effort, little known at the time, a project at Lawrence Berkeley Laboratory/NERSC that resulted in the Parallel Problems Server being ported to the T3E. The tool later was commercialized by Interactive

Supercomputing as Star-P, extending the M language of MATLABTM to run on distributed memory clusters.

Tools

Compilers

The T3D/T3E Fortran and C compilers were hybrid, with the front-ends being those used on Cray's parallel-vector (PVP) systems and the back-end being a heavily modified version of the Compass compiler back-end. While the use of the Compass back-end quickly delivered the ability to generate decent code for the T3D/E PEs, based on the Alpha microprocessors, the shift to distributed memory was a more profound step for which neither the PVP nor Compass components were designed. This difficulty was largely side stepped because of the decision not to support any distributed memory programming language on the T3E, so the compiler developers were able to focus on single-core robustness and performance. The PVP compilers were legendary for their ability to analyze code deeply for parallelization via vectorization, though they had no technology for optimizing the use of cache (which the PVP systems did not have). Conversely, the Compass back-end was not designed to work with a front-end that could provide such robust information about the parallelism inherent in the input program. The rework of both segments of the compiler was possible, but resource constraints and the compiler group's need to split its focus between the T3E and PVP systems hampered the delivered performance of code on the T3E. Due in part to deficiencies in the use of cache, the T3E compilers never reached the performance levels of the compilers for the Alpha processor itself.

Libraries

The strong similarity between the UNICOS and UNICOS/mk operating systems at the system-call level drastically simplified the task of providing (single-processor) libraries for the T3E. Of course extensive parallel libraries needed to be delivered, including synchronization (e.g., barriers, eurekas) as well as the MPI, PVM, and SHMEM libraries. Parallel I/O was supported by independent, positionally nonoverlapping calls to the parallel I/O infrastructure and file system. The Fast File I/O (FFIO) library layer was also ported

from the PVP systems, providing a caching layer on top of the typical system libraries. Parallel scientific libraries were another strong focus, as the Cray scientific library, LibSci, developed for the PVP systems was redesigned to a great extent so as to run effectively on the distributed-memory T3E.

Debugger

The debugger for the T3E was Cray TotalView, which Cray had licensed from BBN for the T3D and ported to its UNICOS MAX operating system from the original platform on the BBN Butterfly and TC2000, and then ported to UNICOS/mk. It supported debugging capabilities that coped effectively with programs up to 128 cores. Beyond bridging the OS differences, the Cray development work focused on exposing the T3E hardware mechanisms and scaling TotalView's debugging capability to larger processor counts.

Profiler

Cray's MPP Apprentice profiler was built on ideas pioneered by ATExpert, a prior tool developed to optimize programs written with Cray's AutotaskingTM software, a precursor to OpenMP. The Apprentice dynamically instrumented code at the basic block level and could report timing results at the loop, condition, or other statement blocks with the time spent executing, synchronizing, and communicating reported separately. Also, the results could be split for an individual processor or summed for all the processors. Further, this information was provided in the context of the user's original source code, which was not uniformly true for parallel performance tools of that era. The instrumentation was done so as to be suitable for execution of very large programs, measured either by lines of code or length of execution, running on a thousand processors or more. The development team pioneered a number of innovations that enabled this advanced level of information.

Conclusions

The User Experience

While the T3D and T3E projects embodied many powerful technical innovations one can argue that the proper measure of their significance to users lies not simply in those innovations but, more importantly, in

the stunning improvement in the applicability of highly scalable computers to a host of important scientific and engineering problems, as evidenced by the number of successor systems extending the path pioneered by T3D and T3E.

Work on T3D began against the background of microprocessors having advanced beyond simple integer and text processors to support high-precision floating-point calculations at useful speeds. Although powerful individually, they could not compete with the purpose-built multiprocessor systems from Cray and others that were designed from the ground up for use on the most challenging scientific and engineering applications. Software, in the form of callable libraries, notably Parallel Virtual Machine, was available to support cooperative work on multiple processors using the message-passing paradigm. However, for the most challenging scientific applications, the hardware support essential for HPC-level performance did not exist. Well-integrated, high-bandwidth, low-latency inter-processor communications capability was simply not available on even the best of the microprocessors. Creating an appropriate interconnect was particularly difficult because the microprocessors had been built, and highly optimized, for working singly on a quite different set of applications. Furthermore, the design and development effort for those microprocessors dwarfed the resource typically available for custom HPC systems. The large commodity microprocessor market justified that effort, but modification of those processors specifically for HPC was not economically feasible.

There had been other efforts to build highly parallel computers from microprocessors notably the Intel Corporation iPSC/860 and Paragon systems that were solely message-passing and the Thinking Machines Corporation (TMC) CM-2 and CM-5, which were blends of SIMD and MIMD functionality with significant architectural innovation in global addressing, synchronization, and programming (CM Fortran). Each of these systems had proved essential points about the ability to decompose application codes for distributed memory and explored a variety of system mechanisms to enable the execution of scalable applications. Cray's MPP design team greatly benefited from detailed discussions with early customers of the Intel and TMC machines regarding their experience of using those systems, and

the mechanisms that would be necessary for effective scaling of the most challenging HPC applications.

In addition to the substantial investment made by Cray in developing the T3D and T3E systems, there was a correspondingly sizable user investment in the application reprogramming effort needed to take proper advantage of the new scaling features, especially distributed memory. The new mechanisms had been designed specifically to enable an increase in the number of compute PEs that could be allocated to a single application before reaching the “point of negative return,” where the increased communications and/or synchronization overhead causes application performance to decrease as the number of applied processors increases. In earlier, fully shared-memory systems those overheads were controlled by both an extremely costly shared-memory interface and a very low bound on the number of processors. Applications codes were successfully exploiting the largest of those shared memory systems, but to realize the full potential of the T3D and T3E required converting codes to the message-passing style of work-partitioning. The next step, after the codes had been converted, was to drive up their scalability by artfully improving the efficiency of the message-passing process and allowing it to overlap with computation. Both of these techniques had long been used to reduce the I/O overhead in HPC applications, but the message-passing problem was more demanding. In some cases, the computational algorithms themselves needed to be modified.

Moving from a commodity cluster to an HPC-capable MPP system required not only the creation of a powerful, scalable message-passing network but also the provision of an adequate I/O system. I/O capability is sometimes an afterthought, but it can often be the performance-limiting feature for HPC applications requiring the input and output of huge amounts of data in combination with the computational load. For the T3D, I/O was channeled through its host vector machine, already endowed with HPC-class I/O; but, being self-hosted, the T3E required its own I/O capability.

For the T3E to qualify as a supercomputer-class MPP it needed not just an extremely flexible and powerful hardware I/O capability, but also the hardware and software to make that capability a fully shared resource able to be focused on one or many jobs independently

of their location within the 3D torus interconnect structure. Segregating I/O handling gave a degree of predictability to the performance of application PEs; and the rare applications that heavily stressed the T3E's I/O capacity, for a given number of I/O PEs, could be further isolated by explicitly placing them in the torus close to the I/O PEs servicing their needs.

With the development of the T3D, Cray quickly realized that application development would be critical to the success of both the T3D and T3E. Consequently, the company fostered and explicitly supported the necessary user activity by instituting a Parallel Application Technology Partnership (PATP) program, which involved researchers and application developers associated with five major HPC centers around the world. Cray provided access to T3D hardware for participating organizations to work alongside Cray applications staff in effectively parallelizing a host of applications of both scientific and commercial interest. This joint effort required both programming expertise and the cooperation of the owners or authors of the codes that were to be modified. The PATP program helped ensure the success of the T3D and T3E programs by significantly contributing to the rapid development of a critical mass of applications.

A prominent early adopter of the CRAY T3D, and an effective member of the PATP program, was the Pittsburgh Supercomputing Center, whose scientists report that early achievements included:

- The first high-resolution model of the Gulf Stream, reproducing benchmark realistic features
- Prediction of the location and structure of severe storms 6 h in advance, providing economic benefit to airline companies
- Microsecond-length simulation of the villin headpiece sub-domain and implementation of improved, more accurate algorithms
- Real-time fMRI: coupling the T3E to an fMRI instrument, allowing observation and analysis of functional brain response in real time
- Elucidation of the mechanism for HIV Reverse Transcriptase
- Simulation of unsteady combustion using a commercial code (FLUENT)

For a typical application, initial conversion to message-passing produced a code that would improve in

performance almost linearly for low numbers of processors but quickly reach the point of negative return, where messaging overhead would begin to dominate over the decreasing computational load per processor. However, with effort and experience, the range of performance improvements that could be achieved by ever-increasing processor counts could be expanded. Typically, impressive results were obtained over a period of several months. In many cases, it was possible to bring to bear the total processing power and memory of a full T3D (often 128–512 PEs) on a single problem. Both the aggregate processing power and the aggregate memory were much larger than previously available on traditional vector supercomputers. Depending on the application, new scientific ground was broken because of the computational power, total memory size, or both.

The T3E Contribution

The history of computing, and especially supercomputing, is a continual co-evolution among hardware designers, software developers, and end-user scientists. The scientists constantly demand much higher performance to achieve the breakthroughs they see just beyond their grasp. The hardware designers work within their physical realities and devise architectures that better deliver the key performance metrics. System software developers simplify access to the hardware structures from the prevailing languages. This process is not linear though, since some new hardware approaches ultimately prove to be usable with current software technology and some, such as Seymour Cray's original CRAY-2, do not. The development of the CRAY T3E is therefore best viewed in the context of research and development done in the prior years by Cray Research and others.

Cray developed the T3D and T3E against a background of fully commoditized but weakly performing clusters. Superimposing Cray's engineering capabilities, and its understanding of the needs of high-performance technical computing, onto the prevailing cluster architectures enabled the company to pioneer the successful HPC transition to massive parallelism. The critical hardware engineering innovation was the low-latency, high-bandwidth interconnect, and its 3D torus topology. Adopting an expressive programming paradigm (message-passing), which was the result of industry-wide collaboration, helped overcome the programming

obstacles experienced by the original CRAY-2. Additionally, the ability to support substantially larger, albeit distributed, memories benefited a huge range of applications; and the I/O architecture, which separated I/O from message-passing traffic, was a shared resource to the entire system, unlike earlier clusters. Coalescing all of these attributes into a single, coherent, easily usable, and manageable total system was UNICOS/mk, the first general-purpose distributed operating system. In the best Cray tradition, the T3E was truly balanced in respect of processor power, memory, and I/O, all combining to deliver maximum performance to the end-user through an efficient software system.

Also, for the first time, true supercomputers were on the same Moore's Law development curve as microprocessors. Users could look forward to increased processor power becoming available at lower cost every 2 years or so, rather than the 4-year cycle that had been typical of earlier supercomputers.

With the advent of the T3E the industry was clearly and demonstrably heading toward enabling scientific users to access unlimited computational power, constrained only by their ability to develop highly scalable applications programs and the size of their budget. Contemporary (2010) systems are now achieving PetaFlop performance levels, barely more than a decade after ASCI Red and the T3E first broke the TeraFlop application barrier in 1998. ExaScale systems, a million-fold increase over the T3E, are within reach. Operational systems with more than 200,000 PEs (cores) are available today, and million-core systems are being contemplated.

Clearly, the continuing successful growth of HPC computing power, and resultant expansion of the range of beneficial applicability, now forms a "virtuous circle" driving that growth faster than Moore's Law. However, it is strongly believed in some circles that major processor and memory changes will be required to take HPC from the PetaFlop to the ExaFlop range. Those changes may dissolve the currently productive marriage of commodity components and HPC, conceivably requiring another T3E-like development effort.

After more than a decade of research and experimentation, the T3E had given the users real leverage based on the relentless progress and economics of the microprocessor industry. The T3E hardware and software had finally bridged the gaps between

commodity and high performance, and had become the industry's first widely used MPP system. The T3E user environment and system tools provided an effective platform to develop and run production applications at very high scale. Applications programmers and system managers responded enthusiastically to their new tools, and the installed base of almost 100 T3E systems established a new baseline of expectations among users and the industry that became the foundation for the next phase of the supercomputing roadmap.

The singular T3E contribution was "proof of concept." DARPA, DOE laboratories, NSF researchers, Cray Research, Intel, TMC, and others had collectively formulated the vision. The CRAY T3E provided hard evidence that those ideas could indeed be transformed into usable, flexible, resilient physical reality at a new, more attractive, price point.

Acknowledgments

The authors wish to acknowledge the contribution of many former colleagues who made helpful suggestions and thoroughly reviewed the text for technical and historical accuracy. Specifically, they wish to thank Mike Booth, Bill Minto, Steve Scott, David Wallace, and William White.

Cray Vector Computers

JAMES L. SCHWARZMEIER
Cray Inc., Chippewa Falls, WI, USA

Synonyms

Pipelining; SIMD (Single Instruction, Multiple Data) Machine

Definition

Vector processing at the instruction set level is translation of a high-level language program into a series of scalar and *vector* instructions. Vector instructions use scalar and vector registers, where each vector register holds a maximum number of elements as specified by implementation-specific MAXVL, typically in the range 32–256. Vector processing at the hardware level executes VL elements of an instruction in *parallel* across an implementation-specific number of lanes or



pipes. Each pipe contains a complete copy of functional units and possibly memory data paths and a colocated subset of elements of the vector register file. Vector Instruction Set Architectures (ISAs) allow easy translation by a compiler of loop-level code parallelism into compact instructions that are very efficient in terms of fetch, decode, and issue. Vector ISAs lend themselves to straightforward replication of hardware components for achieving high levels of parallel execution across multiple functional units and multiple pipes. Vector processors are an easy and natural way to provide long periods of uninterrupted pipelined operation for high processor efficiency.

Discussion

The discussion of Cray vector systems is presented in three parts. First, a historical perspective of the rise, domination, and decline of Cray vector systems is presented. The focus is on reasons for this evolution. Second, a technical discussion of architectural and performance advantages of vector systems is presented. These fundamental advantages drive the introduction of vector-enhanced architectures seen today in the x86 SSE and AVX instructions set extensions and SIMD designs of graphics processing units (GPUs). Finally, a chronology of Cray vector systems is presented from the Cray 1 in 1976 to the Cray X2 in 2007.

Introduction and Historical Perspective: The Rise, Domination, and Decline of Cray Vector Systems

Cray vector computers played a unique role in the history of high-performance computing (HPC) and advancement of computer architectures. The genius of Seymour Cray as evidenced by his long track record of designing high-performance computers with Control Data Corporation (CDC) and introduction of the Cray-1 in 1976 earned him the accolade of “Father of the Supercomputer Industry.” The Cray-1 was introduced with a Reduced Instruction Set Computer (RISC) processor and system clock of 80 MHz, whereas state-of-the-art microprocessors of the time ran with 1–2 MHz. This gave the Cray-1 a huge advantage even for scalar codes, and all customer codes experienced immediate and significant speedup compared to microprocessors. As Cray’s vectorizing compiler improved and customers realized how to write vector loops, an additional

improvement of ×5–10 was possible. Fast scalar processing and low memory latency resulted in excellent vector performance for “short” vector loops compared to competing “long” vector machines. The hallmark of Cray systems has been *balance* between CPU computation, memory bandwidth, IO bandwidth, and software stacks to deliver performance.

Government laboratories had in the Cray-1 a computer capable of doing long simulation runs of realistic physical problems. While 8 MB memory capacity of the Cray-1 is tiny by current standards, scientists of the day were able to resolve problems in 1D and 2D geometries. There also was a base of government data-analysis users whose codes benefited from fast and extensive 64-bit integer and logical operations and random access to memory. Soon scientists and engineers in many private companies were buying Cray vector computers to gain competitive advantage by reducing time to market for new products. Early purchasers of Cray supercomputers were from industries such as petroleum, car-crash simulation, computational chemistry, aerospace, structural mechanics, electronics, and weather/climate. Cray vector machines were so much faster than alternatives that, despite \$10–20 M price tags, demand for the technology allowed Cray Research to grow rapidly. Following the Cray 1 were many generations of Cray vector systems with increasing processor count, memory capacity, clock rate, peak performance, and memory bandwidth. During the mid-late 1980s Cray Research owned approximately 70% of the HPC market.

While Cray systems dominated the HPC market for many years, technology changes in the industry and interest from computing companies much larger than Cray Research began eroding competitiveness of Cray vector machines. As IC technology advanced during the late 1980s, some performance advantages of Cray vector systems became price-performance *disadvantages*. Cray Research did not have resources to design full-custom logic CMOS chips, as was done by large microprocessor companies. Dense CMOS logic allowed microprocessors to have large, low-latency, multilevel caches, which are an inexpensive way to provide bandwidth. Low latency to cache is important to prevent stalls of the processor due to allowing only a small number of outstanding memory references. Microprocessor companies chose to design memory hierarchies

with a bandwidth profile strongly peaked at lowest level cache with modest main memory bandwidth based on commodity DRAM chips. Cray systems traditionally were designed with no caches and high-bandwidth crossbar switches to SRAM memory parts that were banked across individual double words rather than cache lines. This provided a uniform memory access (UMA) shared memory system and was ideal for applications with no spatial or temporal locality. For applications with good locality of reference it became more cost-effective to run on microprocessor systems.

Fast, low-density ECL chips and SRAM memory parts resulted in Cray vector machines with many chips on many printed circuit boards and many wires connecting them – expensive components. New microprocessors of the era, such as the IBM Power 2 and DEC Alpha series, were getting close enough in absolute performance to low-end vector machines that *price/performance* became a significant part of the customer's buying decision. This trend continues to the present.

The HPC market is much smaller than the general microprocessor market. It was difficult for Cray systems to realize economic advantages of large-volume production. This problem was exacerbated as low-end customers began shifting from writing vector code to scalar code, since their applications were being targeted for the larger volume superscalar market. This was especially true of Independent Software Vendor (ISV) codes. By the mid-1990s application codes without significant vector content ran faster on microprocessors than on vector systems, as dictated by Amdahl's Law.

In the mid-1990s there were two other intertwined innovations in the computer industry that signaled a shift away from vector systems. First, massively parallel processor (MPP) machines began entering the marketplace. These systems used hundreds or thousands of commodity microprocessors with locally attached DRAM memory and connected with low-bandwidth networks. Second, what made these systems usable and ultimately led to a new *de facto* standard architecture for HPC systems was a standard *distributed memory* inter-processor programming model. The initial standard model was the Processor Virtual Machine (PVM) interface. In time PVM gave way to today's standard – the Message Passing Interface (MPI) model. The 1987 Gordon Bell Award was awarded to Benner, Gustafson, and Montry at Sandia National Laboratory

(SNL). The scientists did weak scaling studies and used a distributed memory message passing model to give $\times 400\text{--}600$ parallel speedups on a 1024 node NCUBE on three applications from SNL.

While early MPP systems had performance issues and lacked production-quality robustness, these systems were an affordable means to scale the number of processors and amount of memory to levels much higher than supported on shared memory vector systems. However, it can be challenging for even MPP systems to run well when processor counts are in the range $10^4\text{--}10^5$. Cray Research was a late comer to the MPP marketplace, but used its experience in delivering HPC systems to contribute to this market segment. During the 1990s the Cray T3D and T3E MPP systems using the DEC Alpha EV4 and EV5 series of processors soon dominated the MPP marketplace, largely due to unique features supported in the Cray proprietary network interface chip and router.

There is nothing that requires the processor in a distributed memory MPP to be a scalar processor. Cray engineers designed several scalable, distributed memory systems with powerful *vector processors* as compute elements. The Cray SV1 system introduced in 1998 was an early version of *multi-threaded vector* operation. Users could specify with a compiler flag that four vector processors on different modules could operate either as four independent vector processors, or as a single, four-way *multi-threaded* vector processor. The Cray SV1 was followed the Cray X1 system introduced in 2003 and the Cray X2 system introduced in 2007. The Cray X1 improved multi-threading by placing the four vector processors on a Multi-Chip Module with tight synchronization. These systems combine massive parallelism *across* many compute elements and *within* each compute element.

While traditional shared memory Cray vector systems are no longer produced, vector hardware concepts are key to two advances in recent HPC architectures. First, x86 processors are moving toward improving performance through SSE and AVX instructions, which are akin to fixed vector lengths of 2, 4, or more elements per register. Second, graphics processing units (GPUs) gang together a moderate number (8–64) of threads of execution in a single-instruction-multiple-data (SIMD) manner. Equally important to vector hardware concepts are vector *compiler* technologies for efficient use of SSE and



AVX instructions and GPUs. Cray compiler techniques for handling outer loop, four-way thread parallelism, and inner-loop vector parallelism with predicated execution of the Cray X1 and X2 are directly applicable to compiling for multi-threaded vector parallelism of GPUs.

The historical perspective of Cray vector systems ends on the note that vector processing will continue to be an important component of future HPC systems. For this reason, it is helpful to understand in more detail the breadth and inherent advantages of vector architectures.

Architectural and Performance Advantages of Cray Vector Systems

Parallel Hardware Needs Parallelizing Compilers and Parallel Application Codes

For HPC users to benefit from increased parallel capability of hardware, HPC application codes must be written so as to allow compilers to identify at multiple loop levels task and vector parallelism. The triad of parallel hardware, parallelizing compilers, and parallel application codes are equally dependent on one another for efficient utilization of modern HPC systems. The Fortran programming language introduced by John W. Backus of IBM in 1953 was well-suited for expressing numerical discretization and similar algorithms that an auto-parallelizing and vectorizing compiler could analyze.

Application programs can contain parallelism at several levels. At the lowest level, a compiler generates many instructions for each line of code written with a high-level language. Generally groups of these instructions involve different registers and can be issued in parallel on the same clock cycle. This is instruction level parallelism (ILP), and is supported on all processors today. *Vector* processors target intermediate-level *data* parallelism when operations between elements of vectors or arrays are independent and can be expressed as a *single* instruction. An example of a simple vector loop is a TRIAD operation

```
Do i=1,N ← vector parallel over do-i
  z(i) = x(i) + a*y(i)
Enddo
```

A vectorizing compiler with maximum vector length 64 breaks this loop in chunks of up to 64 iterations. Vector instructions for TRIAD load 64 values of x(i) with a

one instruction, 64 values of y(i) with one instruction, do 64 multiplies of a vector times a scalar with a single instruction, etc. For the final iteration vector length is reduced as necessary, on architectures that support it. Vector parallelism also could be considered data level parallelism (DLP).

At the third level is loop-level *task* or *thread* parallelism. Here iterations of the loop contain independent streams of either purely scalar instructions or mixed scalar and vector instructions. For example, a task-parallel TRIAD loop is

```
Do j = 1,M ← task parallel over do-j
  Do i = 1,N ← vector parallel over do-i
    z(i,j) = x(i,j) + a*y(i,j)
  Enddo ! i
Enddo ! j
```

This thread parallel loop could be detected with an auto-parallelizing compiler, or could be identified by user-inserted OpenMP directives before the do-j loop. A vectorizing compiler will translate the inner loop into scalar–vector instructions, whereas a non-vectorizing compiler translates the loop into purely scalar instructions.

The highest level parallelism in HPC codes today is inter-processor parallelism, usually programmed with the MPI model.

Pipelining and Vector Processing

The design of efficient HPC processors is inexorably linked with the concept of *pipelining*. Vector architectures are a natural and effective way to implement pipelining.

Hardware pipelining is the decomposition of logic blocks and data paths into a series of stages that allow back-to-back transmission of intermediate results or final data packets associated with a computation through the processor and memory system. The goal of pipelining is that at every clock cycle functional units and data paths can accept new input data, perform an operation, and advance output data to the next stage of the pipeline. For example, in processors today a floating point multiply operation might be broken into four stages, which leads to a four clock latency for a single multiply to complete. Moreover, in a pipelined design the first and subsequent stages are able to accept new

input operands *every clock*. For example, in a hypothetical pipelined implementation with 4 clock multiply latency the time to complete a stream of 64 MULTIPLY operands is 67 clocks – 4 clocks for the first result to finish the last stage of the multiply unit, plus 63 clocks for each successive result to finish the last stage of the multiply unit. It is apparent that a vector register containing 64 elements provides a natural implementation for providing a “pipelined stream of 64 operands.”

Vector Pipes and “Chime” Time

To boost performance, vector processors can be designed with multiple *pipes*. An n -pipe implementation contains $1/n$ of the maximum number of vector elements of all vector registers colocated with a complete copy of all functional units and possibly data paths to the memory system. In a vector processor with two pipes, a pipelined stream of 64 operands would complete in 32 clocks. The *chime time* of a vector implementation is

```
Vector chime time = MAXVL/#pipes,
```

where MAXVL is the maximum number of elements in a vector register. The biggest challenge for a processor to maintain pipelined operation is in the memory system – it is difficult and expensive to design a memory system that provides load operands in a fully pipelined manner. Pipelining of a memory system extends from address translation, to cache coherence directory consultation (if caches are present), to crossbars and banks of the memory system, and back to processor registers. A well-pipelined memory system can be illustrated with a Cray C90, first introduced in 1991. The Cray C90/16 was a 16-processor shared memory system that ran at nominal 250 MHz with 512 SRAM memory banks, each capable of providing a 64 bit double word (dword) every 4 clock cycles. This provided a peak bandwidth at the banks of $(512 \text{ banks})^*(8\text{B/bank}/4 \text{ clocks})^*(0.25 \text{ GHz}) = 256 \text{ GB/s}$. Each Cray C90 processor had two vector read ports and one vector write port with the goal of supporting high performance for memory-intensive loops like TRIAD. The maximum memory bandwidth a single processor could ask for was four dwords load data and two dwords store data per clock for total of 6 dwords per clock. 16 processors of a Cray C90/16 had a maximum memory request rate of $(16 \text{ proc})^*(6\text{dwds/clock/proc})^*$

$(8\text{B/dword})^*(0.25 \text{ GHz}) = 192 \text{ GB/s}$. The bandwidth at the banks was over-provisioned compared to peak processor request rate by factor $256/192 = \times 1.3$. This was to compensate for inevitable contention between processors in crossbars and memory banks. In practice a fully loaded Cray C90/16 could compute TRIAD with aggregate bandwidth of about 102 GB/s, which is 53% of the maximum processor request rate.

Advantages of Vector Processors

Traditional vector processors have significant advantages over traditional superscalar microprocessors for HPC applications. These include: (a) reduced instruction issue bandwidth requirements, (b) high processor concurrency for latency tolerance, (c) good fit with IC technology for functional units and registers, and (d) vector *chaining*.

(a) Reduced instruction issue bandwidth requirements

A significant advantage of vector ISAs is reduction in instruction issue bandwidth requirements. This lowers power and silicon area requirements for instruction fetch and decode. Vector ISAs afford an easy way for the compiler to convey data parallelism to hardware. For example, with MAXVL = 64 vector load/store instructions have one of two generic formats

<code>v1 [a1, a2] ← constant stride load</code>
<code>[a1, a2] v1 ← constant stride store</code>
<code>v1 [a1, v2] ← gather (indexed load)</code>
<code>[a1, v2] v1 ← scatter (indexed store)</code>

For the constant stride case, the base address of the load is in address register a1 and the stride of the load/store is in register a2. For the gather/scatter case (indirect addressing), vector register v2 contains 64 integer values that are word-offsets relative to the base address of each successive element of the vector reference. The destination register of the load or operand register of the store is v1. A *single* vector load instruction loads 64 words from memory with all 64 addresses generated *automatically* in vector hardware – element-by-element address increment instructions are not required. A scalar processor requires 64 load instructions and 64 address increment instructions for a total of 128 instructions. On a vector processor *scalar* instruction issue bandwidth can be relaxed, since scalar execution in support of vector loops can be “hidden” under vector

chime time. The Cray-1 had length 64 vector registers and one vector pipe, so its chime time was 64 clocks. This gave ample opportunity for scalar instructions to issue without interrupting back-to-back issue of vector instructions. The reduced instruction issue requirement of vector processing is taken advantage of when the Intel x86 ISA added SSE and AVX instructions, which have fixed vector length of 2, 4, or 8, and in GPUs which execute SIMD manner with an equivalent maximum vector length in the range 8–64.

(b) High concurrency for latency tolerance

A second advantage of vector processing is the large concurrency available to hide memory latency. The register file in a Cray X2 processor introduced in 2007 has 32 vector registers with 128 elements each for a total of $128 \times 32 = 4096$ doubleword (quadword for x86) elements. The implementation of the Cray X2 allows each processor to have up to 4096 outstanding dword load elements. The Cray X2 has hardware support for direct load/store access to any address in the system. Each processor has sufficient concurrency to tolerate local memory latency of 310 ns and even remote memory latencies of several microseconds.

(c) Good fit of vector hardware to IC technology

A third advantage of vector processors is they are a good match to IC technology in terms of functional units and register files. The vector register file is large (good for pipelined operation) and simple to control. Each vector processor of the Cray X2 has eight vector pipes. From a logic design perspective, adding vector pipes is a straightforward replication of functional unit groups (FUGs) for each pipe with each FUG having two read ports and one write port into its per-pipe registers. For example, FUG1 on the Cray X2 contains FP Add, INT ADD, a LOGICAL unit, COMPARE, etc. FUG2 contains FP MULTIPLY, INT MULTIPLY, and a SHIFT. The vector load/store unit also has ports into and out of the per-pipe register file. One vector instruction from each FUG can issue per chime, as long as there are no issue conflicts. In contrast, a superscalar microprocessor has a more complex and smaller register file. In a 4-way-issue machine, in *each clock four instructions* for correctness need full access into any individual element of the register file. Issue

logic is more complex, since register and FUG availability and data path conflicts have to be examined to determine how many instructions can issue in the next clock. Complexity of scalar register files grows quadratically with number of registers. Vector register file size can grow with no increase in complexity by increasing maximum vector length.

(d) Vector “chaining”: vector processing of dependent instruction streams

Vector register files and chime time lead to an architectural advantage of vector processing related to pipelining called *chaining*. Chaining occurs when an output vector register of an instruction executed in one functional unit is pipelined as an input vector register into a *dependent* instruction in a different functional unit. Register bypass paths between functional units are standard in microprocessors, but chaining extends this concept with large vector length. From a compiler perspective, chaining allows vectorization *across* independent sets of *dependent* instructions. This architectural advantage of vector processing merits exploration with a concrete example.

The top of Fig. 1 shows a loop of 64 iterations containing a line of code with pairwise-dependent MUL-TIPLY/ADD instructions. Figure 1a shows instruction sequences for Y(I) on a superscalar processor (left half of table) and vector processor with MAXVL = 64 (right half of table). Not shown are load instructions (presumed to hit in cache) that fill input registers R0 on the scalar side and S0 on the vector side. Calculation of Y(I) is parallel (vectorizable) over I, but assume register pressure from other statements in the loop body prevents a superscalar compiler from unrolling or software pipelining calculation of Y for different I. The assumption of no pipelining on the scalar processor may seem unnecessarily restrictive, but the point is to illustrate the advantage of the *length* of registers rather than the *number* of registers. This effect has been seen in real application codes. Consider execution of this line of code on an out of order (OOO), 4-way superscalar processor and an in-order, single-pipe vector processor with MAXVL 64. Assume both processors can load or store one doubleword per clock from cache with 3 clock load latency, and both processors have MULTIPLY and ADD latencies of 4 clocks.

```

DO I = 1,64
...
Y(I) = X1(I)+A*( X2(I)+A*X3(I) )
...
ENDDO

```

← code with long-lived registers

← code with long-lived registers

Instruction #	Instruction	Issue clock	Instruction	Issue clock
	(i=1) [A in R0]		(I=1:64) [A in S0]	
1	R3 [load X3(1)]	0	V3 [load X3(1:64)]	0
2	R10 R3*R0	3	V10 V3*S0	3
3	R2 [load X2(1)]	1	V2 [load X2(1:64)]	1+64=65
4	R11 R2+R10	3+4=7	V11 V2+V10	65+3=68
5	R1 [load X1(1)]	2	V1 [load X1(1:64)]	65+1+64=130
6	R12 R11*R0	7+4=11	V12 V11*S0	130+1=131
7	R10 R1+R12	11+4=15	V11 V1+V12	131+4=135
8	[store Y(1)] R10	15+4=19	[store Y(1:64)] V12	135+4+64=203
...	(i=64)			
1+8*63=504	R3 [load X3(64)]	63*19=1197		
505	R10 R3*R0	1197+3=1200		
...				
523	[store Y(1)] R10	1197+19=1216		

a

Instruction #	Explanation of 'Issue clock' from (a)	
	Superscalar processor	Vector processor
1	Issue @ clock 0	Issue @ clock 0
2	Issue @ 3, cache latency 3 clocks	Issue @ 3, first load element from cache enters MUL unit
3	issue @ 2 since OOO and can issue one load or store per clock	Issue @ 65, 2nd load cannot start until previous load completes
4	Issue @ 7, MUL latency 4 clocks. R2 is ready before R10	Issue @ 68, in-order issue plus cache latency of 3 clocks
5	Issue @ 2 since OOO	Issue @ 130, after last element of previous load completes
6	Issue @ 11, ADD latency 4 clocks	Issue @ 131, since in-order issue
7	Issue @ 15, MUL latency 4 clocks	Issue @ 136, MUL latency 4 clocks
8	Issue @ 19, ADD latency 4 clocks	Issue @ 139, but add 64 clocks since store busies cache before subsequent load can issue
...		
504	Issue @ 504 (64th iteration)	
507	Issue @ 507, cache latency 3 clocks	
...		
523	Issue @ 511, 504+19=523	

b

Cray Vector Computers. Fig. 1 Illustration of vector chaining for loop with dependent pairs of MUL/ADD instructions whose loads hit cache with 3 clock latency. Floating point functional unit latency is 4 clocks. (a) Instructions with issue times for OOO superscalar processor (*left*) and in order, MAXVL = 64 vector processor (*right*). (b) Explanation of issue times for each instruction and processor

Figure 1a is a table showing when each instruction issues on the superscalar and vector processors. The timing table reveals differences in issue constraints and

performance between the two processor types. Figure 1b is a table explaining what determines issue time of each instruction on the two processors. On the superscalar

processor loads of one scalar value can issue every clock and issue OOO. The left half of [Figure 1a](#) shows that by clock 4 performance is limited by the 4 clock *functional unit latencies* of MUL and ADD instructions, leading to a total of 19 clocks for *each* iteration I. Total time for this one line of code for 64 iterations is approximately **1216** clocks.

On the other hand, on the vector processor there is a single vector iteration of length 64 for calculating $Y(1:64)$. Vector loads or stores can execute only once every 64 clocks, since each vector memory reference busies the data path to cache for 64 clocks. As can be seen from the right half of [Figure 1a](#), functional unit latency on the vector processor just adds a few extra clocks to execution time. Issue timings (column 5 of [Fig. 1](#)) for instructions 1, 3, 5, and 8 show that performance is limited by memory system bandwidth – functional unit latencies are basically hidden by pipelining from vector registers. To vector execution time (see green “64” in instruction 8 of [Fig. 1a](#)) is added time for the vector store to complete, as the first vector load following calculation of $Y(1:64)$ cannot begin until the store relinquishes the data path to the cache. Total time for this one line of code for 64 iterations is approximately **203** clocks.

The difference in performance between the vector and superscalar processors for calculation of $Y(1:64)$ is approximately **6×** ($1216/203$). This dramatic speedup is due to *vector architecture*, even though the processors have *equal* peak performance and cache bandwidth. It is much more efficient to stream 64 elements from cache and pipeline them through MULTIPLY and ADD functional units, than to individually fetch words from cache and pay MULTIPLY and ADD latency separately for each iteration. For the *complete* loop in [Fig. 1](#), the difference in performance between the two processors will be less than $6\times$, depending on how much work is in parts of the loop not shown. If arrays $X1(I)$, $X2(I)$, $X3(I)$, and $Y(I)$ are not in cache, performance on both systems will be limited by memory bandwidth and possibly memory latency (superscalar case, due to low concurrency). Of course, to the extent scalar pipelining is possible, the relative speedup of the vector case will be reduced.

Chronology of Cray Vector Systems

While the basic Cray-1 ISA persisted within Cray for nearly 30 years, there have been many implementation

improvements over the years. The following are highlights of Cray vector (and non-vector) systems beginning with the Cray-1. Some specifications are nominal, since different models of a given system can ship with slightly different parameters.

Cray-1S/M

The Cray ISA was one of the first Reduced Instruction Set Computers (RISC) in a commercially available computer. There were 8 integer-only address registers (24-bit) A0–A7, eight general-purpose scalar registers (64-bit) S0-S7, and 8 vector registers (64-bit) V0-V7 with MAXVL = 64. 64 B and T registers were used to do block loads of scalar data from memory and as spill space for A and S registers, respectively. The clock speed was 80 MHz for a peak performance of 160 MFLOPS. The Cray-1 was a single processor system with no caches and up to 32 MB of bipolar memory and supported a load path and store path to memory. Memory latency was 11 clock cycles. Memory was dword-addressable (rather than byte-addressable) and *real* as opposed to *virtual*. [Figure 2](#) shows Seymour Cray and the Cray-1 supercomputer. The iconic “C” shaped footprint allowed routing short-length wires between processor and memory modules. The padded seat perimeter housed power supplies and plumbing for the liquid cooling system. The IO system was housed in a separate cabinet. Dedicated IO processors could read and write main memory and disk without diverting the CPU from computation. An optional solid state disk (SSD) cabinet delivered much higher performance for codes that repeatedly accessed datasets too large to fit in central memory. First customer ship was in 1976.

Cray XMP2/4

The Cray XMP came in 1-4 processor models. The Cray-1 ISA was expanded to add 8 “shared” and 8 “semaphore” registers. These registers allowed fast synchronization between processors and IO resources. A later version of the machine was the Cray XMP-EA (Extended Architecture). Expansion of A and B registers to 32 bits permitted the architecture to address up to 4 GWords (64-bit double words) main memory. Clock speeds up to 117 MHz gave peak performance of 235 MFLOPS per processor. The number of paths to memory increased to two loads and one store path per processor. Memory latency was approximately 13 clock cycles.



Cray Vector Computers. Fig. 2 Seymour Cray and the Cray-1 supercomputer

Flexible chaining was introduced to allow more frequent overlap of functional units. Pioneering work was done on the Cray software stack to implement Macro-Tasking™ libraries that harnessed multiple CPUs on a single user program. A second version of parallelizing software, optimized for finer granularity parallel work, was Micro-Tasking™. User-inserted directives were inserted before a parallel loop, which the compiler translated into efficient work-sharing code using low-latency shared registers. It was common to get parallel speedups greater than 3.5 on a four-processor Cray XMP. The Cray XMP was introduced in 1982.

Cray-2/4

The Cray-2 was renowned for two innovations. First main memory was expanded up to 4 GB, which allowed for much higher resolution scientific computing. Large memory was made possible by use of high density but low bandwidth DRAM chips. This allowed scientists and engineers to begin 3D simulations. Second, the system was very compact – its cabinet footprint was a

1.35 m circle and its height was 1.15 m. The computer used total immersion Flourinert cooling, with plumbing to a distantly located cooling tower. The processor ran at 244 MHz for a peak performance of 488 MFLOPS per processor. The Cray-2 also introduced a 128 KB, compiler-managed, local memory. The Cray-2 system ran the UNICOS operating system, which started the shift toward UNIX-based OS within Cray. The Cray-2 was introduced in 1985. Like the Cray-1, the Cray-2 system was designed by Seymour Cray. Follow-ons Cray 3 and Cray 4 were built by Seymour Cray as part of Cray Computer Company in Colorado Springs during the early 1990s. The Cray 3 and Cray 4 pushed miniaturization of packing and were difficult to manufacture in volume.

Cray YMP/8

The Cray YMP was the follow-on to the Cray XMP and had from 1 to 8 processors. Clock speed of 167 MHz gave peak performance of 333 MLOPS per processor. Memory latency was approximately 17 clock cycles. There were several versions of the YMP designed for different market needs. The baseline system came with a 1 GB SSD and 256 MB memory, but for memory-intensive applications a DRAM memory part option allowed up to 32 GB memory. For low-end markets the YMP-EL was an air-cooled CMOS implementation of the YMP.

Cray C90/16

The Cray C90 had up to 16 processors. Clock speed was a nominal 250 MHz. An implementation change relative to the YMP was to have *two* vector pipes with vector registers 128 elements each, which boosted performance to 1 GFLOP per processor. To maintain processor and memory bandwidth balance, each vector pipe had two load and one store paths to memory. Memory latency was about 23 clocks, and memory capacity was 0.5–8 GB. The C90 was introduced in 1991.

Cray T3D MPP

Though not a vector system, the T3D was Cray's first MPP system. The T3D was hosted by a Cray YMP Model E system, which ran the Cray UNICOS operating system and provided I/O and most system services to the MPP. DEC Alpha EV4 150 MHz processors were the compute engine of a node and ran a microkernel



operating system. Each node contained two EV4 processors connected to a proprietary network interface chip (NIC). The NIC arbitrated access of the two processors to a Block Transfer engine (BLT), which performed direct memory access (DMA) transfers to/from the local node to remote nodes via a link to a proprietary router chip. The router chip supported bidirectional 3D torus connections to other router chips in the system. The first Cray T3D system was shipped in 1993 to the Pittsburgh Supercomputer Center (PSC).

Cray CS6400

Cray Research acquired the assets of Floating Points Systems in 1991, which led to formation of subsidiary Cray Research Superservers in 1993. These systems were based on Sun Microsystems' SuperSPARC microprocessor and utilized high-bandwidth buses in a fault-tolerant, partitionable SMP configuration. The Cray CS6400 team was developing the Starfire system when it was sold to Sun Microsystems after SGI purchased Cray Research in 1996. The Starfire became Sun's very successful Enterprise 10,000 system.

Cray J90/32

The Cray J90 had 8, 16, or 32 processor models. This machine was the follow-on to the Cray YMP-EL, so it was an air-cooled CMOS implementation. The J90 had two noteworthy innovations. First the CPU was split into two chips, one for scalar processing and one for vector processing. Initially both chips ran at 100 MHz, but the Cray J90se (scalar enhanced) model ran with 200 MHz on the scalar chip. The second innovation (for Cray) was presence of a 1KB *scalar* cache. Presence of a cache-based processor introduced new issues for Cray designers, such as maintaining coherence between scalar and vector memory transactions. The cache was software managed, so the compiler was responsible for hardware invalidations when required. The J90 began shipping in 1994.

Cray T90/32

The Cray T90 had 4, 8, 16, and 32 processor models. The T90 had two innovations. First was addition of a mode that would use either IEEE or traditional Cray floating point format. IEEE mode was particularly important to ISVs who wanted single source code to run on many platforms. Second was a redesigned memory

system in which vector memory references no longer had synchronized access to data paths, but requests and responses were individual packets for each element of the vector reference. Vector loads no longer completed in order, and the processor could issue multiple vector loads (however, still only 8 vector registers) without stalling unless a destination register in an instruction was not completely full. A T90/32 could execute the STREAMS benchmark at 360 GB/s. The clock rate was 450 MHz, which required Fluorinert cooling. Like the Cray C90, the T90 had two vector pipes, so its peak compute rate was 1.8 GFLOPS per processor. The T90 was introduced in 1994.

Cray T3E MPP

The Cray T3E system was a self-hosted follow-on to the Cray T3D system. The 3D torus network was maintained from the T3D, although the T3D used *dimension-order* routing while the T3E used *direction-order* routing. The latter routing enabled better network fault tolerance. Link bandwidth and functionality of the T3E router chip were enhanced over the T3D. Two T3E router enhancements were adaptive routing of packets as network contention was detected (also improved fault tolerance) and a virtual barrier network. The T3E used a single DEC Alpha EV5 or EV56 processor connected to the NIC. One of the most significant new features of the T3E NIC was "E registers." There were 512 user-accessible E registers that were manipulated by the local processor to do direct, un-cached DMA transfers to/from memory of remote nodes. The ability to easily use E registers led to the successful SHMEM communication model, which implemented the first direct, high-performance GET/PUT semantic communication model for MPPs. The first Cray T3E system was delivered to PSC in 1996. The T3E's performance and scalability led it to become the gold standard of the MPP market in the late 1990s.

Cray SV1/32

Silicon Graphics purchased Cray Research in 1996. The Cray SV1 was completed in the Cray Research division of Silicon Graphics. The SV1 was binary compatible with the Cray YMP and J90, so computed solely using traditional Cray floating point arithmetic and had vector registers with 64 elements each. There were three innovations introduced in the SV1. First each SV1 cabinet

had a 256 KB software-coherent merged scalar–vector cache. For vectors the cache primarily served to filter bandwidth rather than reduce latency. Second the concept of “Multi-Streaming” was introduced. Here, under control of software, groups of four processors of an SV1 cabinet could be ganged together at the task level of parallelism in a program to serve as a single, 4-way multi-threaded vector processor. This was the first attempt within Cray to develop a more powerful “single processor,” but the concept was little used due to lack of key hardware support. The third innovation was multiple SV1/32 cabinets could be combined with a Cray proprietary Giga-Ring™ interconnect to form a “Scalable Vector” system up 32 SV1/32 “nodes.” The original SV1 systems ran at 300 MHz, and the SV1ex follow-on ran at 500 MHz. Peak performance of a single processor was 1 GFLOP and 32 GFLOPS for the 32 processor node of SV1ex. A full 32-node system had a peak of 1.0 TFLOPS. The Cray SV1 began shipping in 1998.

Cray XMT

In 2000 Tera Computer Company bought the Cray Research division of Silicon Graphics. The new company was renamed Cray Inc. The experience of Tera engineers in designing and building highly multi-threaded systems led to the Cray XMT system. This system was uniquely able to satisfy market needs for applications with abundant levels of non-predictable, graph-based parallelism.

Cray X1

The Cray X1 was the first truly scalable vector-based system designed by Cray. It shipped in late 2002. As described below the Cray X1 system had a limited version of coarse-grained multi-threaded vector operation, which shares with Cray XMT systems some of the same compiler techniques for loop-based task and vector parallelism. The Cray X1 had many innovations over past Cray vector machines.

1. There was a major upgrade of the Cray-1 ISA: The Cray X1 ISA supported only IEEE arithmetic; 64 64-bit A registers; 64 64-bit S registers; 32 64-bit V registers with 64 elements each. 8 vector mask (VM) registers, each with 64 elements and one-bit wide, were added to allow rapid evaluation

of conditional code using predicated execution. Furthermore, almost all vector instructions were subject to predicate execution as specified by VM registers, element by element. A relaxed memory model was defined with a rich set of synchronization and Atomic Memory Operation (AMO) instructions were defined.

2. There were two processor modes. In Single Stream Processor (SSP) mode, each vector/scalar CPU was a separate processor. In Multi-Streaming Processor (MSP) mode, a group of four SSPs packaged on an MCM was treated by the compiler as a 4-way multi-threaded vector processor. Applications were compiled either in SSP mode or MSP mode via a compilation flag. The compiler could either automatically detect outer loop task-level parallelism with vector code generation on inner loops, or the compiler could key off user-inserted Cray Streaming Directives (CSDs) to schedule SSP threads across loops possessing subroutine call trees. Each SSP thread could discover vector parallelism as the thread traversed the call tree. A fast MSYNC synchronization instruction was both a control and memory barrier between SSPs in an MSP.
3. Each SSP processor was a two-pipe CPU with vector registers of 64 elements. The scalar part of the SSP ran at 400 MHz while the vector unit ran at 800 MHz. This led to a peak 64-bit rate of 3.2 GFLOPS, and 32-bit mode had peak of 6.4 GFLOPS.
4. Each SSP was operated in a fully *decoupled* manner. The first level of decoupling was that scalar execution ran “ahead” of vector execution by hundreds of clock cycles in each SSP. The scalar processor issued, completed, and marked for graduation scalar instructions early while vector instructions were dispatched to deep queues to await scalar operands, if needed. The second level of decoupling was early, non-blocking processing of vector load addresses into the memory system to prevent into load buffers data that would be moved into a vector register when the load executed at the vector execution pipeline. Vector store addresses were also decoupled from vector store data to allow for pre-allocating store requests in cache.
5. The MSP had peak 64-bit flop rate of 12.8 GFLOPS. The four SSPs on the MCM shared a 2 MB L2 cache,

while each SSP had a 64 KB scalar cache. High bandwidth directory caches in the Memory (M) controller chips processed cache coherence messages without degrading performance.

6. Each of four MCMs on a node was connected to 16 M chips with peak nodal memory bandwidth of 200 GB/s. Each M chip had two bidirectional cache-line address sliced ports into a proprietary Cray X1 network. The network had multiple 2D slices between cabinets, and had cross-bar routing among nodes in a cabinet.
7. The Cray X1 was scalable up to 1024 nodes: fully cache coherent system but only caching node-local data; entire system is globally addressable – processor and network support native ISA remote scalar and vector memory references with “remote translation” (map all system memory so there were no network TLB misses). The latter feature was borrowed from the Cray T3E NIC.

Cray X2

The Cray X2 was the follow-on to the Cray X1 with a number of noteworthy differences. First the concept of MSP was dropped. On the X1 some applications ran very well in MSP mode, but the marketplace generally favored SSP mode as being more efficient – Amdahl’s Law exacts a large penalty if parts of an application do not have task-level parallel loops to use the 4 SSP threads. Second, each X2 CPU was an 8-pipe vector unit with vector registers of 128 elements each, double the length in the Cray X1. Scalar and vector clock rates were 800 MHz and 1.6 GHz, respectively. This resulted in peak performance of 25 GFLOPS per processor. Each CPU had a 0.5 MB vector/scalar cache. Vector Atomic Memory Operations (AMOs) were added to allow high performance on certain loops with potential memory conflicts. Another difference was the X2 used a fat-tree topology for the network. The Cray X2, like the X1, was globally addressable across the system. The proprietary router processed single dword or cache line requests at very high bisection bandwidth.

The Cray X2 was the latest and last proprietary vector computer designed at Cray Inc. However, many of ideas of vector processing apply to future versions of the x86 ISA and implementations and programming models of GPUs. In particular, vectorizing compiler

technology will continue to play a large role in delivering performance on future HPC systems.

Related Entries

- [NVIDIA GPU](#)
- [Pipelining](#)
- [Vector Extensions, Instruction-Set Architecture \(ISA\)](#)

Bibliographic Notes and Further Reading

Vector processing and related architectures preceded founding of Cray Research in 1972. The ILLIAC IV from Burroughs was an early SIMD computer. A control unit broadcasted an instruction to an Array Subsystem containing 64 processing units with local memory and mode bit control [1]. Two vector machines that appeared in the early 1970s were the Texas Instruments ASC [2] and the Control Data Corporation (CDC) STAR-100 [3]. However, it was the Cray-1, introduced in 1976, that combined a vector RISC ISA with technology improvements that made vector processing so dominant in the first 20 years of the modern HPC era. The Cray XMP and YMP design teams were led by Steve Chen [4].

Many academic investigators have studied vector architectures. James E. Smith, while at Cray Research in the early 1990s and at the University of Wisconsin-Madison, published extensively on vector processing [5, 6]. The Berkeley Intelligent RAM (IRAM) project included design of a processor-in-memory chip that sought to address the growing imbalance between fast processors and low memory bandwidth [7]. The chip took advantage of traditional strengths of vector processing, such as low power for issue and control, low design complexity, mature vector compiler technology, etc., and combined those with on-chip DRAM memory bandwidth. One target application area for such chips is on-demand media processing. The first edition of the classic text on computer architectures by Hennessy and Patterson [8] is recommended for a quantitative approach to computer architectures and comparisons between scalar and vector processing. Detailed performance models are presented extending from processor to memory system. The authors emphasize importance of scalar performance and low memory

latency leading to overall low start-up time for vector loops. Vector machines and SIMD machines share many features and can be programmed similarly. The book by Guy Blelloch gives in-depth examples of how algorithms with irregular memory access can be coded in vector/SIMD manner [9]. Studies of vectorizing compiler techniques, important for usability of vector systems, can be found in [10–13]. A detailed description of many advancements of the Cray X2 system is given in [14].

Bibliography

1. Siewiorek DP, Bell CG, Newell A (1982) Computer structures: principles and examples, McGraw-Hill, New York
2. Watson W (1972) The TI-ASC, A highly modular and flexible supercomputer architecture. In: Proceedings of the AFIPS, vol 41, pt I. AFIPS Press, Montvale, pp 221–228
3. Hintz RG, Tate DP (1972) Control data STAR-100 processor design. In: Proceedings of the Compcon 72, New York. IEEE Computer Society Conference, Washington DC, pp 1–4
4. Chen S (1983) Large-scale and high-speed multiprocessor system for scientific applications. In: Proceedings of the NATO advanced research work on high speed computing, Research Center, Jülich, West Germany (Also in Hwang K (ed) (1984) Supercomputers: design and applications. IEEE, Washington, DC)
5. Espasa R, Valero M, Smith JE (1998) Vector architectures, past, present, and future. In: International conference on supercomputing, Melbourne, Australia. ACM, New York
6. Smith JE, Faanes G, Sugumar R (2000) Vector instruction support for conditional loops. In: Proceedings of the 27th annual international symposium on computer architecture, June 2000, Vancouver, BC
7. Kozyrakis C, Gebis J, Martin D, Williams S, Mavroidis I, Pope S, Jones D, Patterson D, Yelick K (2000) Vector IRAM: a media-oriented vector processor with embedded DRAM. In: 12th hot chips conference, Palo Alto, CA, August 2000
8. Hennessy J, Patterson D (1990) Computer architecture, a quantitative approach, Morgan Kaufmann Publishers, San Francisco, CA
9. Blelloch GE (1990) Vector models for data-parallel computing, MIT Press, ISBN 026202313X
10. Callahan D, Dongarra J, Devine D (1988) Vectorizing compilers: a test suite and results. In: Supercomputing'88, Orlando, FL, November, 1988. ACM/IEEE, pp 98–105
11. Allen R, Kennedy K (1987) Automatic translation of FORTRAN programs to vector form. ACM Trans Program Lang Syst 9(4):491–542
12. Kuck D, Budnik PP, Chen S-C, Lawrie DH, Towle RA, Strebendt RE, Davis EW, Jr., Han J, Kraska PW, Muraoka Y (1974) Measurements of parallelism in ordinary FORTRAN programs. Computer 7(1):37–46
13. Padua D, Wolfe M (1986) Advanced compiler optimization for supercomputers. Commun ACM 29(12):1184–1201
14. Abts D, Bataineh A, Scott S, Faanes G, Schwarzmeier J, Lundberg E, Bye M, Schwoerer G (2007) The cray BlackWidow: a highly scalable vector multiprocessor. In: Best paper SC 07, Reno, Nevada, November 2007. IEEE, ACM, New York

Cray XMT

LARRY KAPLAN
Cray Inc., Seattle, WA, USA

Definition

The Cray XMT is a shared memory parallel computer consisting primarily of aggressively multi-threaded processors derived directly from the Tera Multi-Threaded Architecture (MTA) (see ►Tera MTA) [1]. It is packaged using the same infrastructure and support as the Cray XT3 (see ►Cray XT3 and Cray XT Series of Supercomputers). The main difference in hardware from the XT3 is the replacement of the socket 940 Opterons on the compute nodes with Cray Threadstorm processors. These processors implement the Multi-Threaded Architecture with support for 128 hardware threads in each processor. Systems of up to 512 Threadstorm processors are supported.

Discussion

Introduction

Shared memory parallel programming can provide simplicity to the programmer by allowing the placement of data to be ignored. All memory is essentially equally distant in such a system. However, implementing a shared memory system in a scalable fashion has several challenges. The biggest challenge is making a distributed memory implementation of the memory appear as uniformly shared to the programmer by hiding the latency to the memory as seen by the processors. In addition, several other important features are required to make best use of the shared memory environment including support for fine-grained synchronization. The Cray XMT implements these features using the custom Threadstorm processor and SeaStar2 interconnect.

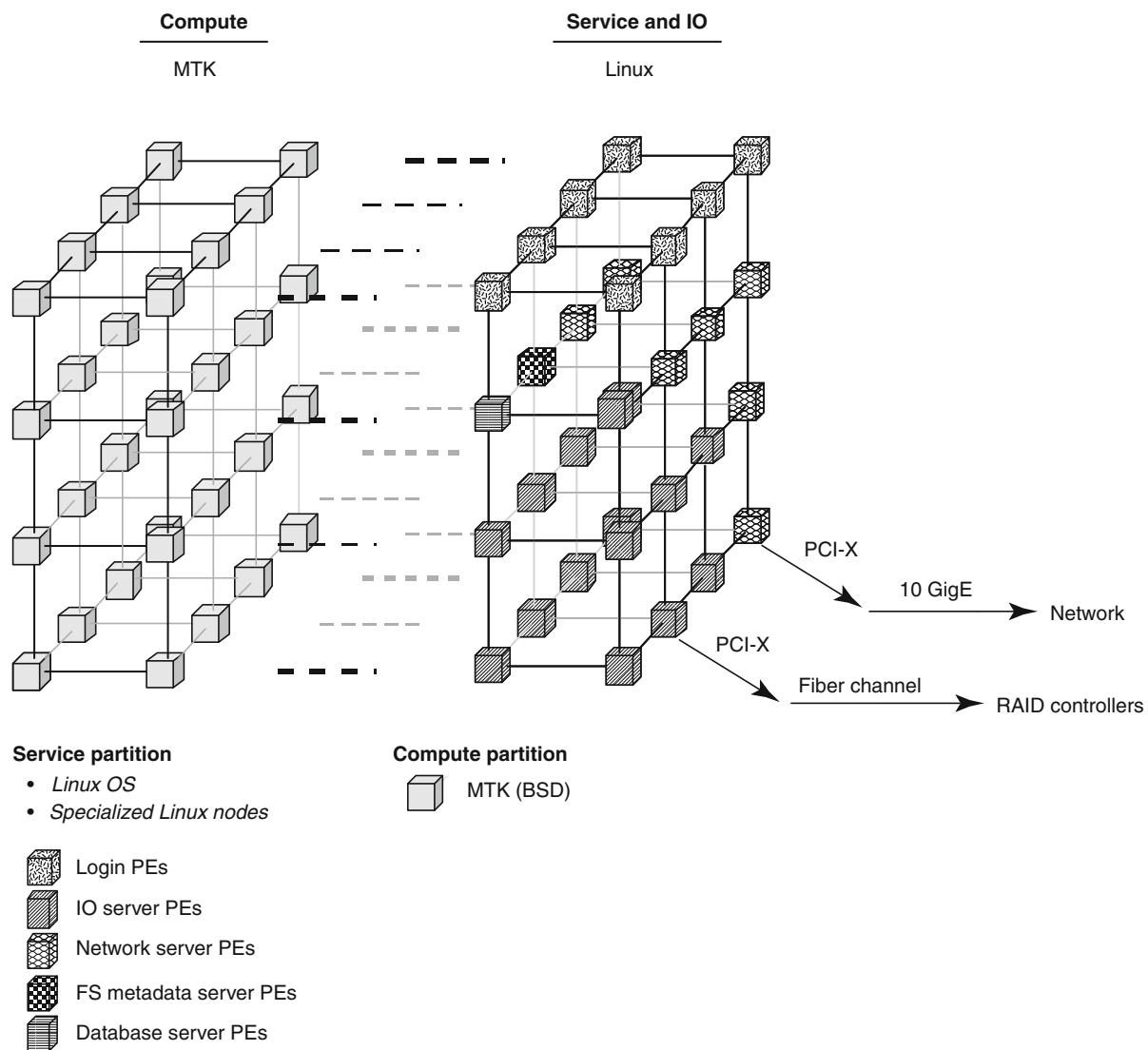
Note that in addition to the main Threadstorm multi-threaded processors on the XMT compute nodes, an XMT also contains service nodes that use standard AMD Opterons. The system hardware architecture is shown in Fig. 1.

These service nodes provide external connectivity via PCI-X cards, and a login environment with compilation and other programmer tools. These nodes can also be programmed using standard Linux tools. Except where explicitly noted, most of the discussion presented

here is specific to the multi-threaded compute portion of the system.

Threadstorm

The Threadstorm processor is a direct descendant of the MTA-2 processor. As with its predecessor, Threadstorm contains 128 hardware threads or *streams* multiplexed onto a single execution pipeline on a cycle-by-cycle basis. By context-switching every cycle, the execution latency of any individual instruction can be effectively



Cray XMT. Fig. 1 XMT architecture

reduced or hidden from the point of view of the processor. This is especially important for memory operations in instructions because the latencies for these operations can be relatively long in terms of processor clock cycles.

In addition, each stream within the processor, in concert with the compiler, implements a release consistent memory model [2] and can have up to eight memory references (e.g., loads and/or stores) outstanding simultaneously. This allows for a total of 1,024 memory references to be outstanding from a given Threadstorm processor.

Threadstorm also includes an integrated DDR1 memory controller, primarily due to the fact that it sits in an Opteron socket 940, which expects such functionality to be present. In addition, Threadstorm contains a HyperTransport (HT) interface over which it communicates with SeaStar, which implements the high-speed interconnect.

The processor and memory controller, which in Threadstorm is on the same die as the processor, supports fine-grained synchronization through the use of *state bits* stored in memory. Conceptually each 64-bit memory location supports four additional bits that together are used to implement various forms of synchronization. Unlike in the original MTA, some assumptions on the use of those bits are leveraged to allow the implementation to only have two extra bits per memory location. The directly supported forms of synchronization include mutual exclusion locks, single word producer-consumer, among others.

Because the processor is latency-tolerant, most memory used in the Cray XMT is distributed across all of the memory units that reside at every processor. A pseudo-random distribution is used to avoid any stride access conflicts or other memory reference patterns that might create hot memory units, assuming the references are not accessing the exact same memory word (or small set of words).

Some other differences between the MTA-2 processor implementation and Threadstorm include:

- Number of processors supported increased from 256 to 8,192
- Memory distribution changed from every word to every eight words

- Data TLB increased in size to cover 512 terabytes (was only 128 terabytes)
- Amount of memory supported per node increased from 4 gigabytes to 16 gigabytes

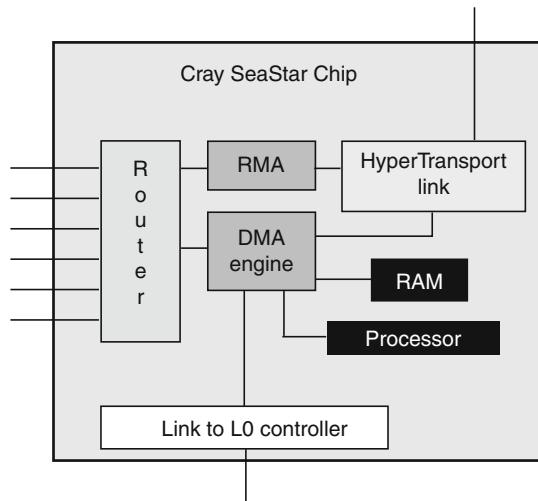
SeaStar

The Cray SeaStar 3-D Torus interconnect (see ►[Cray XT4 and Seastar 3-DTorus Interconnect](#)) was originally designed for use in the Cray Redstorm system that ultimately became the Cray XT3. It is primarily designed for message passing types of communication through the use of a direct memory access (DMA) engine and an embedded processor. In order to support the XMT, a *Remote Memory Access* (RMA) block was added that could process the memory reference style of communication used by Threadstorm. The resulting ASIC is known as SeaStar2 and is shown in Fig. 2.

RMA transactions are used exclusively between Threadstorm nodes. Message passing transactions are used to communicate with the Opteron-based service nodes. Portals [3] is used to drive the message-passing transactions though only a small subset of the Portals API is implemented on the compute nodes.

Programming Model

The Cray XMT inherits the majority of its programming model from the Tera MTA and presents a flat, shared memory to the programmer that is accessed



Cray XMT. Fig. 2 SeaStar2

using enhanced C and C++ languages via a compiler that provides automatic parallelization [4]. Fortran is not supported on the Cray XMT.

The Cray XMT programming model is supported by an advanced runtime library that is linked in with the application. Due to certain aspects of the hardware design, this runtime is able to assume various responsibilities normally associated with an operating system [5]. Such responsibilities include thread-level scheduling and exception handling. The MTA hardware allows for thread creation and destruction in user mode and delivers all exceptions directly to the privilege level in which they are raised. Events such as floating point and memory synchronization exceptions are delivered directly to user mode if they are raised there.

Various tools from the MTA have also been updated and repackaged for the Cray XMT. Debugging is provided by the mdb debugger that is based on gdb [6]. The compiler analysis tool Canal is available. Tracing is provided by Tview. Bprof provides block-level profiling. Cray Apprentice2 provides a graphical interface for viewing the output of these tools [7].

Operating System

The compute processors of the Cray XMT run an operating system called MTK, which is derived from the 4.4 Berkeley Software Distribution (BSD) of Unix with a custom microkernel. The microkernel handles most of the hardware-specific processor functionality, including memory allocation and process-level scheduling, while the BSD layer provides a familiar environment for applications. The operating system treats all of the XMT compute processors as a single instance with a Uniform Memory Access (UMA) model of memory. The main exception to this treatment is that instructions are replicated to every Threadstorm processor in order to make it easier to deliver the required instruction fetch bandwidth.

The service processors of the XMT are identical to those in the Cray XT3 and run a version of Linux derived from SuSE Linux Enterprise Server (SLES). Each service node runs its own instance of Linux.

Built on top of the low-level Portals communication protocol used between the compute and service nodes, the Lightweight User Communication (LUC) library is available for user programs to transfer data between the

service and compute nodes. TCP/IP is also supported between the two node types.

Infrastructure and Administration

As previously described, from a packaging standpoint, the Cray XMT is simply a Cray XT3 system where the Opterons on the compute nodes have been replaced by Threadstorm processors (and SeaStar2 interconnect ASICs are present). As such, all of the other infrastructure and support for XT3 systems, other than the XT3 compute node programming environment, applies to XMT. By using the XT3 infrastructure, the Cray XMT can leverage the reliability, manufacturability, and economies of scale that XT3 provides.

Cray XMT systems are administered in a very similar fashion to XT3. They have the same System Maintenance Workstation (SMW) and Hardware Supervisory System (HSS) as is used in XT3, with some extensions to specifically support the Threadstorm processor. As with XT3, users log into Opteron-based service nodes for access to the system and programming environment. Jobs are then launched from the service, or login, node and run on the compute nodes.

Target Applications

The Cray XMT excels at applications that operate on large volumes of unstructured data such that the data does not fit into the memory of a single node of a typical computer and that data is also not easy to partition for locality to multiple nodes of such a computer. Some application areas that have this type of data include social media analysis, power grid contingency analysis, high-throughput video analysis, and some forms of text document processing.

Follow-on Designs

The concepts contained in XMT and Threadstorm have been considered for some other designs.

Scorpio

As part of the research for the DARPA HPCS program, Cray investigated the design of a highly multi-threaded, multi-core processor called Scorpio that planned to include several of the features present in the Threadstorm processor. These features included the multiplexing of many hardware streams onto an execution

pipeline and the implementation of extra memory state bits on every 64-bit word of memory for fine-grained synchronization (though the bit definitions were somewhat different than with XMT). Scorpio has never been manufactured.

XMT2

A new design of the Threadstorm processor and XMT system is being investigated by Cray to address some of the current shortcomings of the XMT. The main areas of improvement being considered include:

- DDR2 memory controller (instead of DDR1)
- Larger per-node memory sizes
- Use of XT5 infrastructure (rather than XT3)

Related Entries

- ▶ [Cray XT3 and Cray XT Series of Supercomputers](#)
- ▶ [Cray XT4 and Seastar 3-D Torus Interconnect](#)
- ▶ [Multi-threaded Processors](#)
- ▶ [Tera MTA](#)

Bibliography

1. Alverson A et al (1990) The Tera computer system. In: Proceedings of the 4th international conference on supercomputing. ACM Press
2. Sarita V, Adve KG (1996) Shared memory consistency models: a tutorial. In: IEEE Comput, December 1996
3. Brightwell R et al (2002) Portals 3.0: protocol building blocks for low overhead communication. In: Proceedings of the international parallel and distributed processing symposium, IEEE 2002
4. Cray Inc (2009) Cray XMT™ programming environment user's guide, S-2479-14, 12/2009, <http://docs.cray.com/books/S-2479-14/S-2479-14.pdf>. Accessed December 2009
5. Alverson G et al (1995) Scheduling on the Tera MTA. In: Proceedings of the workshop on job scheduling strategies for parallel processing, Springer
6. Cray Inc (2009) Cray XMT™ debugger reference guide, S-2467-14. <http://docs.cray.com/books/S-2467-14/S-2467-14.pdf>. Accessed December 2009
7. Cray Inc (2009) Cray XMT™ performance tools user's guide, S-2462-14. <http://docs.cray.com/books/S-2462-14/S-2462-14.pdf>. Accessed December 2009
8. Cray Inc (2009) Cray XMT™ system overview, S-2466-14. <http://docs.cray.com/books/S-2466-14/S-2466-14.pdf>. Accessed December 2009

Cray XT Series

- ▶ [Cray XT3 and Cray XT Series of Supercomputers](#)
- ▶ [Cray XT4 and Seastar 3-D Torus Interconnect](#)

Cray XT3

- ▶ [Cray XT3 and Cray XT Series of Supercomputers](#)
- ▶ [Cray XT4 and Seastar 3-D Torus Interconnect](#)

Cray XT3 and Cray XT Series of Supercomputers

JEFF BROOKS¹, GERRY KIRSCHNER²

¹Cray Inc., St. Paul, MN, USA

²Cray Incorporated, St. Paul, MN, USA

Synonyms

Cray red storm; Cray SeaStar Interconnect; Cray XT series; Cray XT3; Cray XT4; Cray XT5; Cray XT6; MPP

Definition

The Cray XT3 supercomputer is a large-scale Massively Parallel Processing (MPP) supercomputer from Cray Inc. The design for the Cray XT3 is based on the Red Storm supercomputing system which was designed in cooperation with Sandia National Laboratory (Sandia). The system was announced in November of 2004. Subsequent products have included the Cray XT4, Cray XT5, and Cray XT6 supercomputers, each based on successive versions of AMD Opteron processor, but essentially retaining the basic Cray XT3 architecture. The Cray XT series of supercomputers has proven to be a very successful product line, selling over 1,500 cabinets in a 5-year period.

Discussion

In 2001, the U.S. Department of Energy's National Nuclear Security Administration (NNSA) set aside funding for Sandia to obtain a new high-end computational capability to address mission needs. Sandia sent

out a Request for Information; however, no existing architecture was able to satisfy the lab's scalability and cost requirements. Subsequently, a Request for Quotation was issued by Sandia. Two suppliers responded with proposals; however, neither was able to fully meet the requirements laid out in the proposed Statement of Work with existing or planned products.

The proposal from Cray indicated a willingness to custom engineer a supercomputer to meet Sandia's needs within the budgetary envelope which was approximately \$90M. The project would require a custom ASIC, custom packaging and cooling, a custom classified/unclassified interconnect switch (known as the "red-black" switch), a custom hardware supervisory system (HSS), and a custom software stack. Despite this, the Red Storm project was completed and delivered in only 30 months. Concurrently, with this initial delivery, Cray launched the Cray XT3 supercomputer in November of 2004 in Pittsburgh, PA, at the SC2004 conference.

Architecture

The Cray XT3 system is a massively parallel system consisting of two types of nodes, grouped into two partitions, respectively:

- Compute nodes run application programs. All compute nodes run a Cray XT3 light weight kernel.
- Service nodes handle support functions such as login management, I/O, and network management. All service nodes run a full Linux-based Cray XT3 operating system.

The nodes are tightly coupled with an interconnection network that supports fast MPI traffic, as well as a fast I/O to a global, shared-file system. The interconnection network is based on a 3-D torus topology that combines HyperTransport and proprietary protocols. The Cray SeaStar chip on each node functions as the router chip. It has seven bidirectional ports, six for the 3-D interconnection network and one for the HyperTransport link to the node's processor. [Figure 1](#) shows the basic architecture of a Cray XT3 system.

Each Cray XT3 system also includes a Cray RAS and Management System (CRMS). The CRMS includes a System Management Workstation (SMW), which functions as the administrator's single-point interface for

booting, monitoring, and managing Cray XT3 system components. The SMW communicates with all nodes and RAID disk controllers via a private Ethernet network.

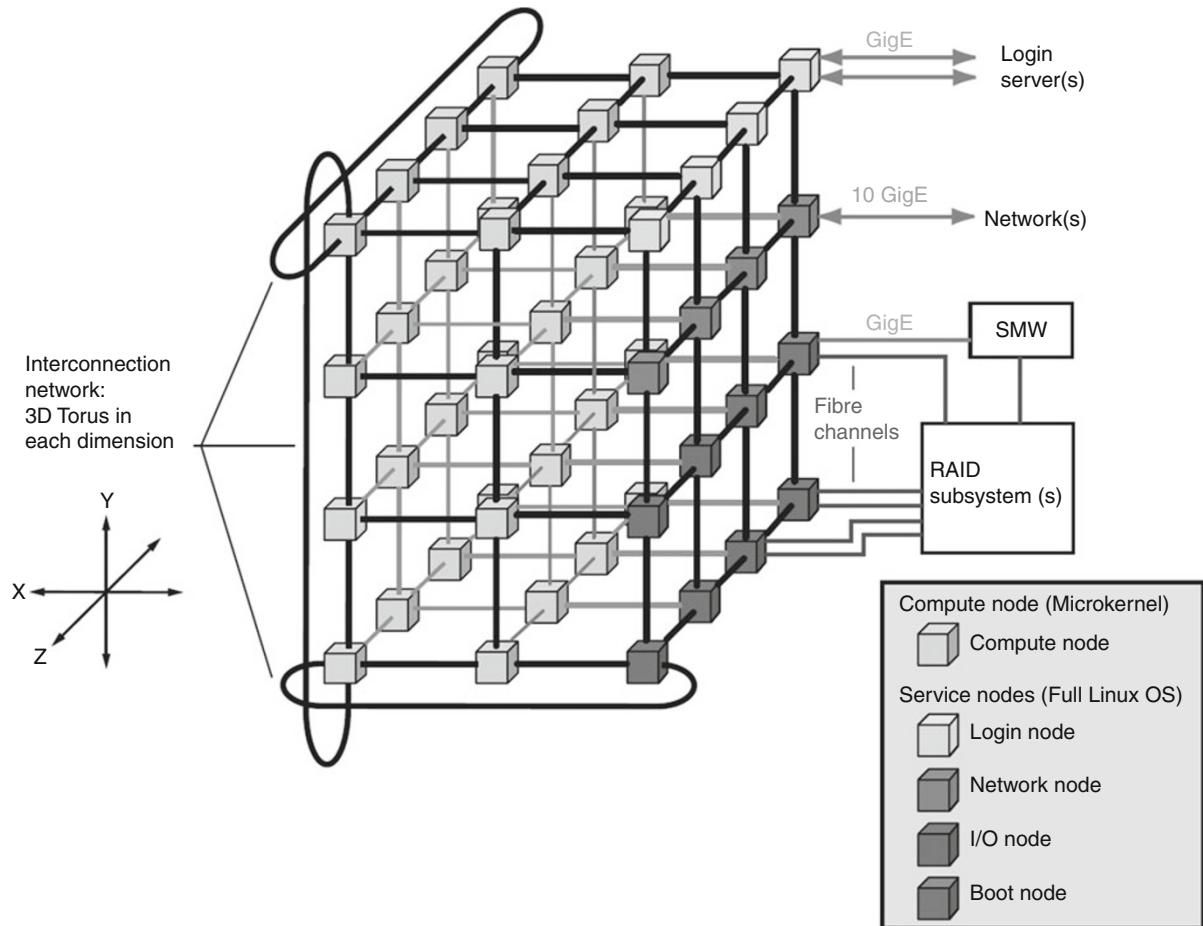
Processor

The AMD OpteronTM processor was selected for the Red Storm project. There were several reasons for this:

1. The processor featured an open, high-speed interface called HyperTransportTM. This would allow the processor to be tightly integrated with a custom high-performance interconnect chip. Because HyperTransport was an open interface, there was extensive third-party IP available. This would serve to reduce risk and development time for the interconnect ASIC.
2. The processor is *fully compatible* with the X86 processor architecture. This ensured compatibility with a vast quantity of existing software, including compilers, libraries, and applications. This would serve to reduce risk associated with operating system and application software.
3. The processor featured extensions to the X86 architecture specified by AMD allowing applications to use 64-bit addressing. This mode is called X86-64. Applications can be ported to 64-bit mode, yet full compatibility and performance is retained with 32-bit applications. Intel ultimately followed this convention, ensuring its market relevance for the long term.
4. The AMD Opteron pulled the function of the Northbridge chip onto the processor die itself. In particular, the memory controller is on the Opteron itself. This resulted in a very low-latency interface to memory (~51 ns) and a dedicated high-bandwidth interface for each processor (6.4 GB/s).
5. Since the Northbridge is in the Opteron itself, there is considerable savings in components (one integrated circuit per node) and power (approximately 11 W/node) since no separate Northbridge chips would be necessary. This served to reduce component count and increases reliability.

Node Memory

Each node has four slots for memory DIMMs. These slots provide 1–16 GB of local memory for the node.



Cray XT3 and Cray XT Series of Supercomputers. Fig. 1 Cray XT3 system architecture

Each DIMM has 72 bits: 64 data bits and 8 error-correcting code (ECC) bits.

Memory is protected using single-symbol correction/ double-symbol detection (SSC/DSD) enhanced ECC. This enhanced ECC can detect and correct single-symbol errors, or can detect two-symbol errors, but cannot correct them.

Compute Node

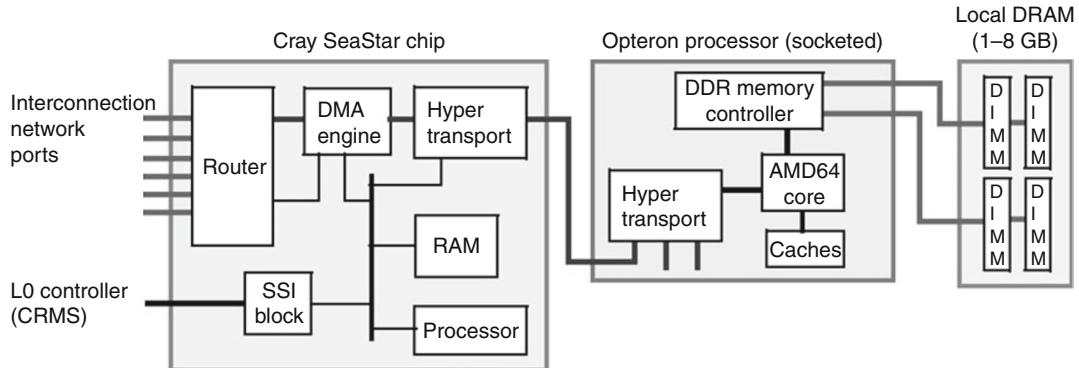
Figure 2 shows a block diagram of a Cray XT3 compute node. Each compute node consists of one processor socket populated with minimum of a 2.0 GHz single-core AMD Opteron processor, four DIMM slots which provide 1–8 GB local memory, a Cray SeaStar chip that connects the processor to the high-speed interconnection network, and an L0 controller (also referred to

as blade control processor) interface which is used for system management.

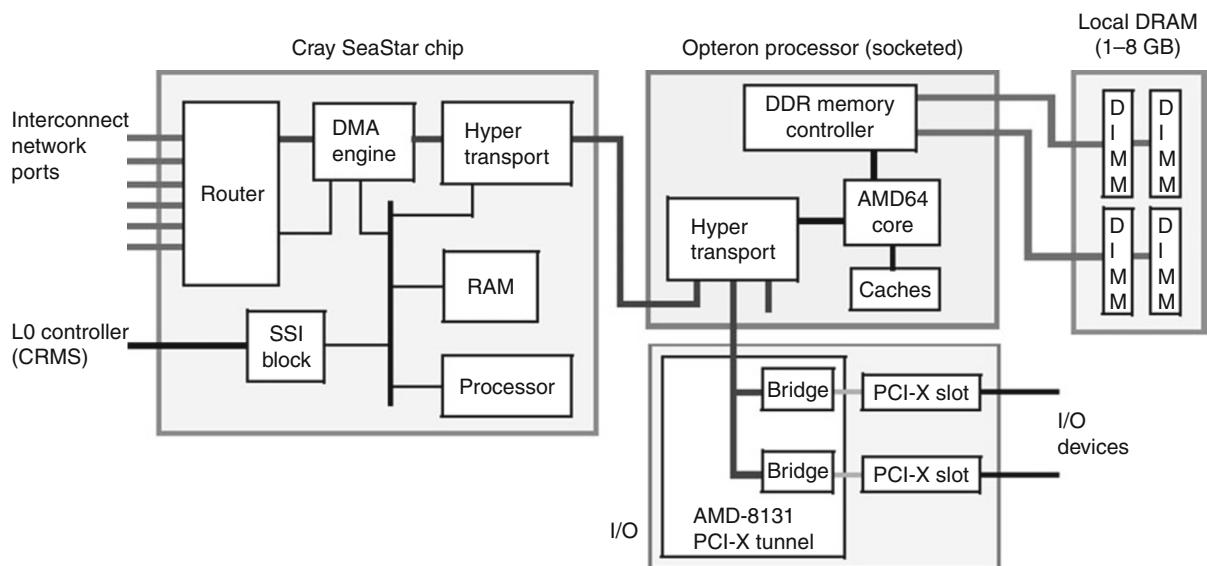
Service Nodes

Figure 3 shows the architecture of a service node. Service nodes use the same processors, memory, and SeaStar chips as compute nodes. In addition to these components, the service node has an AMD-8131 HyperTransport PCI-X tunnel chip that drives two PCI-X slots. Each slot is on its own independent 64-bit/133 MHz PCI-X bus. PCI-X cards plug into the PCI-X slots and interface to external I/O devices. Later versions of the Cray XT series implemented PCI-Express slots on these nodes.

Each Cray XT3 system includes several types of service nodes. Each type performs dedicated function



Cray XT3 and Cray XT Series of Supercomputers. Fig. 2 Compute node architecture



Cray XT3 and Cray XT Series of Supercomputers. Fig. 3 Service node architecture

and requires a minimum of one PCI-X card. The types include:

- *Login Node*: Users login into the system via login nodes. Each login node includes one or two single-port Gigabit Ethernet PCI-X card that connects to a user workstation. Copper and fiber cards are available.
- *Network Service Node*: Each Network service node contains one 10 Gigabit Ethernet PCI-X card that can be connected to customer network storage devices.
- *I/O Node*: Each I/O node contains one or two dual-port Fibre Channel (FC) Host Bus Adapter (HBA)

card. The Fibre Channel ports connect to the system's RAID storage.

- *Boot Node*: Each system requires one boot node. A boot node contains one FC HBA and one GigE PCI-X card. The FC HBA connects to the RAID and the GigE card connects to the System Management Workstation of the CRMS.

Interconnection Network

The Cray XT3 system uses its interconnection network to link nodes in a 3D torus topology and to facilitate the system's high-communication bandwidth. Physically, this network includes the Cray SeaStar routers (the heart of the network), router ports, and cables.

Key performance data for the interconnection network are:

- A bidirectional injection bandwidth of more than 2 GB/s per direction. This is the bandwidth from the processor core into the SeaStar router.
- A sustainable bandwidth of 6.5 GB/s for each of the six directions of the torus. The SeaStar router supports an aggregate bandwidth of almost 40 GB/s.
- An MPI latency of less than 7 μ s between a processor pair.

This Cray SeaStar network allows efficient MPI messaging and also high-sustainable bandwidth to the Object Storage Servers (OSS) driving the disks systems for the global Lustre filesystem.

A Cray SeaStar chip a HyperTransport connection to the node's AMD processor – its six ports connect the 3D torus. The X, Y, and Z dimensions use two ports each. The network uses a Cray proprietary protocol.

Network connections are made three ways: within the blade, on a backplane that the blades plug into, and with groups of network cables. The network cables can be configured in multiple ways to create various topologies based on system size.

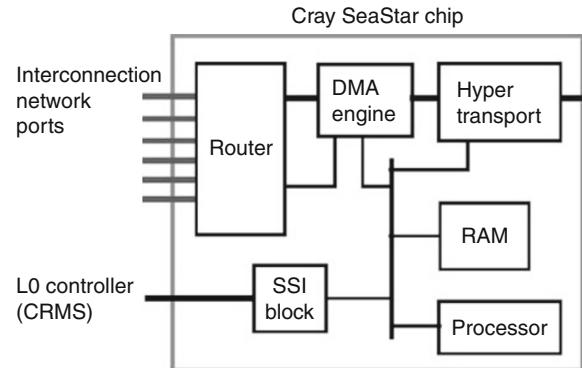
Configuration tables are used for data flow mapping across the interconnection network. These tables are loaded at system boot time. They can also be reloaded to reconfigure the interconnection network following a hardware failure.

Cray SeaStar Chip

There is one Cray SeaStar chip for each Opteron processor in the system. The chip connects the Opteron processor to the 3D interconnection network. It offloads all send/receive work so that compute work is not interrupted. Here is a block diagram of the SeaStar chip ([Fig. 4](#)):

The SeaStar chip is a system-on-chip design that combines third-party hardware components and interface definitions with Cray custom logic and functionality. It contains the following major blocks:

- A HyperTransport interface provides a bidirectional link between the Opteron processor and the SeaStar chip. The HyperTransport Link has a maximum data



Cray XT3 and Cray XT Series of Supercomputers.

Fig. 4 Cray SeaStar chip

transfer rate of 3.2 GB/s in each direction between the Opteron processor and the SeaStar chip.

- A custom router that enables the Opteron to communicate with other nodes via the interconnection network. The router has six ports that connect to the interconnection network. The peak bidirectional bandwidth of each port is 7.6 GB/s, with a measured sustained bandwidth of 6.5 GB/s.
- A communication interface that consists of a Direct Memory Access (DMA) engine, on-chip processor, and on-chip RAM connected to a local bus. These components function as a message processor that route send/receive data packets to their appropriate destination.
- A synchronous serial interface (SSI) that connects the module controller (known as the L0) to the internal SeaStar local bus. This connection allows the L0 controller access to the Opteron memory and status registers. The L0 controller is part of the CRMS, which is used for booting, maintenance, and monitoring the Cray XT3 system.

In order to achieve the lowest possible system latency, it is necessary to provide a path directly from the application to the communication hardware without the time-consuming traps and interrupts associated with traversing a protected operating system kernel. The SeaStar chip uses the Portals interface to provide this path.

More information about the Cray SeaStar interconnection network is available in the entry on [▶Cray XT4 and Seastar 3-D Torus Interconnect](#).

Packaging

Cray XT3 Compute Cabinet

Each compute cabinet contains three module chassis as follows:

- Each chassis is populated with any combination of eight blades (compute or service).
 - A compute blade contains four compute nodes, including providing four Opteron processors, up to 16 GB of DIMM memory per processor, and four SeaStar chips. A single compute cabinet can hold up to 96 compute nodes.
 - A service blade consists of two service nodes, including two Opteron processors and 2–16 GB of memory per blade, depending on the functions of the service nodes. The blade has four SeaStar chips to allow for a common board design and to simplify primary interconnect configurations. Some service blades also contain PCI-X slots for external connections. A single compute cabinet can hold up to 48 service nodes.
- The cabinet contains all power and cooling equipment for these blades; no external equipment is required.
- Each cabinet is air-cooled. A single blower assembly, located toward the front, below the cages, air cools all components within the cabinet. The blower pulls underfloor air into the cabinet and forces air vertically through the three chassis. Warm air exhausts through the top of the cabinet. A blower speed controller varies the speed of the blower to maintain the correct air pressure and temperature.

Ethernet switches and the cabinet controller (known as the L1 controller) are located at the rear of the cabinet. These components are part of the CRMS.

The following figure shows the component locations of a compute cabinet. A single mechanical assembly contains three identical chassis that house blades. Blades plug into the front of the chassis. Each chassis has a backplane assembly at the back ([Fig. 5](#)).

Software

Cray XT3 system software is based on software from the Sandia Red Storm project, plus the addition of commercially available tools and features. This creates a system

that, to the application programmer, is an optimized parallel processing machine and, to the system administrator, features a single boot image and a single root file system.

The software allows programmers to optimize applications that have fine-grain synchronization requirements, large-scale processor counts, and significant communication requirements.

Cray XT3 system software falls into four major areas:

- Operating system
- File system
- Programming Environment
- RAS Management System software

Cray XT3 Operating Systems

The Cray XT3 operating system functions fall into three categories, identified by the role of the components on which they run:

1. Service nodes run the full Linux Cray XT3 operating system.
2. Compute nodes run a Light Weight Kernel called *Catamount*.
3. CRMS components (the L0 and L1 controllers) run an embedded real-time Linux kernel.

SuSE Linux Operating System

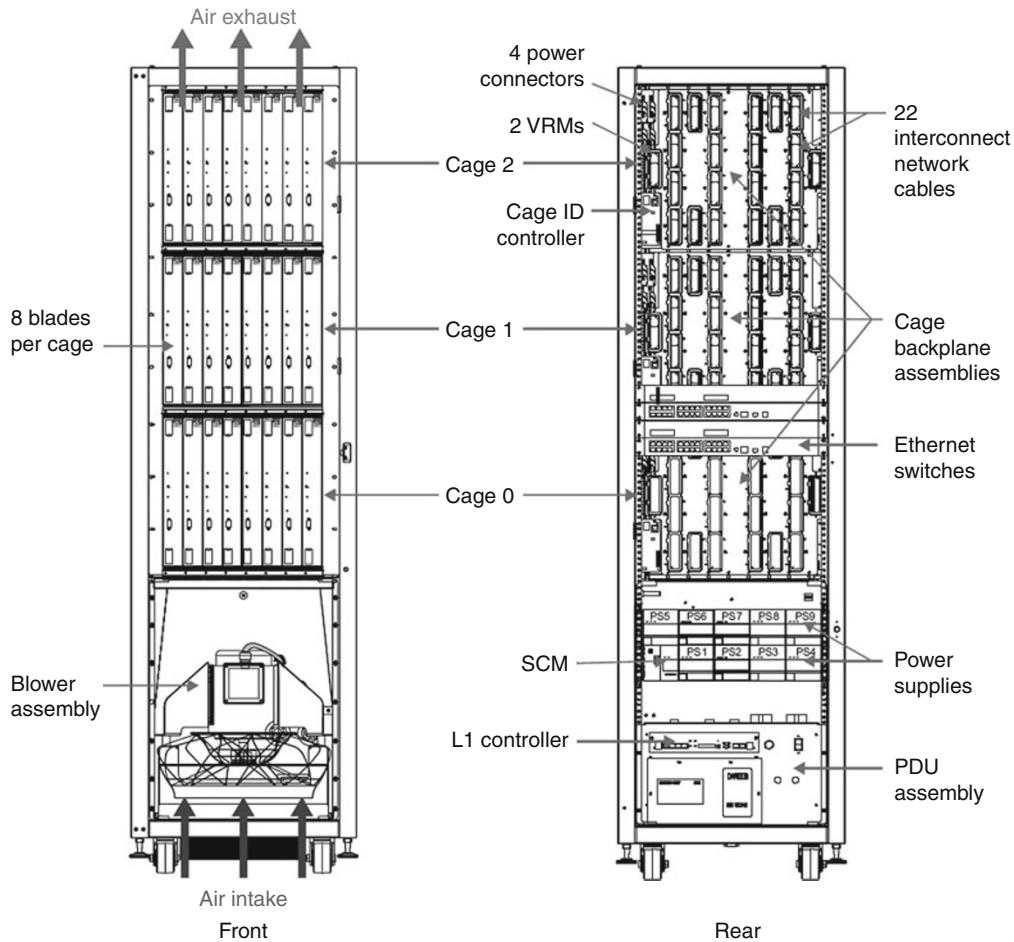
Cray XT3 system service nodes run a full, Linux-operating system based upon SuSE Enterprise Server 8.2. This was later upgraded to SuSE Enterprise Server release 9.0 in 3Q2005 ([Fig. 6](#)).

The operating system supports six types of service nodes:

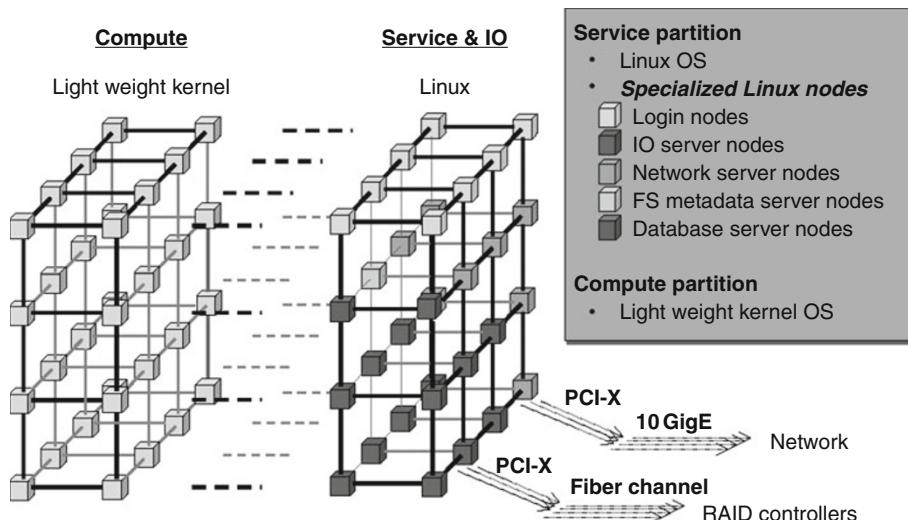
- Login Nodes
- I/O Server Nodes
- Network Server Nodes
- File System metadata Server Nodes
- Database Server (Admin) Nodes
- Boot Nodes

SuSE Linux Overview

With SuSE Linux, the Cray XT3 system provides users with all standard Linux interfaces and working environments. SuSE Linux was selected for the Cray XT3 operating system for a number of reasons:



Cray XT3 and Cray XT Series of Supercomputers. Fig. 5 Compute cabinet



Cray XT3 and Cray XT Series of Supercomputers. Fig. 6 Node functions



- *Open Source.* The availability of source code for the operating system allows it to be customized. This is not possible with proprietary operating systems.
- *Widely Supported.* Several vendors provide Linux support, in addition to the legion of programmers who are constantly enhancing the operating system.
- *Familiar to Users.* Many HPC users develop codes on Linux desktops. Most users are more comfortable with this familiar operating system than they would be with a proprietary solution.
- *64-bit aware.* At the time, it was chosen for the Cray XT3 system, SuSE was one of few companies actively targeting the 64-bit market place. SuSE software was already ported to the AMD Opteron architecture, so its close-working relationship with AMD made it the obvious choice.
- *Cooperation.* SuSE worked with Cray to further the development of SuSE Linux into the High-Performance Market Place.

Single System View (SSV)

The Cray XT3 system presents a Single System View (SSV) to users and system administrators. The following list of SSV features is provided:

- *Single point of Administration:* An administrator can perform system administration tasks from the system boot node, without needing to log into additional nodes. Any administrative file in the machine can be modified from the root node.
- *Global file space:* User home directories and data directories are visible across the entire machine and may be accessed from anywhere using the same path.
- *Single user process space:* All user processes on the system are visible from any node. Each has a unique PID and can be signaled from any node.

Lightweight Kernel (LWK)

Because the Cray XT3 system was designed to scale to very large processor counts, it runs a computational environment that has very little system overhead, using a lightweight kernel (LWK) named *Catamount*. This LWK, developed jointly with Sandia National Laboratories, is a small, low-complexity OS that manages access to the physical node resources and infrastructure

for application execution, yet keeps system overhead at a minimum. It provides virtual memory addressing, physical memory allocation, memory protection, access to a high-performance message-passing layer, and a scalable job loader. While parallel jobs retain an agent (*yod*) on the login node that launched them, actual computation runs under the LWK on the compute nodes. And, as compute regions do not need to run standard Linux daemons, all standard requests are executed from the service partition. This significantly reduces to the LWK, and more compute cycles are reserved to the user. The ultimate benefit is that parallel jobs run in shorter times.

Some system calls are handled by the LWK, others are forwarded either to the *yod* agent running on the login node, or directly to I/O nodes for services that require a high degree of concurrency. The Catamount LWK is made up of two parts: the quintessential kernel (QK) and the Process Control Thread (PCT).

QK – The Quintessential Kernel

The QK provides an operating system kernel for the compute nodes that is resilient and scalable with minimal system overhead.

Resilient: Its simple design provides only essential functions required to run applications, much less than a full OS. It has been designed, tested, and evolved over two generations of MPPs.

Scalable: The QK design eliminates synchronization or communication between QK's on different nodes. This independence allows the system to scale to thousands of processors. Performance is achieved by using a simple and minimal system protocol that minimizes overhead for invoking kernel functions. Except for PCT, there are no processes or daemons that can interrupt programs and degrade performance of tightly coupled applications.

Process Control Thread (PCT)

On top of QK, a special user-level process, the Process Control Thread (PCT), performs services on behalf of application processes. Its primary responsibility is to start user applications, track them, schedule them, and relay completion and signal information between the application and *yod*.

LWK and Message Passing

The native messaging protocol for the Cray XT3 system is Portals version 3.3. It is a low-latency, low-overhead protocol, ideal for scalable, high-performance network communications. It is connectionless (i.e., it does not stay connected across consecutive communications), so the amount of system RAM memory used to provide buffers for message passing is independent of both the number of compute nodes in the system and the number of compute nodes assigned to a particular job.

The LWK is designed to support a high-performance MPI implementation. It lays applications out in memory in a linear fashion so that virtual memory can be mapped to real memory in a relatively simple offset scheme. As a result, Portals entries can span large chunks of real memory, and it is possible to map *all* memory using the available Portals table entries in the Cray SeaStar chip.

Kernels on the compute nodes and on the service nodes implement the Portals communication primitives. The Cray Seastar chip includes firmware that offloads the Portals message-passing overhead from the Opteron processor. At the user level, application processes communicate with one another by linking libraries that support the Portals interface.

Programming Environment

The Cray XT3 Programming Environment provides comprehensive support for 64-bit application development under an MPI model. It includes the following:

- Compilers
- Application libraries
- Application and system state monitors
- Application launch utilities
- Debugger
- Performance API
- Modules utility

Single Copies of Tools

On the Cray XT3 System, a single instance of each programming environment tool (compilers, debuggers, and so on) can be accessed from any login nodes on the system as if it were loaded on a disk connected to that node. Likewise, the Lustre file system provides a consistent view from anywhere within the system so that any

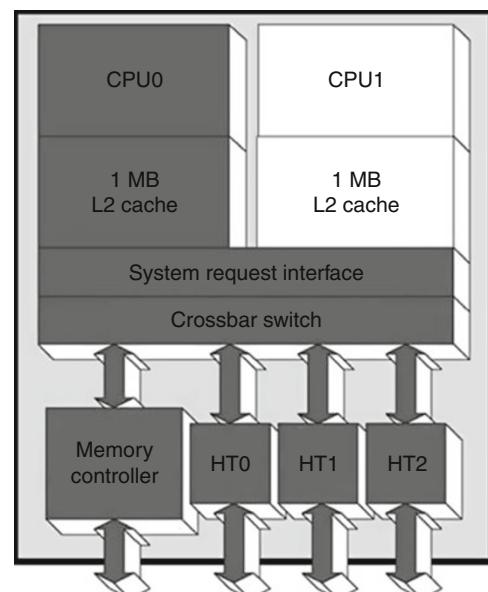
files that are required for program execution will have an identical path name no matter where the job is run.

Product Enhancements

XT3 Dual Core

The peak performance of Cray XT3 systems was significantly increased with the introduction of AMD Opteron dual-core processors. The AMD Opteron dual-core design increases application performance without having to redesign the socket or board.

[Figure 7](#) depicts the processor die with two CPU cores, each core having 1 MB L2 cache. The caches in some sense are shared in case one CPU sees a cache miss for its associated cache. The data however are available in the other CPU's cache, so it is directly loaded from there through the System Request Interface (SRI). The AMD Opteron was designed to add a second core, with a port already existing on the crossbar/SRI. The dual-core drops into existing AMD Opteron AM2 sockets, thus also enabling upgrades of single-core systems. The two CPU cores share the same memory and HT resources found in single core. The integrated memory controller and HT links route out the same as today's



Cray XT3 and Cray XT Series of Supercomputers.

Fig. 7 Interface for second processor in the AMD opteron dual core design



implementation. Almost all of Cray's initial single-core Cray XT3 customers took advantage of this upgrade.

Cray Linux Environment

The original operating system for the Cray XT3 was based on SuSE Linux for the Service Nodes and Catamount for the Compute Nodes. With quad-core processors coming, Cray needed a more full-featured OS on the compute node but needed to retain the scalability properties of Catamount. Cray transitioned to an operating system for Cray XT systems called the Cray Linux Environment (CLE) that is Linux based throughout.

Cray Linux Environment (CLE) uses a lightweight kernel operating system for compute nodes. The lightweight kernel includes a runtime environment based on the SUSE SLES distribution. It provides outstanding performance and scaling characteristics, matching the performance seen on the Catamount software stack.

CLE on Compute Nodes offers the full functionality of a Linux kernel, including:

- POSIX system calls (as supported by SUSE Linux kernel)
- Programming models, including OpenMP, MPI (version 2.0, based on MPICH), Cray SHMEM, and CAF. (*Note: Other programming models work as well. These include Global Arrays (the communication layer used by NWChem) and Charm++ (used by NAMD). Cray does not distribute this software but interacts closely with its developers (PNNL and the University of Illinois respectively) to make sure they work on the Cray XT systems.*)
- Application networking (Sockets)
- POSIX threads

PCI-Express SIO Blades

To improve the performance of Service and I/O Nodes (SIO) on Cray XT systems Cray transitioned from PCI-X to PCIe interfaces. The following figure shows the architecture of a Cray XT dual-core service node. Service nodes use an AMD Opteron dual-core processor, but the same DDR memory and interconnect SeaStar2 processors as compute nodes. In addition to these components, the service node has Broadcom HT-2100 HyperTransport PCIe tunnel chip that drives two PCIe slots.

The Broadcom HT-2100 modular bay on which the tunnel chip is mounted provides two PCIe slots for cards to interface to external I/O devices:

- a 16x PCIe slot provides 4 GB/s full duplex communication
- an 8x PCIe slot provides 2 GB/s full duplex communication

Each PCIe slot has independent buffers and shares (fairly) the bandwidth back to the attached Opteron. The HT-2100 HyperTransport PCIe tunnel chip is connected to the dual-core Opteron via a 4-GB/s HT2 link (Fig. 8).

Follow on Products

Cray XT4

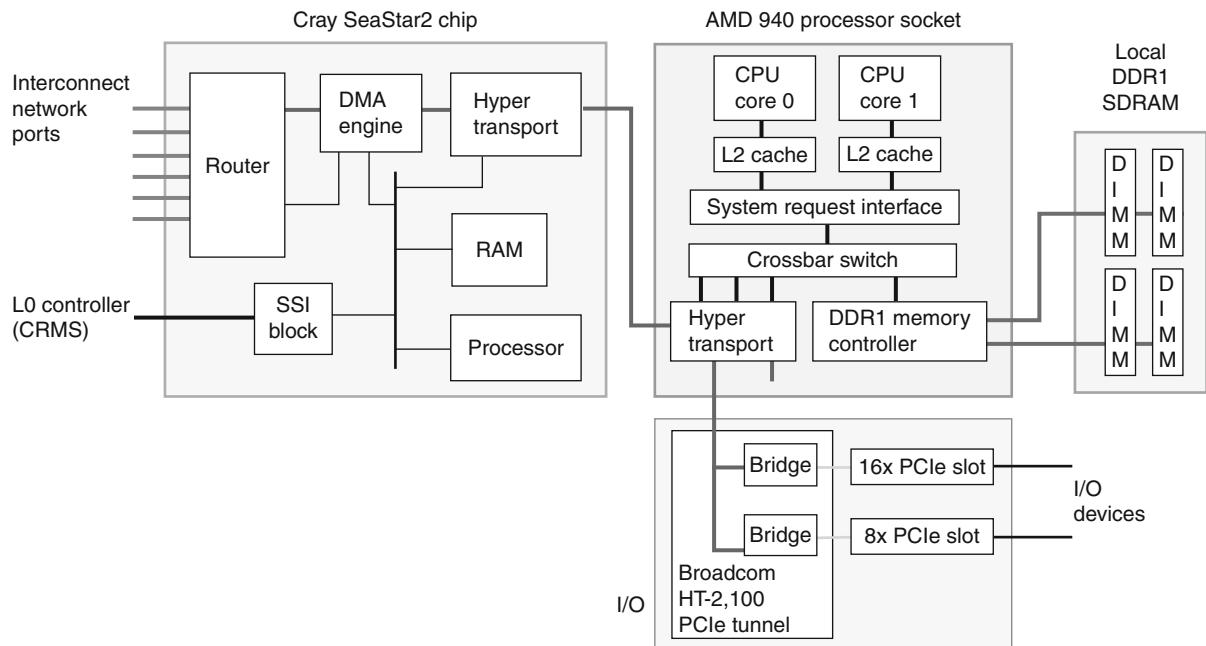
The Cray XT4 represented the second system in the Cray XT line of supercomputers. A new compute blade was designed to accommodate the new processors from AMD and the rest of the Cray XT3 system (service blades, boot infrastructure, software stack, cabinets, backplanes, power supplies, etc.) was retained. The network router chip, SeaStar, was enhanced and renamed SeaStar2. The Cray XT4 provided a significant performance increase over the Cray XT3 system in the same floor space. Initial Cray XT4 systems shipped with dual core AMD Opteron processors and DDR2 memory.

Later enhancements of the Cray XT4 compute nodes include the use of Quad-Core AMD Opteron processors. These processors were also capable of four floating-point results per cycle per processor core which provided a large increase in peak performance. Service and I/O nodes (SIO) were upgraded to dual-core AMD Opteron processors. Cray XT3 customers could upgrade to the Cray XT4 system by simply replacing compute blades.

The largest Cray XT4 system installed is “Franklin,” a 102-cabinet system installed at the National Energy Research Scientific Computing center (NERSC).

Cray XT5

The Cray XT5 provided another significant improvement in performance over the previous generation Cray XT4. The processor technology for the Cray XT5 included the quad-core “Barcelona” and “Shanghai” processors from AMD, and also the Six-Core “Istanbul”



Cray XT3 and Cray XT Series of Supercomputers. Fig. 8 Cray XT service node architecture (PCIe interface)

processor. A significant change from the Cray XT3 and Cray XT4, the Cray XT5 system packed four dual-socket nodes on a single compute blade. The form-factor for the compute blade was retained resulting in an overall doubling of compute density. Each system cabinet contains up to 192 AMD Opteron processors.

The system cabinet was enhanced to accommodate the increase in power and density. The backplane was designed to handle higher current and an enhanced power supply and blower system was designed to power and cool the system.

The compute node on the Cray XT5 system is either 8 or 12-core, depending on the Opteron used. The interconnect router chip was also improved for the Cray XT5. The new SeaStar2+ chip yields higher bandwidth and lower latencies than previous generation SeaStar and SeaStar2 routers.

The service blades and service infrastructure of the Cray XT5 was retained from the Cray XT4 system.

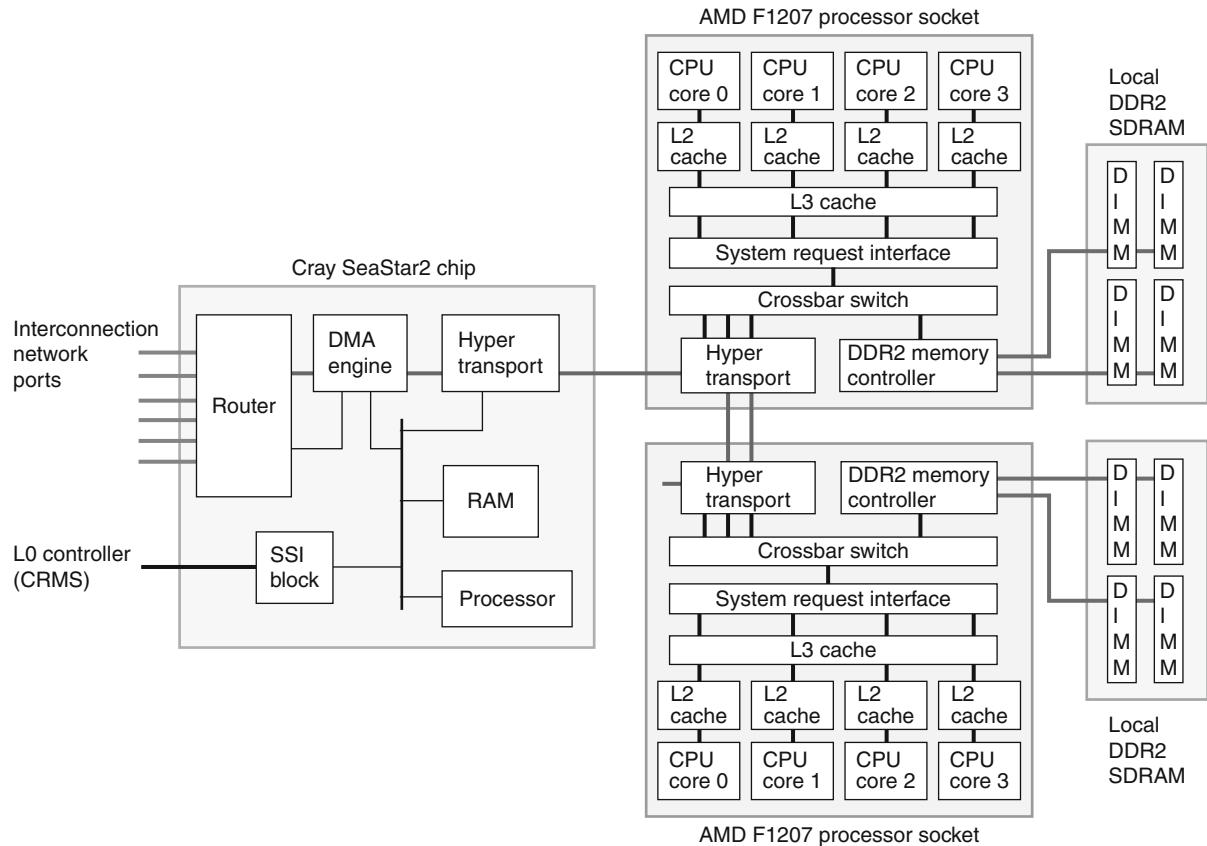
The following figure shows a block diagram of a Cray XT5 compute node. Each compute node consists of two processor sockets populated with AMD Opteron quad-core (or six-core) processors, eight DIMM slots, a Cray SeaStar2 ASIC that connects one processor to the high-speed interconnection network, and an L0 controller

interface used for system management and coherent HyperTransport connections between the two Opteron sockets. With the quad-core sockets, this provides an eight-way NUMA shared memory node (Fig. 9).

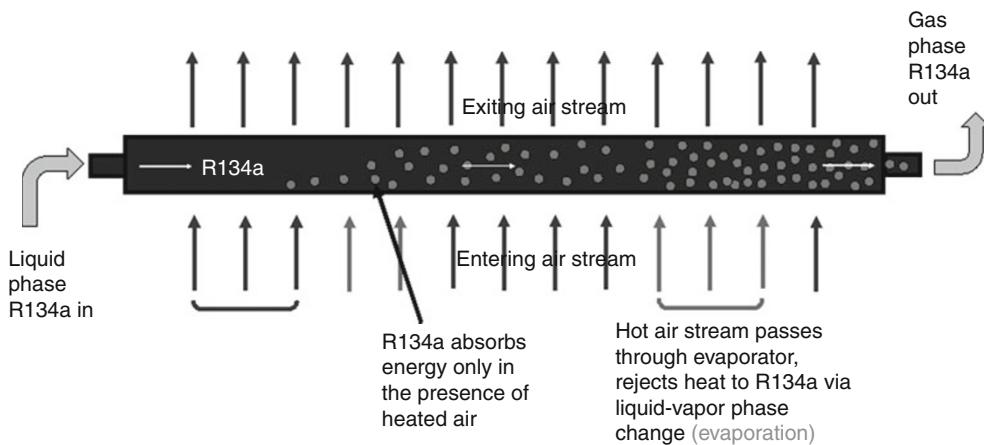
Cray ECOphlex Cooling

The Cray XT5 also introduced a new phase-change heat-removal system known as “Cray ECOphlex” cooling. Unlike traditional water cooling solutions available in the market, the heat generated by the compute blades is rejected to a fluid refrigerant – the R134a – via a liquid-vapor phase change (evaporation) as illustrated in the figure below. The bottom-to-top airflow on the Cray XT5 results in a limited surface area that can be used for heat removal at the top of the system cabinets. Since phase-change cooling is about an order-of-magnitude more effective per unit area compared with traditional water coils, it represented an ideal match for the Cray XT5 system (Fig. 10).

The Cray ECOphlex system includes one or more Heat Exchange Units (HEU, also known as XDP). This HEU is connected to the building water circuit, and its purpose is to recondense the R134a which runs in a closed loop between the evaporators in each system cabinet and the condenser coil in the HEU. A single



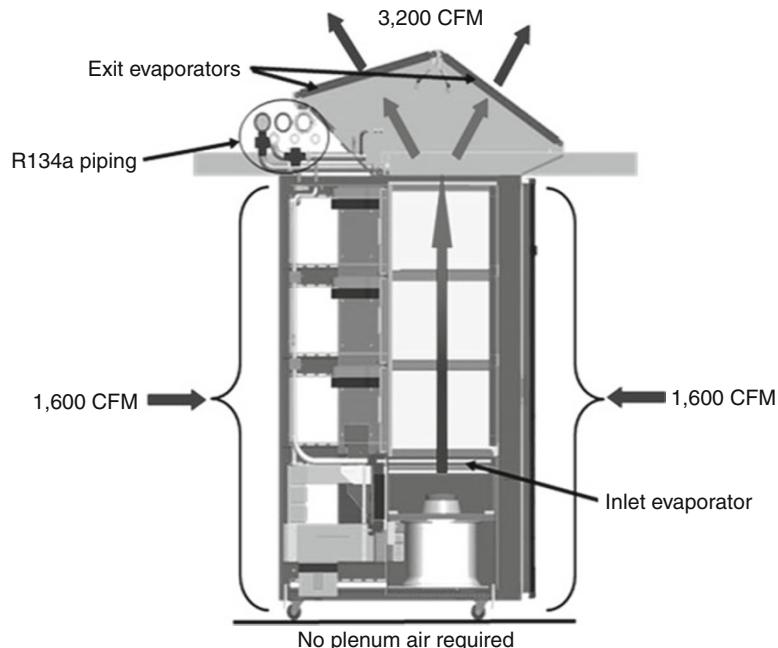
Cray XT3 and Cray XT Series of Supercomputers. Fig. 9 Cray XT5 compute node architecture (2 processors)



Cray XT3 and Cray XT Series of Supercomputers. Fig. 10 Liquid-vapor phase change

blower located in the bottom of each system cabinet is used to move air over the system blades. The blower pulls computer room air into the cabinet and forces air through the three chassis and ultimately through the

R134a evaporators. A blower speed controller varies the speed of the blower to maintain the correct air pressure and temperature and control logic within the XDP ensures that R134a is never below the dew point.



Cray XT3 and Cray XT Series of Supercomputers. Fig. 11 Schematic of Cray XT5 cabinet with ECOphlex cooling

Additionally, this liquid cooling allows customers to potentially dramatically reduce yearly cooling costs by leveraging the “Free Cooling” concept. The concept of “Free Cooling” is taking advantage of cool outdoor air to help save energy in data center chilled-water cooling systems.

In a data center facility, a typical mechanically generated chilled water system consists of the following equipment that uses electrical energy:

- Water chiller
- Computer room air-conditioning units
- Chilled water pumps
- Condenser water pumps
- Cooling tower fans

The water chiller is by far the biggest energy user (approximately 60%) in a chilled water system. If the water chiller’s compressor(s) can be shut down during cool weather, the outside ambient temperature can be used to help save energy in the chilled water system. The Cray XDP cooling system can operate with a higher chilled water set point temperature than a typical computer room air-conditioning unit. Thus, it is possible to bypass the chiller more with ECOphlex cooling compared to traditional air-cooling.

A traditional approach to free cooling is an indoor, water-cooled chiller connected to an outdoor closed-loop cooling tower. Typically, automatic valves and crossover piping are employed to bypass the chiller during when outside conditions are suitable.

An additional benefit of free cooling is extending the useful life of the water chiller by reducing its operating hours (Fig. 11).

The largest Cray XT5 system delivered is the 200 cabinet system at Oak Ridge National Laboratory, known as “Jaguar.” Jaguar holds the distinction of being the fastest system in the world as of November of 2009 (Fig. 12).

Cray XT6

Cray introduced the Cray XT6 in March of 2010. The Cray XT6 provided several major enhancements over the previous generation Cray XT5.

1. The Cray XT6 Compute Node uses two socket G34 (45 nm technology) AMD “Magny Cours” processors with eight or twelve cores per socket. Compute nodes on the Cray XT6 use two processors and hence have 16 or 24 cores per node.



Cray XT3 and Cray XT Series of Supercomputers. Fig. 12 200 cabinet ORNL Jaguar system with Cray ECOphlex cooling

2. The Cray XT6 Compute Node memory uses DDR3 technology rather than the DDR2 memory of previous generation Cray XT systems.
3. A new system cabinet was developed (the Series-6 cabinet). Enhancements included a more efficient ECOphlex evaporator design, a new system blower, and an enhanced power distribution system.
4. The Cray Linux Environment 3 (CLE3). This version of the system software includes a “Cluster Compatibility Mode” which allows users to install and run Independent Software Vendor (ISV) packages with no changes. CLE3 retained the “Extreme Scalable Mode” as the default OS image for native applications. A user can select either partition at job submittal time.

Shipments of the Cray XT6 started in the first half of 2010. At the time of this writing, the largest Cray XT6 system is installed at HECToR, the United Kingdom’s National Supercomputer Service. This system has over 44,000 processor cores.

Cray XT4

- [Cray XT3 and Cray XT Series of Supercomputers](#)
- [Cray XT4 and Seastar 3-D Torus Interconnect](#)

Cray XT4 and Seastar 3-D Torus Interconnect

DENNIS ABTS
Google Inc., Madison, WI, USA

Synonyms

[Cray red storm](#); [Cray XT series](#); [Cray XT3](#); [Cray XT4](#); [Cray XT5](#); [Cray XT6](#); [Interconnection network](#); [Network architecture](#)

Definition

The Cray XT4 system is a distributed memory multi-processor combining an aggressive superscalar processor (AMD64) with a bandwidth-rich 3-D torus interconnection network that scales up to 32K processing nodes. This chapter provides an overview of the Cray XT4 system architecture and a detailed discussion of its interconnection network.

Discussion

The physical sciences are increasingly turning toward computational techniques as an alternative to the traditional “wet lab” or destructive testing environments for experimentation. In particular, computational sciences can be used to *scale* far beyond that of traditional experimental methodologies; opening the door to large-scale climatology and molecular dynamics, for example, which encompass enough detail to accurately

model the dominant terms that characterize the physical phenomena being studied [2]. These large-scale applications require careful orchestration among cooperating processors to ply these computational techniques effectively.

The genesis of the Cray XT4 system was the collaborative design and deployment of the Sandia “Red Storm” computer that provided the computational power necessary to assure safeguards under the nuclear Stockpile Stewardship Program which seeks to maintain and verify a nuclear weapons arsenal without the use of testing. It was later renamed the Cray XT3 and sold commercially in configurations varying from hundreds of processors, to tens of thousands of processors. An improved processor, faster processor–network interface, along with further optimizations to the software stack and migrating to a lightweight Linux kernel prompted the introduction of the Cray XT4; however, the underlying system architecture and interconnection network remained unchanged.

System Overview

The Cray XT4 system scales up to 32k nodes using a bidirectional 3-D torus interconnection network. Each *node* in the system consists of an AMD64 superscalar processor connected to a Cray SeaStar chip [5] (Fig. 1) which provides the processor–network interface, and six-ported router for interconnecting the nodes. The system supports an efficient distributed memory message passing programming model. The underlying message transport is handled by the Portals [3] messaging interface.

This chapter focuses on the Cray XT interconnection network that has several key features that set it apart from other networks:

- Scales up to 32K network endpoints
- High injection bandwidth using HyperTransport (HT) links directly to the network interface
- Reliable link-level packet delivery
- Multiple virtual channels for both deadlock avoidance and performance isolation
- Age-based arbitration to provide fair access to network resources

Subsequent sections cover these topics in more detail.

There are two types of nodes in the Cray XT system. Endpoints (nodes) in the system are either

compute or system and IO (SIO) nodes. SIO nodes are where users login to the system and compile/launch applications.

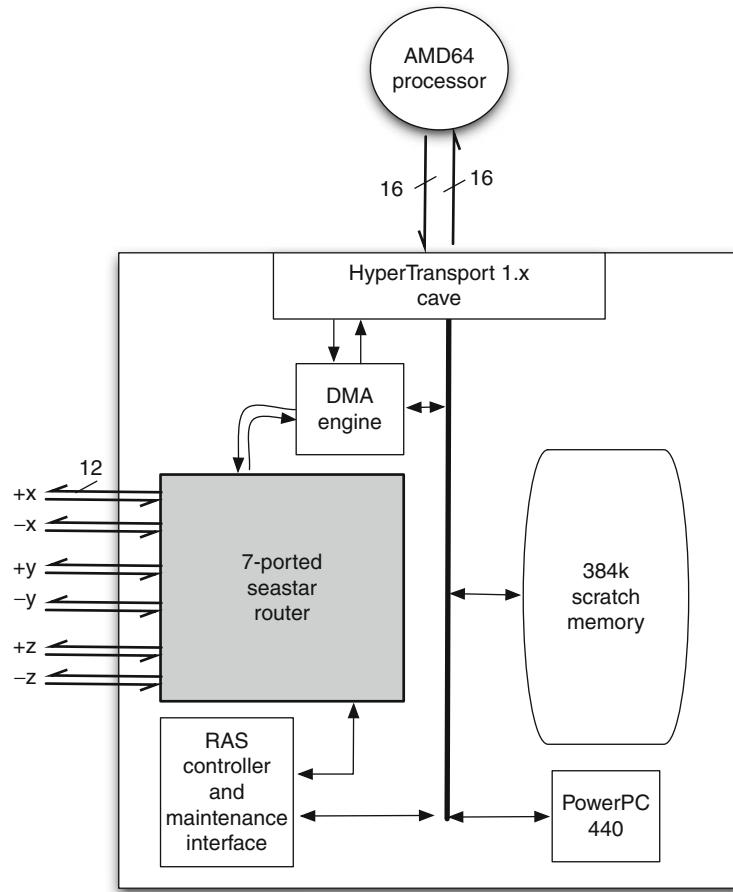
Topology

The Cray XT interconnect can be configured as either a *k*-ary *n*-mesh or *k*-ary *n*-cube (torus) topology. As a torus, the system is implemented as a *folded* torus to reduce the cable length of the wrap around link. The seven-ported SeaStar router provides a processor port, and six network ports corresponding to $+x$, $-x$, $+y$, $-y$, $+z$, and $-z$ directions. The port assignment for network links is not fixed, any port can correspond to any of the six directions. The noncoherent HyperTransport (HT) protocol provides a low latency, point-to-point channel used to drive the Seastar network interface.

Four virtual channels are used to provide point-to-point flow control and deadlock avoidance. Using virtual channels avoids unnecessary head-of-line (HoL) blocking for different network traffic flows; however, the extent to which virtual channels improve network utilization depends on the distribution of packets among the virtual channels.

Routing

The routing rules for the Cray XT are subject to several constraints. Foremost, the network must provide error-free transmission of each packet from the *source* node identifier (NID) to the *destination*. To accomplish this, the distributed *table-driven* routing algorithm is implemented with a dedicated *routing table* at each input port that is used to look up the destination port and virtual channel of the incoming packet. The lookup table at each input port is not sized to cover the maximum 32K node network since most systems will be much smaller, only a few thousand nodes. Instead, a hierarchical routing scheme divides the node name space into *global* and *local* regions. The upper three bits of the destination field (given by the destination[14:12] in the packet header) of the incoming packet are compared to the global partition of the current SeaStar router. If the global partition does not match, then the packet is routed to the output port specified in the global lookup table (GLUT). The GLUT is indexed by destination[14:12] to choose one of eight global partitions. Once



Cray XT4 and Seastar 3-D Torus Interconnect. Fig. 1 High-level block diagram of the SeaStar interconnect chip

the packet arrives at the correct global region, it will precisely route within a local partition of 4,096 nodes given by the destination[11:0] field in the packet header.

The tables must be constructed to avoid deadlocks. Glass and Ni [9] describe *turn* cycles that can occur in k -ary n -cube networks. However, torus networks are also susceptible to deadlock that results from overlapping virtual channel dependencies (this only applies to k -ary n -cubes, where $k > 4$) as described by Dally and Seitz [7]. Additionally, the SeaStar router does not allow 180° turns within the network. The routing algorithm must both provide deadlock-freedom and achieve good performance on benign traffic. In a fault-free network, a straightforward dimension-ordered routing (DOR) algorithm will provide balanced traffic across the network links. Although, in practice, faulty links will occur and the routing algorithm must route around the bad

link in a way that preserves deadlock freedom and attempts to balance the load across the physical links. Furthermore, it is important to optimize the buffer space within the SeaStar router by balancing the number of packets within each virtual channel.

Avoiding Deadlock in the Presence of Faults and Turn Constraints

The routing algorithm rests upon a set of rules to prevent deadlock. In the turn model, a positive first ($x+, y+, z+$ then $x-, y-, z-$) rule prevents deadlock and allows some routing options to avoid faulty links or nodes. The global/local routing table adds an additional constraint for valid turns. Packets must be able to travel to their local area of the destination without the deadlock rule preventing free movement within the local area. In the Cray XT, network, the localities are split with yz planes.

To allow both $x+$ and $x-$ movement without restricting later directions, the deadlock avoidance rule is modified to $(x+, x-, y+, z+ \text{ then } y-, z-, z+ \text{ then } z+, z-)$. Thus, free movement is preserved. Note that missing or broken X links may induce a non-minimal route when a packet is routed via the global table (since only $y+$ and $z+$ are “safe”). With this rule, packets using the global table will prefer to move in the X direction, to get to their correct global region as quickly as possible. In the absence of any broken links, routes between compute nodes can be generated by moving in x dimension, then y, then z. Also, when $y = Y_{\max}$, it is permissible to dodge $y-$ then go $x+ / x-$. If the dimension is configured as a mesh – there are no $y+$ links, for example, anywhere at $y = Y_{\max}$ then a deadlock cycle is not possible.

In the presence of a faulty link, the deadlock avoidance strategy depends on the direction prescribed by dimension order routing for a given destination. In addition, toroidal networks add *dateline* restrictions. Once a dateline is crossed in a given dimension, routing in a higher dimension (e.g., X is “higher” than Y) is not permitted.

Routing Rules for X Links

When $x+$ or $x-$ is desired, but that link is broken, $y+$ is taken if available. This handles crossing from compute nodes to service nodes, where some X links are not present. If $y+$ is not available, $z+$ is taken. This $z+$ link must not cross a dateline. To avoid this, the dateline in Z is chosen so that there are no nodes with a broken X link and a broken $y+$ link. Although the desired X link is available, the routing algorithm may choose to take an alternate path when the node at the other side of the X link has a broken $y+$ and $z+$ link (note the $y+$ might not be present if configured as a mesh), then an early detour toward $z+$ is considered. If the X link crosses a partition boundary into the destination partition or the current partition matches the destination partition and the current Y matches the destination Y coordinate, route in $z+$ instead. Otherwise, the packet might be boxed in at the next node, with no safe way out.

Routing Rules for Y Links

When the desired route follows a Y link that is broken, the preference is to travel in $z+$ to find a good Y link. If $z+$ is also broken, it is feasible to travel in the opposite direction in the Y dimension. However, the routing

in the node in that direction must now look ahead to avoid a 180° turn if it were to direct a packet to the node with the faulty links. When the desired Y link is available, it is necessary to check that the node at that next hop does not have a $z+$ link that the packet might prefer (based on XYZ routing) to follow next. That is, if the default direction for this destination in the next node is $z+$ and the $z+$ link is broken there, the routing choice at this node would be changed from the default Y link to $z+$.

Routing Rules for Z Links

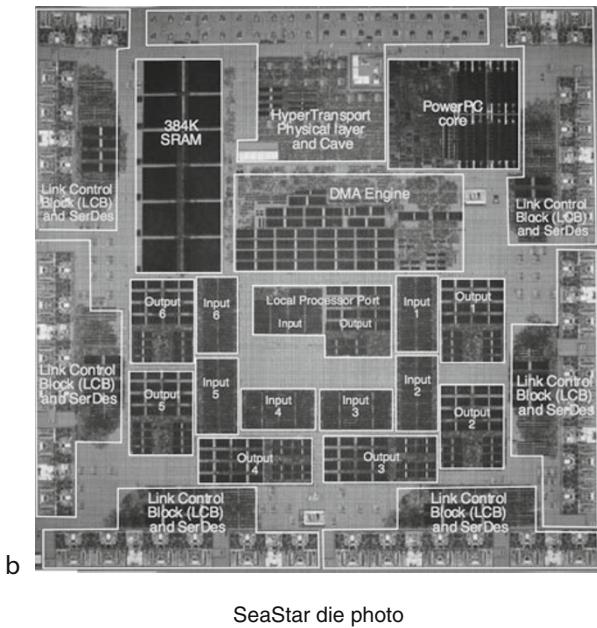
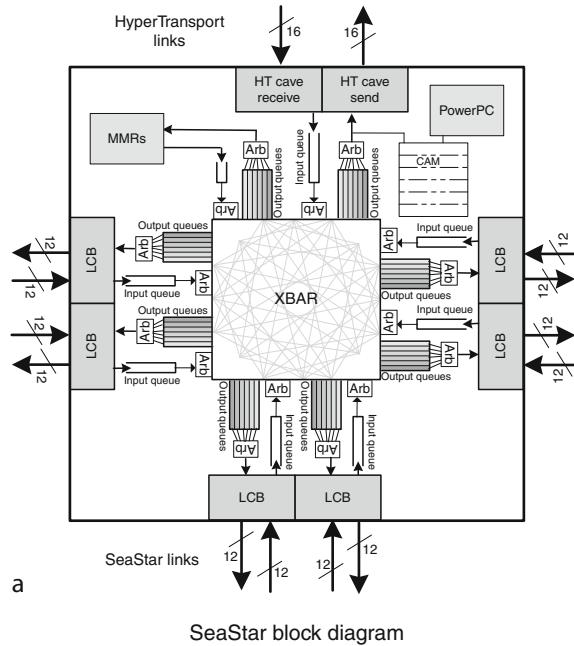
When the desired route follows a $z+$ link that is broken, the preference is to travel in $y+$ to find a good $z+$ link. In this scenario, the Y link look ahead is relied up to avoid the node at $y+$ from sending the packet right back along $y-$. When the $y+$ link is not present (at the edge of the mesh), the second choice is $y-$. When the desired route is to travel in the $z-$ direction, the logic must follow the $z-$ path to ensure there are no broken links at all on the path to the final destination. If one is found, the route is forced to $z+$, effectively forcing the packet to go the long way around the Z torus.

Flow Control

Buffer resources are managed using credit-based flow control at the data-link level. The link control block (LCB) is shown at the periphery of the SeaStar router chip in Fig. 2. Packets flow across the network links using virtual cut-through flow control – that is, a packet does not start to flow until there is sufficient space in the receiving input buffer. Each virtual channel (VC) has dedicated buffer space. A 3-bit field (Fig. 3) in each flit is used to designate the virtual channel, with a value of all 1s representing an *idle* flit. Idle flits are used to maintain byte and lane alignment across the plesiochronous channel. They can also carry VC credit information back to the sender.

SeaStar Router Microarchitecture

Network packets are comprised of one or more 68-bit *flits* (flow control units). The first flit of the packet (Fig. 3) is the *header* flit and contains all the necessary routing fields (destination[14:0], age[10:0], vc[2:0]) as well as a tail (t) bit to mark the end of a packet. Since most XT networks are on the order of several



Cray XT4 and Seastar 3-D Torus Interconnect. Fig. 2 Block diagram of the SeaStar system chip

Cray XT4 and Seastar 3-D Torus Interconnect. Fig. 3 SeaStar packet format

thousand nodes, the lookup table at each input port is not sized to cover the maximum 32k node network. To make the routing mechanism more space-efficient, the 15-bit node identifier is partitioned to allow a two-level hierarchical look up: A small eight-entry table identifies a *region*, the second table precisely identifies the node within the region. The region table is indexed by the upper 3-bits of the *destination* field of the packet, and the low-order 12-bits identifies the node within 4k-entry table. Each network port has a dedicated routing table and is capable of routing a packet each cycle. This provides the necessary lookup bandwidth to route a new packet every cycle. However, if each input port used a 32k-entry lookup table, it would be sparsely populated for modest-sized systems, and use an extravagant amount of silicon area.

A two-level hierarchical routing scheme is used to efficiently look up the egress port at each router. Each router is assigned a unique node identifier,

corresponding to its destination address. Upon arrival at the input port, the packet destination field is compared to the node identifier. If the upper three bits of the destination address match the upper three bits of the node identifier, then the packet is in the correct *global partition*. Otherwise, the upper three bits are used to index into the 8-entry *global lookup table* (GLUT) to determine the egress port. Conceptually, the 32k possible destinations are split into eight, 4k partitions denoted by bits destination[11:0] of the destination field.

The SeaStar router has six full-duplex network ports and one processor port that interfaces with the Tx/Rx DMA engine (Fig. 2). The network channels operate at 3.2 Gb/s \times 12 lanes over electrical wires, providing a peak of 4.8 GB/s per direction of network bandwidth. The link control block (LCB) implements a sliding window go-back-N link-layer protocol that provides reliable chip-to-chip communication over the network links.

The router switch is both input-queued and output-queued. Each input port has four (one for each virtual channel) 96-entry buffers, with each entry storing one flit. The input buffer is sized to cover the round-trip latency across the network link at 3.2 Gb/s signal rates. There are 24 staging buffers in front of each output port, one for each input source (five network ports, and one processor port), each with four VCs. The staging buffers are only 16 entries deep and are sized to cover the crossbar arbitration round-trip latency. Virtual cut-through [11] flow control into the output staging buffers requires them to be at least nine entries deep to cover the maximum packet size.

Age-Based Output Arbitration

Packet latency is divided into two components: *queueing* and *router* latency. The total delay (T) of a packet through the network with H hops is the sum of the queueing and router delay.

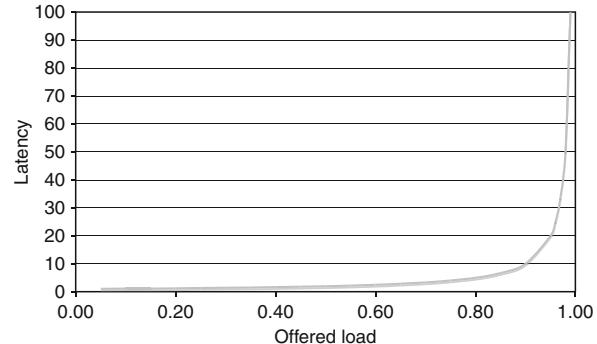
$$T = HQ(\lambda) + Ht_r \quad (1)$$

where t_r is the per-hop router delay (which is ≈ 50 ns for the SeaStar router). The queueing delay, $Q(\lambda)$, is a function of the offered load (λ) and described by the latency-bandwidth characteristics of the network. An approximation of $Q(\lambda)$ is given by an M/D/1 queue model (Fig. 4).

$$Q(\lambda) = \frac{1}{1 - \lambda} \quad (2)$$

When there is very low offered load on the network, the $Q(\lambda)$ delay is negligible. However, as traffic intensity increases, and the network approaches saturation, the queueing delay will dominate the total packet latency.

As traffic flows through the network it merges with newly injected packets and traffic from other directions in the network (Fig. 5). This merging of traffic from different sources causes packets that have further to travel (more hops) to receive geometrically less bandwidth. For example, consider the 8-ary 1-mesh in Fig. 5a where processors P0 through P6 are sending to P7. The switch allocates the output port by granting packets fairly among the input ports. With a round-robin packet arbitration policy, the processor closest to the destination (P6 is only one hop away) will get the most bandwidth – 1/2 of the available bandwidth. The processor two hops away, P5, will get half of the bandwidth into router node 6, for a total of $1/2 \times 1/2 = 1/4$ of the available



Cray XT4 and Seastar 3-D Torus Interconnect. Fig. 4
Offered load versus latency for an ideal M/D/1 queue model

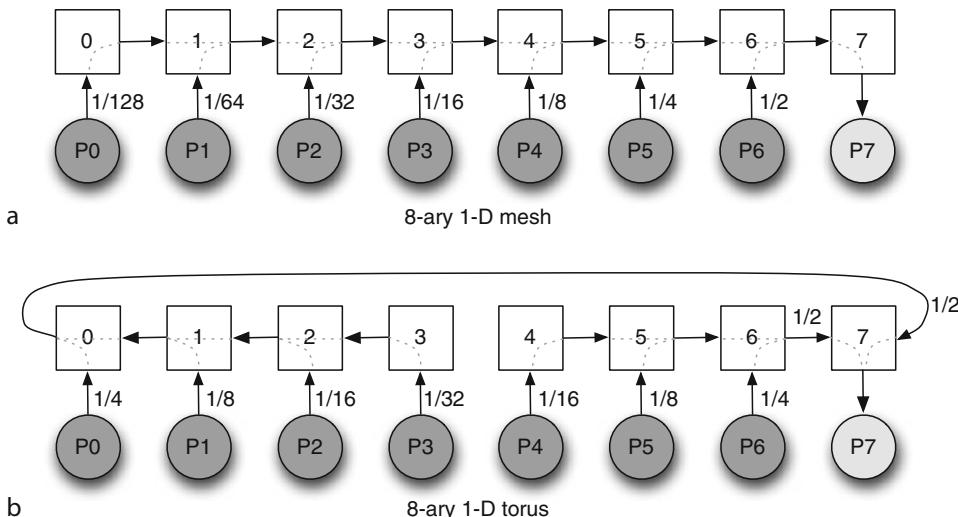
bandwidth. That is, every two arbitration cycles node 7 will deliver a packet from source P6, and every four arbitration cycles it will deliver a packet from source P5. A packet will merge with traffic from at most $2n$ other ports since each router has $2n$ network ports with $2n - 1$ from *other* directions and one from the processor port. In the worst case, a packet traveling H hops and merging with traffic from $2n$ other input ports, will have a latency of:

$$T_{\text{worst}} = \frac{L}{(2n)^H} \quad (3)$$

where L is the length of the message (number of packets), and n is the number of dimensions. In this example, P0 and P1 each receive 1/64 of the available bandwidth into node 7, a factor of 32 times less than that of P6. Reducing the variation in bandwidth is critical for application performance, particularly as applications are scaled to increasingly higher processor counts. Topologies with a lower diameter will reduce the impact of merging traffic. A torus is less affected than a mesh of the same radix (Fig. 5a and b), for example, since it has a lower diameter. With dimension-order routing (DOR), once a packet starts flowing on a given dimension it stays on that dimension until it reaches the ordinate of its destination.

Key Parameters Associated with Age-Based Arbitration

The Cray XT network provides *age-based arbitration* to mitigate the affects of this traffic merging as shown in Fig. 5, thus reducing the variation in packet delivery time. However, age-based arbitration can introduce a



Cray XT4 and Seastar 3-D Torus Interconnect. Fig. 5 All nodes are sending to P7 and merging traffic at each hop

starvation scenario whereby *younger* packets are starved at the output port and cannot make forward progress toward the destination. The details of the algorithm along with performance results are given by Abts and Weisser [1]. There are three key parameters for controlling the aging algorithm.

- AGE_CLOCK_PERIOD – a chip-wide 32-bit count-down timer that controls the *rate* at which packets age. If the age rate is too slow, it will appear as though packets are not accruing any queuing delay, their ages will not change, and all packets will appear to have the same age. On the other hand, if the age rate is too fast, packets ages will saturate very quickly – perhaps after only a few hops – at the maximum age of 255, and packets will not generally be distinguishable by age. The resolution of AGE_CLOCK_PERIOD allows anywhere from 2 ns to more than 8 s of queuing delay to be accrued before the age value is incremented.
 - REQ_AGE_BIAS and RSP_AGE_BIAS – each hop that a packet takes increments the packet age by the REQ_AGE_BIAS if the packet arrived on VC0/VC1 or by RSP_AGE_BIAS if the packet arrived on VC2/VC3. The age *bias* fields are configurable on a per-port basis, with the default bias of 1.
 - AGE_RR_SELECT – a 64-bit array specifying the output arbitration policy. A value of all 0s will select

round-robin arbitration, and a value of all 1s will select *age-based* arbitration. A combination of 0s and 1s will control the ratio of round-robin to age-based. For example, a value of 0101...0101 will use half round-robin and half age-based.

When a packet arrives at the head of the input queue, it undergoes routing by indexing into the LUT with destination[11:0] to choose the target port and virtual channel. Since each input port and VC has a dedicated buffer at the output staging buffer, there is no arbitration necessary to allocate the staging buffer – only flow control. At the output port, arbitration is performed on a per-packet basis (not per flit, as wormhole flow control would). Each output port is allocated by performing a 4-to-1 VC arbitration along with a 7-to-1 arbitration to select among the input ports. Each output port maintains two *independent* arbitration pointers – one for round-robin and one for age-based. A 6-bit counter is incremented on each *grant* cycle and indexes into the AGE_RR_SELECT bit array to choose the per-packet arbitration policy.

Related Entries

- ▶ Anton, A Special-Purpose Molecular Simulation Machine
 - ▶ Clusters
 - ▶ Cray XT3 and Cray XT Series of Supercomputers
 - ▶ Distributed-Memory Multiprocessor

- Infiniband
- Petascale Computer
- SoC (System on Chip)
- Top500

Bibliographic Notes and Further Reading

The genesis of the Cray XT4 system was the collaborative design and deployment of the Sandia “Red Storm” computer that provided the computational power necessary to assure safeguards under the nuclear Stockpile Stewardship Program which seeks to maintain and verify a nuclear weapons arsenal without the use of testing. Brightwell et al. [4] provide an early look at the SeaStar interconnection network used by the Sandia Red Storm supercomputer.

Hoisie et al. [10] use common high-performance computing (HPC) benchmarks and modeling to compare performance of three leading supercomputers: the Cray XT (Red Storm), IBM BlueGene/L, and ASCI Purple supercomputers.

Dally presents a performance analysis of k -ary n -cube networks [6]. While a more comprehensive analysis, with several examples from industry, is found in Dally and Towles [8].

Bibliography

1. Abts D, Weisser D (2007) Age-based packet arbitration in large-radix k -ary n -cubes. In: SC '07: Proceedings of the 2007 ACM/IEEE conference on supercomputing, ACM, pp 1–11, New York, 2007
2. Alam SR, Kuehn JA, Barrett RF, Larkin JM, Fahey MR, Sankaran R, Worley PH (2007) Cray xt4: an early evaluation for petascale scientific simulation. In: SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing, ACM, pp 1–12, New York, 2007
3. Brightwell R, Lawry B, MacCabe AB, Riesen R (2002) Portals 3.0: protocol building blocks for low overhead communication. In: IPDPS '02: Proceedings of the 16th international parallel and distributed processing symposium, IEEE Computer Society, p 268, Washington, 2002
4. Brightwell R, Pedretti K, Underwood KD (2005) Initial performance evaluation of the cray seastar interconnect. In: HOTI '05: Proceedings of the 13th symposium on high performance interconnects, IEEE Computer Society, pp 51–57, Washington, 2005
5. Brightwell R, Pedretti KT, Underwood KD, Hudson T (2006) Seastar interconnect: balanced bandwidth for scalable performance. IEEE Micro 26(3):41–57
6. Dally WJ (1990) Performance analysis of k -ary n -cube interconnection networks. IEEE Trans Comput 39(6):775–785
7. Dally WJ, Seitz CL (1987) Deadlock-free message routing in multiprocessor interconnection networks. IEEE Trans Comput 36(5):547–553
8. Dally WJ, Towles B (2004) Principles and practices of interconnection networks. Morgan Kaufmann, San Francisco
9. Glass CJ, Ni LM (1992) The turn model for adaptive routing. In: ISCA '92: Proceedings of the 19th annual international symposium on computer architecture, pp 278–287, 1992
10. Hoisie A, Johnson G, Kerbyson DJ, Lang M, Pakin S (2006) A performance comparison through benchmarking and modeling of three leading supercomputers: blue gene/l, red storm, and purple. In: SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, ACM, p 74, New York, 2006
11. Kermani P, Kleinrock L (1979) Virtual cut-through: a new computer communication switching technique. Comput Netw 3:267–286

Cray XT5

- Cray XT3 and Cray XT Series of Supercomputers
- Cray XT4 and Seastar 3-D Torus Interconnect

Cray XT6

- Cray XT3 and Cray XT Series of Supercomputers
- Cray XT4 and Seastar 3-D Torus Interconnect

Critical Race

- Race Conditions

Critical Sections

- Synchronization

Crossbar

- Buses and Crossbars
- Networks, Multistage

CS-2

►Meiko

CSP (Communicating Sequential Processes)

A. W. ROSCOE, JIM DAVIES
Oxford University, Oxford, UK

Synonyms

Communicating sequential processes (CSP)

Definition

Communicating Sequential Processes (CSP) is a mathematical notation for describing patterns of interaction. It has been used in the analysis of concurrent behavior in a variety of applications; it has inspired the design of concurrency mechanisms and primitives in several programming languages; it remains a focus for research and development in both academia and industry.

Process Language

In the CSP notation, *processes* are used to specify the behavior of components, to express assumptions about behavior, or to characterize behavioral properties. In each case, behavior is described in terms of the occurrence and availability of abstract entities called *events*: these are transactions, or atomic, multi-way synchronizations, between combinations of processes.

Processes may be combined using a number of operators, denoting choice, sequencing, and parallelism. The resulting language has a range of algebraic equivalences, and is often referred to as a *process algebra*. These equivalences help to explain the meaning of the operators, and support the automatic transformation of process descriptions into forms more convenient for analysis or implementation.

The alphabet of a process is the set of all events that require its participation. If an event appears in the alphabet of two or more processes in a parallel combination, then we say that this event is *shared* between

them. A shared event cannot occur without the simultaneous participation of all of the processes involved, and so each process added to a parallel combination represents a further constraint upon the order in which shared events may occur.

The simplest operator is *Stop*, which describes a process that can do nothing, and hence prevents the occurrence of any event in its alphabet. The prefix operator \rightarrow is used to introduce an event into a process description: if a is any event, and P is any process, then $a \rightarrow P$ is a process that is ready to engage in event a and, should a occur, will then behave as P . This choice of language is deliberate: a may be shared with other processes, and so $a \rightarrow P$ cannot simply “do” a ; indeed, if it is sharing a with *Stop*, then a will never occur.

A generalized prefix notation encompasses a process that offers the choice of a set of events: $?x : A \rightarrow P(x)$ is prepared to communicate any event $a \in A$ and then behaves like the corresponding process $P(a)$. You can think of A as being a menu of alternatives that the process offers.

This choice is further generalized to one between processes by the external choice operator $P \square Q$. This offers the first events of P and Q and behaves accordingly, so that

$$a \rightarrow P(a) \square b \rightarrow P(b) = ?x : \{a, b\} \rightarrow P(x)$$

In the case where P and Q have overlapping sets of initial events, $P \square Q$ acts *nondeterministically*: in $(a \rightarrow P) \square (a \rightarrow Q)$, once a has been communicated then the process can behave like P or Q , at its *internal* choice.

Because internal choices like this arise in CSP, both for this reason and others, there is an operator $P \sqcap Q$ that expresses internal, or nondeterministic choice. This process may choose to behave like either P or Q and neither the user nor other processes have any influence over which. Thus

$$(a \rightarrow P) \sqcap (a \rightarrow Q) = a \rightarrow (P \sqcap Q)$$

The sequential composition $P ; Q$ behaves as P until that process reaches a point in its behavior described by *Skip*, denoting successful termination; the subsequent behavior of the combination is then that of Q .

$$(a \rightarrow \text{Skip}) ; Q = a \rightarrow Q$$

for any event a and any process Q . *Skip* is distinct from *Stop*, which does not even terminate successfully.

In the parallel composition $P \parallel Q$, the two processes P and Q will evolve independently, while sharing in the occurrence of any event that is common to the two alphabets. If a and b are *not* shared between the two processes, then

$$a \rightarrow P \parallel b \rightarrow Q = a \rightarrow (P \parallel b \rightarrow Q) \sqcap b \rightarrow (a \rightarrow P \parallel Q)$$

that is, a and b may be observed in either order.

If P and Q can neither communicate independently nor agree on a shared event, then $P \parallel Q$ is *deadlocked*, and equivalent to *Stop*. For example, if a and b are shared between the two processes, then

$$a \rightarrow b \rightarrow P \parallel b \rightarrow a \rightarrow Q = \text{Stop}$$

the left-hand component of the parallel composition is able to perform b , but only after a has occurred; the right-hand process is able to perform a , but only after b has occurred; each event requires the cooperation of both processes, so no progress is possible.

The interleaving operator $P \parallel\!\!| Q$, on the other hand, allows each of P and Q to communicate freely: no communications are synchronized even when they belong to the alphabets of P and Q ; it follows that $P \parallel\!\!| Q = P \parallel Q$ when P and Q have disjoint alphabets.

The hiding operator $P \backslash X$ removes all actions in the set X from both the sight and control of the external environment. Thus

$$(a \rightarrow P) \backslash \{a\} = P \backslash \{a\}$$

It is frequently used to conceal the internal communications of a parallel composition as in $(P \parallel Q) \backslash (\alpha P \cap \alpha Q)$ where αP and αQ are respectively the alphabets of P and Q .

If R is a relation between the set Σ of all visible events and itself, then $P[R]$ is a process that can perform b whenever P can perform a for any pair (a, b) related by R . Sometimes, the special cases where R is either a function f or an inverse function f^{-1} are considered separately.

The language constructs above are usually considered the “core” of the notation. However, Hoare [7] also introduces interrupt: $P \triangle Q$ behaves like P until an initial event of Q occurs, after which Q takes over. This is typically used to handle externally occurring exceptions

as in

$$\text{resettable}(P) = P \triangle (e \rightarrow \text{resettable}(P))$$

where e is an event representing an external exception. Roscoe [19] introduces an operator for handling internal exceptions: $P \Theta_{\text{error}} Q$ behaves like P until the event error occurs, at which point Q takes over. Thus

$$(a \rightarrow P) \Theta_a Q = a \rightarrow Q$$

Process Semantics

The language of processes described above can be given a formal meaning, or semantics. Any process can be associated with a unique set of sequences, or *traces*, comprising every sequence of events, within its alphabet, that this process will allow. For example, the process *Stop* allows only the empty sequence $\langle \rangle$, while the prefix process $a \rightarrow P$ will allow any sequence in which the first event, if there is one, is a , and for which the remaining events could be performed by P .

The trace semantics is enough for many applications, but does not provide for an adequate treatment of nondeterminism. For example, the processes $a \rightarrow \text{Stop}$ and $(a \rightarrow \text{Stop}) \sqcap \text{Stop}$ have the same set of traces, the set $\{\langle \rangle, \langle a \rangle\}$. Yet the first of these *must* be ready to perform a , whereas all that can be said for the second is that it *may* be ready to perform a (or it may deadlock).

For nondeterministic processes, another level of semantic information may be required. Each trace of a process may be associated with a set of sets of events, or *refusal sets*, comprising every combination that may be blocked by the process once that trace has been performed. The combination of a trace and a single refusal set is called a *failure*: if the process, having performed that trace, were required to participate in one of the events from the refusal set, then no further progress would be made.

To provide a full characterization of the process language, the failures semantics must be augmented by an appropriate treatment of undefined expressions. Where the behavior of a process P is determined solely by an equation such as $P = P$, or even $P = P \sqcap (a \rightarrow \text{Stop})$, then there is not enough information to uniquely determine a set of failures; such a process is said to be *divergent*. Processes like $AS \backslash \{a\}$, where $AS = a \rightarrow AS$, are also divergent. In the definitive *Failures-Divergences* semantics, each process is associated with a set of failures and

a set of divergences: traces leading to a point where the subsequent behavior may be described by a divergent process. Besides traces and failures-divergences, there are a variety of other models. See [19] for details.

In analyzing the semantics of CSP, it is useful to have an additional form of choice: $P \triangleright Q$ may choose to offer the initial actions of P , but *must* offer the initial choices of Q . It is equivalent to $(P \square a \rightarrow Q) \setminus \{a\}$ for a an event that neither P nor Q performs.

Any finite CSP program (one whose traces are of bounded length) is equivalent to one written using only *Stop*, *Skip*, prefix-choice, internal choice \sqcap , sliding choice \triangleright , and a constant *div* representing a *divergent* process: one equivalent to $A\$ \setminus \{a\}$ that simply performs internal (τ) actions for ever. This fact tells us that, in some sense, the other operators are not language *primitives* and that every program is equivalent to a sequential one. The fact that this transformation can be accomplished using algebraic laws is the core of an *algebraic* semantics for CSP (see Chap. 13 of [19]).

Refinement and Abstraction

In applications of CSP, processes are used to describe components, assumptions, or properties, and then compared in terms of their semantics. The usual method of comparison is based on the notion of *refinement*, which corresponds to subsetting or containment of the semantics: a process P is a refinement of another process Q , written $Q \sqsubseteq P$, if every behavior of P – every trace, or every failure and divergence – is also a behavior of Q .

In the trace semantics, refinement can be used to check whether a particular sequence of interaction is possible, or to check whether every possible sequence of interaction is acceptable according to some specification. However, for nondeterministic processes, it cannot be used to demonstrate that a sequence *must* be possible: for this, refinement in the failures–divergences semantics is required.

Most comparisons will be made between processes described at different levels of abstraction: one of the processes will describe a design or implementation, the other a property or specification, using a smaller set of events. To effect such a comparison using refinement, the events not mentioned in the property description must be abstracted.

CSP provides a number of mechanisms for describing abstraction, including renaming, hiding and *lazy*

abstraction: $\mathcal{L}_A(P)$ describes now the process looks to a user who cannot see the events in A but assumes there is another user who can see and control them. This is identical to hiding in the traces model, but subtly different in other models since $P \setminus A$ assumes that the events of A become automatic and uncontrolled.

A key feature of the process language is that all of the operators are monotonic with respect to the refinement ordering: if $Q \sqsubseteq P$, then $C[P]$ refines $C[Q]$ where $C[\cdot]$ is any CSP context, namely, a piece of CSP syntax with a free process variable.

Tools and Applications

The multiplication of states and entities that is inherent in concurrent behavior means that (fully) manual analysis of process semantics is infeasible. Effective tool support is required: to examine the properties of a process through step-by-step rewriting, or to check whether or not one process is a refinement of another, through exhaustive exploration.

A *machine readable* syntax for the language has been defined, called CSP_M : process descriptions written in this syntax can be checked for type consistency using the *Checker* tool, and rewritten step-by-step using a tool called *ProBE*. A refinement checking tool called *Failures–Divergences Refinement* (FDR) is also available: the refinement of processes with millions of control states can be checked in less than a minute; designs with billions of states have been checked successfully. Processes with larger, finite state spaces can be checked using *compression* operators and related techniques.

A number of people, for example [9], have embedded theories of CSP into theorem provers such as Isabelle and PVS: such embeddings make it easier to prove general results such as language equivalences and healthiness conditions, but are not as effective as FDR at proving results about specific systems.

A key application area for CSP has been in the analysis of cryptographic protocols. Gavin Lowe discovered that these could be conveniently modeled in CSP and checked for security on FDR, and he and others developed tools – most notably a front end for FDR called Casper [14] – that can input a protocol in a natural notation and either find an attack on it or prove there is no such attack. The fact that Lowe, early on in this work, found a famous attack on a well-known protocol [13] led

to an explosion of work around the world on the formal verification of security protocols.

Apart from its use in security, CSP has been used successfully for the development of critical systems for avionics, automotive, military, and embedded systems. Some of these are based on modeling directly in CSP_M , and some, like Casper, rely on front ends that translate another notation to CSP_M .

Language Versions

There are several different versions of the process language, the best known being the mathematical notation used in Hoare's 1985 book *Communicating Sequential Processes* [7], used here. An extended version of that and the "machine readable" variant CSP_M are used in Roscoe's 1997 book *The Theory and Practice of Concurrency* [16]. CSP_M is a significant extension of the 1985 notation, with more powerful notions of sequencing and abstraction, as well as directives for expediting the checking of traces and failures–divergences requirement. CSP_M also contains a functional programming language related to Haskell [2] that is used both for building processes' events and parameters, and for laying out networks. These same two versions of CSP are used in Roscoe's 2010 book *Understanding Concurrent Systems*.

The main difference between the Hoare and Roscoe presentations is that whereas Hoare assumes that every process has an intrinsic alphabet for use in the parallel operator, Roscoe uses versions of parallel that specify alphabets explicitly: for example $P[|A|]Q$ (in the CSP_M syntax) runs P and Q with interface A .

Hoare introduced an early version of the notation in 1978, in the Communications of the ACM [6]. This was a language of programs rather than constraints or patterns of behavior; however, it embodied two of the characteristic features of the later notation: that the behavior of a process should be characterized entirely by its external interaction, and that parallel composition should be a primary means of adding structure to process descriptions.

The CSP notation in both its 1978 and process algebra versions was the inspiration for the programming language OCCAM [8], implemented on the INMOS Transputer: both the language and the processor are discussed elsewhere in this volume. It has inspired also the design

of concurrency mechanisms for the Java language [11] and the Go language released in 2009 by Google [5].

An account of the history of CSP up to the mid-1980s and its influences on OCCAM and the Transputer can be found in [10], a scientific biography of Hoare.

Timed, prioritized, and probabilistic extensions have been suggested for the CSP notation, and attempts have been made to integrate it with object modeling and state-based notations such as B [1]. Of these, there has been considerable industrial use of CSP for timed and state-based systems, but always by translation from the extended notation into CSP_M and then checking on FDR. For example, the passage of time can be represented by the regular occurrence of a *tock* event.

Discussion

Many other process algebras have been proposed for the study of concurrency, most notably Milner's CCS and its many derivatives, including the π -calculus. Among these, CSP is characterized by a rigorous adherence to a small, coherent set of design principles:

- Each event is an atomic, symmetric, multi-way synchronization, corresponding to an abstract transaction involving a fixed collection of processes.
- Each process gives only a partial account of when an event may occur: until closure is achieved through abstraction, other processes may further constrain its occurrence.
- Processes are characterized completely by their externally observable behavior: there are no "internal" or "hidden" events in the language.
- There is no explicit treatment of priority, timing, or degree of parallelism; these require more specific interpretations of interaction and communication.
- The semantics is captured by models which describe a process as one or more sets of observations of *linear*, as opposed to *branching* behaviors.

As a result, the language has certain properties not found in other algebras: most notably, processes describing components, assumptions, or properties at different levels of abstraction may be compared using notions of refinement.

In other languages, comparisons between processes are made only at the same level of abstraction, using various notions of simulation; analyses across different levels of abstraction are conducted using separate

formalisms, such as temporal logic. Such logics are often excellent at expressing succinct properties of systems, but using CSP descriptions as refinement specifications frequently has an advantage when capturing a more complete description of a system [12]. It is possible to express some subsets of Linear Temporal Logic in CSP, and aspects of fairness, but any property that can be checked over standard CSP semantics must be determined by a process's *linear* as opposed to branching behavior.

On a theoretical note, it can be shown that every *closed* property R of CSP models, and many others, can be characterized by refinement checks of the form $F(P) \sqsubseteq G(P)$ for CSP contexts F and G [17]. Here, R is closed if, whenever $R(P)$ does not hold, there is some n such that there is no Q which both satisfies R and is equivalent to P for the first n steps of its behavior.

Recent results (see Chap. 9 of [19]) have demonstrated that the CSP notation is in some sense a *universal* language for concurrency: every language whose operational semantics satisfies the conditions to be *CSP-like* can be simulated (at the level of strong bisimulation) by CSP. Any such language is automatically given compositional semantics over CSP models and a theory of refinement. One such language is π -calculus: see [18].

Related Entries

- Bisimulation
- Deadlocks
- Determinacy
- Formal Methods-Based Tools for Race, Deadlock, and Other Errors
- Pi-calculus
- Process Algebras

Bibliographic Notes and Further Reading

A comprehensive account of CSP is presented in Roscoe's book *Understanding Concurrent Systems*, published by Springer in 2010 [19], a significant update of his 1997 book *The Theory and Practice of Concurrency* [16]: this includes a guide to the latest version of the "machine readable" language, and a detailed account of tools, applications, and extensions. It explains the links with other ideas from concurrency such as shared variables, data-flow and buffered channels, mobility, time, and priority by showing how to model these

in CSP. The vision for the process language, and the failures-divergences semantics, is set out in Hoare's 1985 book, *Communicating Sequential Processes*; key stages in its development can be seen in Hoare's 1978 paper of the same name [6] and the 1984 paper by Brookes et al. [3]. Schneider [21] is a book on CSP that emphasizes the theory of Timed CSP (see also [4, 15]). The application of CSP to security protocols is covered in [20].

Bibliography

1. Abrial JR (1996) The B-book: assigning programs to meanings. Cambridge University Press, Cambridge
2. Bird R (1998) An introduction to functional programming using Haskell. Prentice-Hall, Hertfordshire, UK
3. Brookes SD, Hoare CAR, Roscoe AW (1984) A theory of communicating sequential processes. J ACM 31(3):560–599
4. Davies J (1993) Specification and proof in real-time CSP. Cambridge University Press, Cambridge
5. Google (2010) The go programming language. <http://golang.org/>. Accessed July 2010
6. Hoare CAR (1978) Communicating sequential processes. Commun ACM 21(8):666–677
7. Hoare CAR (1985) Communicating sequential processes. Prentice Hall (Available from <http://www.usingcsp.com>)
8. Inmos Ltd. (1988) Occam2 reference manual. Prentice Hall
9. Isobe Y, Roggenbach M (2005) A generic theorem prover of CSP refinement. TACAS 2005, Springer
10. Jones CB, Roscoe AW (2010) Insight, innovation and collaboration. Reflections on the work of C.A.R. Hoare. Springer, London
11. Lea D (1996) Concurrent programming in java: design principles and patterns. Addison Wesley, Reading
12. Leuschel M, Currie A, Massart T (2001) How to make FDR Spin: LTL model checking of CSP by refinement. FME 2001. Springer
13. Lowe G (1996) Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In: Proceedings of TACAS '96. Lecture notes in computer science, vol 1055. Springer, Berlin
14. Lowe G (1977) Casper: a compiler for the analysis of security protocols. Proceedings of CSFW 1997
15. Reed GM, Roscoe AW (1998) A timed model for communicating sequential processes. Theor Comput Sci 58:249–261
16. Roscoe AW (1997) The theory and practice of concurrency. Prentice Hall, Hertfordshire, UK
17. Roscoe AW (2003) On the expressive power of CSP refinement. Form Asp Comput 17(2):93–112
18. Roscoe AW (2010) CSP is expressive enough for π . In: Reflections on the work of C.A.R. Hoare, Springer, London
19. Roscoe AW (2010) Understanding concurrent systems. Springer, London
20. Ryan PYA, Schneider SA, Goldsmith MH, Lowe G, Roscoe AW (2001) The modelling and analysis of security protocols: the CSP approach. Addison-Wesley, Reading
21. Schneider SA (2000) Concurrent and real-time systems: the CSP approach. Wiley, New York

Cyclops

MONTY DENNEAU
IBM Corp., T.J. Watson Research Center, Yorktown Heights, NY, USA

Synonyms

Cyclops-64

Definition

Cyclops is a targeted-purpose, highly parallel, massively multicore family of computers developed at the IBM T.J. Watson Research Center. The design is characterized by several novel architectural features and by an aggressive exploitation of ASIC chip technology. Machines under construction range in performance from tens of teraflops to a full petaflop.

Discussion

Introduction

Cyclops falls into the category of targeted high-performance computing systems. These machines are built with an intentionally unconventional balance of resources in order to optimize cost-performance for specific application domains. Cyclops was designed to target proprietary optimization codes for a select group of IBM clients.

Five observations of the execution of these codes on commercial machines motivated several of the key architectural decisions:

1. The majority of instructions were not used most of the time. This was true even for so-called RISC processors. Instructions were found that were never emitted by any compiler.
2. Much of the architectural, design, and implementation work that went into optimizing single-thread performance was wasted. Many processor resources were idle much of the time, wasting clock and leakage power.
3. Data caches consumed a large part of the silicon resources, but often increased the energy per operation and, in some cases, even decreased performance.

4. The traditional splitting of the general-purpose register file into fixed and floating-point components almost always resulted in the wrong balance. For applications without floating point data, all of the floating-point registers were wasted. For applications with very high floating point content, many of the fixed-point registers went unused.
5. When processors were running the same small program, the dedication of a large instruction cache to just one or two processors wasted silicon resources.

Thread Unit

The fundamental Cyclops computing element is the 64-bit general-purpose thread unit, the instruction set architecture for which was developed without any particular tie to previous machines.

The thread unit contains a 4-port (2 read, 2 write) 64×64 register file and a 32-KB SRAM (Static Random Access Memory). It executes one scalar instruction per cycle. Instructions and data are, respectively, 32 and 64 bits wide. There are only 60 basic instruction types.

Instructions are executed from a 32-element prefetch buffer (PIB). While execution is not speculative, instructions are prefetched using a 512-entry 2-bit branch history table (BHT).

The name thread unit is historical – a thread unit is actually a complete processor that can execute a fixed point instruction every cycle.

Processor

Two thread units and a shared floating-point unit constitute a processor. The floating-point unit can start one double precision add and one double precision multiply every cycle. Underflow is truncated to zero and overflow is forced to infinity – Cyclops has no interrupts on arithmetic exceptions. The multiply unit can also start a $64 \times 64 \rightarrow 136$ integer multiply-accumulate operation every cycle.

A processor also has 100-bit outgoing and incoming ports to the chip-wide crossbar described below.

Instruction Cache Group

Five processors (ten thread units) and a shared Instruction Cache constitute an Instruction Cache Group.



The 32-KB shared Instruction Cache is 8-way associative and has 64-byte lines. Replacement is Least-Recently-Used.

On each cycle the Instruction Cache can provide one 16-instruction line. These are aligned to 16-byte (4-instruction) boundaries. Requests to the same line are combined and satisfied simultaneously.

Crossbar

On-chip Cyclops communication is provided by a large pipelined crossbar. This avoids the locality constraints that accompany, for example, nearest-neighbor interconnects. A full crossbar ensures that programmers do not have to consider the particular positions of processors on the chip.

The crossbar has 96 ports, each approximately 100 bits wide. It is pipelined so that, in the absence of blocking, transactions can be initiated on every cycle. Transactions can encompass either single items or blocks of items. Forward and return transactions share the same wires, but otherwise have separate facilities in order to prevent deadlock.

Of the 96 crossbar ports, 80 connect to the processors, 4 to the Instruction Caches, 4 to the DRAM controllers, 6 to the 3D router, 1 to the combining router, and 1 to the parallel host interface.

Loads and Stores through the Cyclops crossbar have an important ordering property, which is best illustrated by example: If a processor stores data into global memory and sets a flag to indicate that the data has been written, then another processor observing that the flag has been set will subsequently read the correct data. Unlike most other multiprocessors, Cyclops does not require or have a synch instruction.

The unloaded crossbar latency is 10 cycles.

Storage

Each of the 160 Cyclops processors contains 32 KB of SRAM. The SRAM is partitioned at runtime into two parts. The first is local scratchpad, which is most often directly accessed by the owning processor, but which can also be explicitly accessed by any other processor on the chip.

The second is a contribution to global interleaved SRAM. This memory segment is interleaved across all participating processors, so that the first 64 bytes are in the SRAM of the first processor, the next 64 bytes in the

SRAM of the second processor, and so on. In order to enforce ordering, access to global interleaved memory is always done over the crossbar, even if a particular item is local to the accessing processor.

Attached to each Cyclops chip is one gigabyte of external DDR2 DRAM, implemented as four independent quadrants, each with its own port to the chip. Data is interleaved in 64-byte blocks, first across the quadrants, and then within the internal banks of the DRAM modules. The peak bandwidth into external DRAM is 16 GB/s.

Only the processors on a chip can directly address the memory associated with that chip. Remote access is handled by message routing hardware described below and SHMEM utilities.

Atomic Memory Operations

All levels of Cyclops storage support atomic memory operations. These include both fetch-and-add and fetch-and-logic with any 2-input logical operation. Alternative versions perform the update without returning data.

Signal Bus

The signal bus is a 16-bit wide 160-way pipelined OR tree used to support fast barrier operations. To preserve ordering, processors write to the signal bus over the crossbar, but read from it locally. Each processor observes the logical OR of the contributions of all 160 processors. Three bits are typically used to implement each barrier.

A-Switch

Each Cyclops chip contains a simple 6-port 3D router referred to as the A-Switch. Chips are physically connected in a 3D mesh. The petaflop version of Cyclops is a $24 \times 24 \times 24$ cube.

Each of the six external channels simultaneously sends and receives data at a rate of 4 GB/s, for an aggregate external data bandwidth of 48 GB/s. A small amount of additional bandwidth is present for control.

Each channel consists physically of 12 differential pairs (lanes), each running at 5 Gbits/s. Nine of the lanes are for payload, two for block error correction, and one is reserved as a spare. The block error correction is strong enough to allow total failure of any single lane.

There are four classes of traffic. Two are for normal forward and reply messages, and two are for forward and reply messages that must be rerouted to avoid a bad chip.

Messages typically travel first along the x -axis, then along the y -axis, and finally along the z -axis of the mesh. The number of hops along each axis is contained in the header.

If chip A cannot reach chip B because of a failing chip along the route, then chip C is found such that A can reach C and C can reach B. As the message passes C, its class is changed to the reroute class. This mechanism prevents deadlock.

In order for processors to collectively participate in the creation of messages, traffic injected into the A-Switch is mediated by six outgoing channel controllers, one for each direction. Processors assemble components of messages in memory, and then atomically queue a pointer to the appropriate channel controller.

Similarly, there are six incoming channel controllers responsible for dejecting data from the A-Switch. Data is written to FIFOs allocated in memory. Processors remove message pointers from an incoming work FIFO, process the corresponding data, and then notify a channel controller when done so that it can free the space.

B-Switch

In a machine with over 13,000 processor chips, collective operations such as floating-point reductions, global max/min, and barriers can, if implemented in software, negatively affect performance. The Cyclops processor chip incorporates a component referred to as the B-Switch, which accelerates these operations.

Traffic to and from the B-Switch uses the same physical links as the A-Switch, with B-Switch traffic always taking priority. Processors directly inject packets into and deject packets from the B-Switch. B-Switch transactions are strictly between neighbors in the 3D mesh.

Central to the unit is a set of 16 trigger tags. Each tag is programmed with criteria for activation based on B-Switch packets received from the six neighboring chips. The tag can either require that all members of some particular subset of the neighbors have sent B-Switch packets, or that any member of a particular subset has sent a packet.

In addition to the activation criteria, the tag also contains the location of an applet in a small instruction memory. When a tag is activated, it posts a notification of that event on an activation queue.

The B-Switch contains a simple embedded processor that is a subset of the primary Cyclops processor. Only instructions important for collective operations are implemented. For example, floating-point addition and comparison are supported, but multiplication is not.

The B-Switch processor executes, one by one, the applets on the activation queue. During execution, the processor typically sends new packets to adjacent chips and also notifies the sources that an operation is complete and resources are now free.

A typical B-Switch operation is to determine a maximum floating point value across all chips in a $24 \times 24 \times 24$ machine, and then to broadcast that value to all participants. This operation is done in two phases, each with an associated tag. First, the comparisons are done in a B-Switch tree that terminates near the center of the cube. Then, the chip at the root of the tree changes the tag and, in the reverse direction, broadcasts the maximum value to all participants.

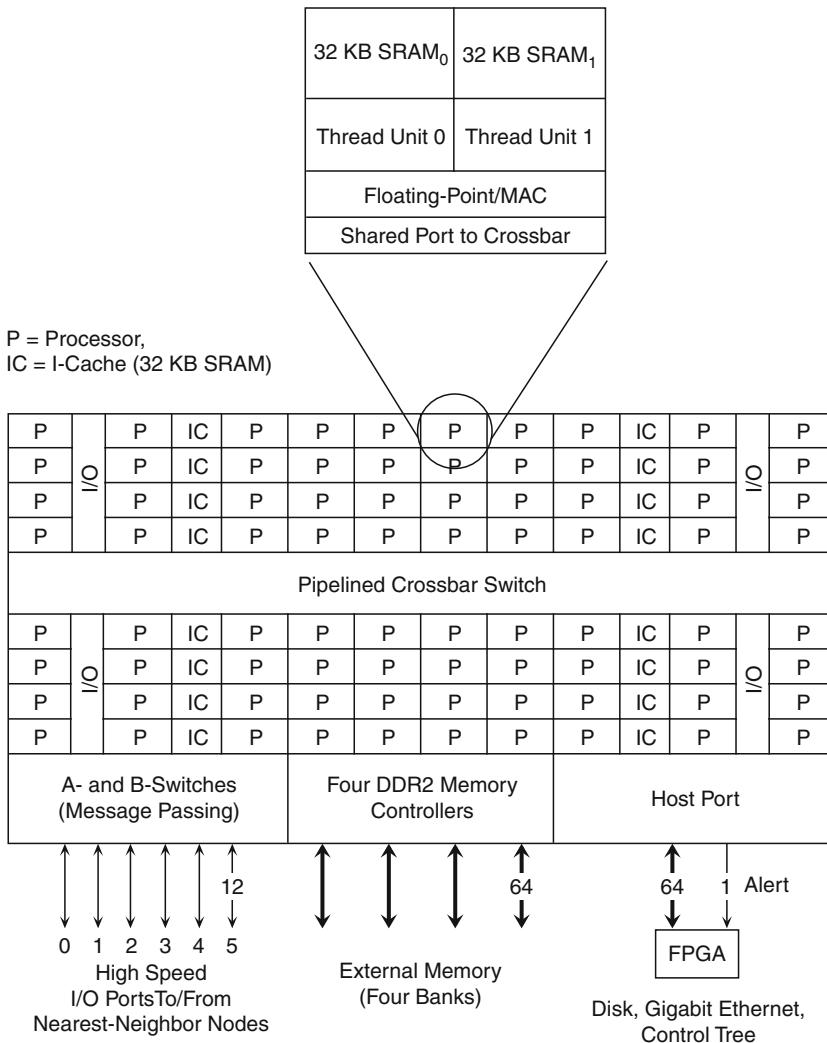
An overview of the Cyclops chip is shown below (Fig. 1).

Single-Step

Cyclops has a unique single-step property. It can be started from cycle 0, run M cycles, stopped and examined, then run an additional N cycles, and it will behave exactly as if it were run $M + N$ cycles without interruption.

Several mechanisms were required in order to enable single-step. There is a unique central oscillator, and the rising edges of the clock can be thought of as being numbered. All chips update their state in the same way on edge N , even though they may receive edge N at different times.

The major challenges to single-step were the DRAM and HSS (High Speed SerDes) subsystems. On any given cycle these can have transactions or data in flight. When the machine pauses, Cyclops sinks in-flight results to completion buffers. Upon starting again, the machine sources results from the completion buffers in such a way that they appear on the same clock cycles that they would have if no pause had occurred.



Cyclops. Fig. 1 Overview of the Cyclops chip

Processor Sparing

Cyclops was the first chip designed within IBM to exploit large scale processor sparing in order to reduce silicon costs. Even in a mature technology, yields of perfect die the size of the $21 \text{ mm} \times 23 \text{ mm}$ Cyclops die would not be high. A single failed logic transistor or wire results in the die being discarded.

The Cyclops ASIC can tolerate any number of failed processors, with the sole consequence being a linear reduction in performance and SRAM capacity. This is achieved by mapping out the failed processors and compressing the remaining interleaved memory space so that it appears to be contiguous.

The compression is done in two steps. First, fast arithmetic hardware divides the address by the number of good banks and produces the quotient and remainder. The quotient is the address in the target bank. Next, the remainder is used to index a remap table that lists, without gaps, the good banks. The output of the remap table is the target bank.

Mechanical Package

Each Cyclops ASIC is contained on its own $6'' \times 9''$ water-cooled blade. The blade contains one gigabyte of off-the-shelf DRAM, an FPGA for external Ethernet

access and hard disk drive control, power supply components, and support for an integral hard disk drive.

Water-cooling is implemented by means of a cold plate constructed from fused sheets of etched copper. The junction temperature of the ASIC is typically held to under 30 °C, enhancing reliability and improving system speed.

Twenty-four blades are mounted on each side of a midplane, and three midplanes make up a rack of 144 blades. The petaflop installation of Cyclops has 12 rows of 8 racks each.

Midplanes are connected within rows using high-performance flex. Row-to-row connections are made with 12X Infiniband cable.

Power consumption of the petaflop machine is under two megawatts.

Programming Model

Within a chip, Cyclops is a 160-way SMP. In the simplest mode of operation, a single program is loaded into shared memory. Processor identity determines which parts of the program to execute and on what data to operate. There is also a lightweight thread library similar to pthreads.

Between chips there are simple message-sending and Shmem-like shared storage libraries. There is no support for MPI.

The primary software for Cyclops was developed under the direction of Dr. Guang Gao at the University of Delaware.

Performance

While Cyclops was not designed for high floating point performance, it still achieves over 70 GFlops/s multiplying double-precision matrices stored in external DRAM. The corresponding power is 1 watt/GFlop.

The chip achieves approximately 0.09 GUPS (giga updates per second) to external DRAM.

Design

The Cyclops ASIC, a 500-MHz 90-nm design, was the largest die ever processed by the IBM ASICS division. It had the densest logic implementation of any ASIC, the highest content of low threshold voltage logic, the highest latch count, and the highest peak power. As a

result, it posed many unique challenges to the existing design methodology and tools.

Work on early versions of Cyclops began in the late 1990s to address first the challenges of raytracing, and later protein-folding. The ASIC was designed by very small teams at the IBM T.J. Watson Research Center, the IBM Burlington and Fishkill facilities, and by Dr. Gregory Chudnovsky at the Polytechnic Institute of New York University. Packaging and electronics were developed by Central Scientific Services at IBM Research under the direction of James Speidell.

Future Work

Aside from the expected improvements in speed and processor count resulting from the expected course of technology, the next generation of Cyclops will take advantage of two major advances. First, multiple processor and memory die will be tightly integrated on the same module, greatly increasing the bandwidth between them. Second, cost-effective optics and different topologies will enable a large increase in global long-hop bandwidth.

Probably the single most distinctive and valuable feature of the Cyclops architecture is the very large pipelined crossbar. This is expected to persist, with increasing size and bandwidth, through all future generations.

Appendix

The Cyclops die is shown in the photograph below (Fig. 2). Colors are due to diffraction. The photograph was taken before fabrication of the upper metal layers.

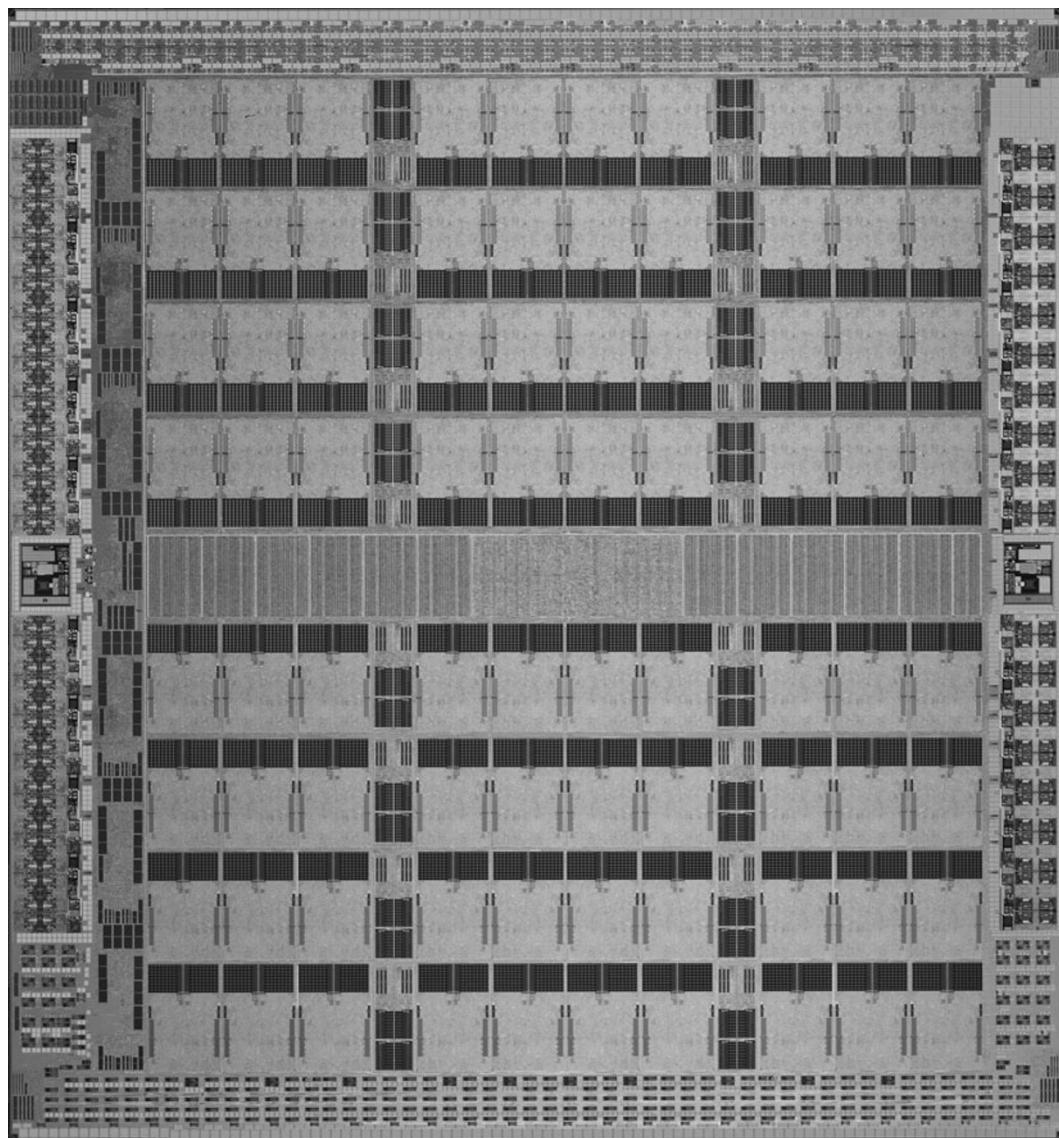
On the left edge are nine 8-bit HSS (High Speed SerDes) receivers, with a phase-locked loop in the center. On the right edge are the corresponding nine 8-bit HSS transmitters.

To the right of the HSS receivers are the A-Switch and B-Switch.

On the top and bottom edges are four DRAM controllers.

Running horizontally across the center is the 96-port pipelined crossbar.

The two thin columns approximately one-third and two-thirds of the way across the die are 16 Instruction Caches.



Cyclops. Fig. 2 The Cyclops die

Finally, most of the die is filled by the 8×10 array of processors, each of which contains two thread units and a floating-point unit. Two blocks of SRAM can be seen attached to each processor.

Cyclops-64

►Cyclops

Cydra 5

MICHAEL SCHLANSKER
Hewlett-Packard Inc., Palo-Alto, CA, USA

Definition

The Cydra 5 was a mini-supercomputer designed by Cydrome and completed in 1987. The Cydra 5 heterogeneous multiprocessor consisted of a general-purpose shared-memory multiprocessor based on Motorola

68020 processors along with a numeric processor that used Cydrome's Directed Dataflow™ architecture. The numeric processor achieved about one half the performance of a contemporary supercomputer at about one tenth of the price.

Discussion

Introduction

Cydrome (originally known as Axiom) was founded by chief architect Bob Rau, along with Arun Kumar, Ross Towle, Dave Yen, and Wei Yen. The key design goal for the Cydra 5 was to provide near-supercomputer levels of computing performance at departmental prices. The Cydra 5 was designed to accelerate existing unmodified FORTRAN applications commonly used for scientific computing. Goals included providing 50 (25) MFLOPS of peak single (double) precision floating-point performance and achieving greater computation efficiency by sustaining a higher fraction of peak performance than competing vector and multiprocessor architectures.

System Architecture

A primary feature of the Cydra 5 was the Directed Dataflow architecture that provided specialized hardware for parallel loop execution. The Cydra 5 supported compiler-directed parallelization of a very general class of loops, thus providing greater flexibility for accelerating existing codes than competing vector and multiprocessor architectures. The architecture combined the MultiConnect register file, predicated execution, and a specialized branch to support software-pipeline loops without code replication needed by other hardware architectures. In addition to supporting traditional vectorizable loops, this architecture efficiently supported a very general class of loops with recurrences, conditionals, arbitrary fixed stride, and randomly scattered memory references.

Software pipelining [1, 2] exploits parallelism by overlapping the execution of a sequence of loop iterations. When loop iterations are executed sequentially, latencies between the operations within each iteration limit loop parallelism. This problem is alleviated by initiating subsequent loop iterations before the completion of prior iterations. Software pipelining executes identical copies of the loop body at periodic intervals. An optimized periodic schedule is identified to minimize

the delay between successive loop iterations. The time delay between successive loop iterations is known as the Initiation Interval (II). The II measures the throughput of loop execution as one new loop iteration is completed every II cycles.

VLIW processors evolved from earlier microprogrammed processors that employed complex data paths to carry operands from registers to and from function units. The processing efficiency was greatly reduced by a number of obstacles. A key obstacle was the need to develop operation schedules that jointly accommodate all resource. Complex schedules often required a distinct and coordinated schedule for both function unit use and register use. Processors often contained multiple specialized register files that held operands as inputs to and outputs from function units. A single bus often supported multiple register files or multiple function units. This increased scheduling complexity and reduced efficiency, limiting program parallelism. Such machines could not be effectively exploited by compilers and were best programmed by clever humans. Another key challenge is to accommodate register lifetimes that often limit parallelism by introducing bottleneck dependences arising from register reuse. The lifetime of a variable formalizes the need hold a value for an extended period of time and is the period of time starting when a variable is computed and terminating on the variable's last use. A register cannot be reused to hold a new variable until the end of the prior variable's lifetime.

The Cydra 5 architecture addressed above difficulties using the Context Register Matrix (CRM) or Multi-Connect as it was called inside Cydrome. The CRM provided a number of features to allow compiler technology to automate the construction of compact and efficient software pipelines for a broad class of loops. The CRM evolved from the earlier PolyCyclic interconnect [3]. The polycyclic interconnect required the shifting of data as new values were created. This required custom memory structures and could not be implemented using high-performance RAM. The CRM integrated standard RAMs with novel register addressing units to provide similar benefits.

The Cydra 5 system contained multiple General-Purpose Processors (GPPs) and a numeric processor (NP) integrated as a heterogeneous multiprocessor running the Unix System 5 operating system. While

the primary role of the NP was to accelerate scientific applications, the GPPs were used to run nonnumeric applications. This architecture alleviated troublesome limitations of specialized attached processors such as the Floating Point Systems AP 120b [4] that ran numeric programs with no general-purpose or I/O capability. Such architectures required careful rewriting of applications to stream data from a main program running on the host into numeric kernel programs running on an attached processor. Unlike these machines, the Cydra 5 appeared as a single processor that allowed applications to be compiled either for the GPPs or for the NP.

The general-purpose processors provided an extensible general-purpose computing capability based on 16.67 MHz Motorola 68020 processors. Up to seven GPPs could be added to expand the general-purpose computing capability. In addition to hosting compilations needed for the Numeric Processor, the GPPs supported a large variety of conventional computing tasks under the Cydrix operating system which was derived from AT&T System V Unix.

Cydra 5 Numeric Processor

The numeric processor was a Very Long Instruction Word (VLIW) [5] processor implemented in Emitter Coupled Logic (ECL) gate arrays for very high performance. The NP achieved its 25-MHz clock speed using simple hardware to perform performance critical tasks and, where possible, relied on compiler and runtime software to implement complex operations that could be performed in software without excessive performance loss.

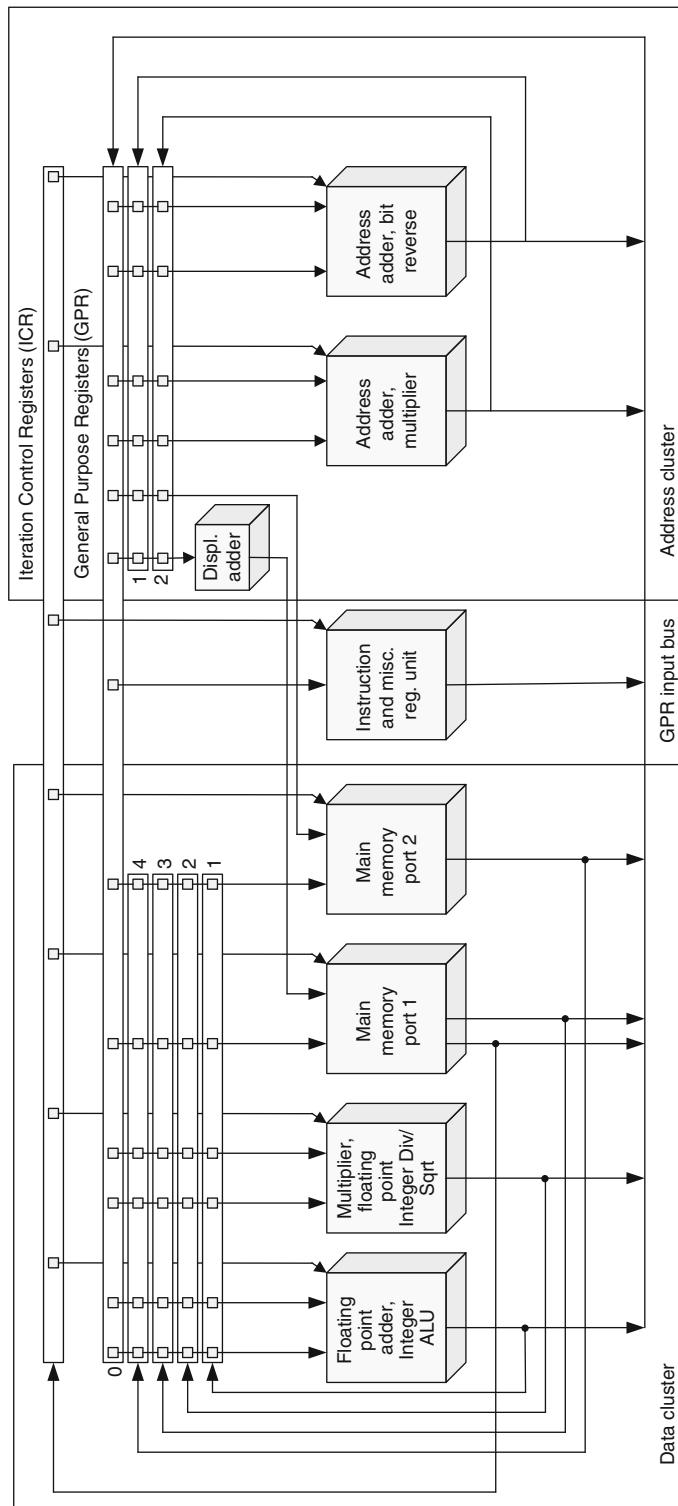
The numeric processor provided a 256-bit wide instruction container that was interpreted in either UniOp or MultiOP execution modes. In UniOp mode, the VLIW instruction was interpreted as six sequential instructions with explicitly encoded delays between those instructions as needed to satisfy interoperation dependences and resource constraints. The UniOp mode was used outside of high-performance loops and provided a relatively compact encoding for sequential code. In MultiOp mode the VLIW instruction was interpreted as a single instruction that allowed the issue of seven operations in a single cycle. Simultaneous operations included two arithmetic (floating or integer), two address, two memory, and one branch operation.

Multi-Op mode was primarily used to accelerate highly parallel innermost loop code but could also be used outside of accelerated loops when sufficient parallelism was available. The combination of the Cydra 5's wide instruction issue along with its deep pipelining produced large amounts of instruction-level parallelism. The total number of operations in flight (issued but not complete) could exceed 70 concurrent operations.

The Cydra 5 NP relied on explicit compile-time scheduling to eliminate complex hardware resource-dependence checking and arbitration. Hardware offered no instruction interlocks or ability to dynamically delay dependent operations, and the compiler was responsible for inserting explicitly encoded delays needed to enforce a correct program schedule. Instructions provided capabilities needed to program explicit delays. Resource usage was modeled with reservation tables [6] that encoded the pattern resources over a number of machine cycles relative to the time instruction issue. A machine description database formally defined the hardware's characteristics capturing both resource needs and latencies of all operations and precisely defined constraints that must be met for correct execution. This database was used by the compiler to ensure that all resource and dependence constraints are met.

The Cydra 5's NP datapath is shown in Fig. 1. This datapath was designed to sustain a peak bandwidth of two floating point operations on every cycle for important numeric loops. Two floating point units, two memory units, and two address units were needed to satisfy this goal. The machine executed six simultaneous non-branch operations each capable of reading two inputs and writing one output on each cycle. Twelve input ports and six output ports were needed to sustain full performance in parallel code regions. A single file with so many ports could not be constructed using available ECL gate arrays. In order to reduce complexity for providing needed register file ports, the Numeric Processor is divided into data and address clusters.

The CRM holds operands flowing into and out of all function units. Each row consists of cross-point registers that are replicated as a multiported file (indicated by small squares). When a value is written into a row, it is broadcast across the row so that cross-point registers hold identical values. Thus, each function unit has dedicated write access and can generate a new value



Cydra 5, Fig. 1 Cydra 5's NP datapath

into its dedicated row on every cycle. Columns provide distinct read access ports for two inputs from any row on each cycle.

Often, the length of variable lifetimes must be greater than the initiation interval. When a software pipeline is executed on a conventional horizontal architecture-computed operands are systematically overwritten, after looping to the next iteration, by the same operation that computed the operand on prior iterations. This overwrite often occurs even before the end of the operands lifetime. This problem can be alleviated by copying the operand before the operand is overwritten or by unrolling code and renaming registers. Each of these techniques has complex limitations.

The CRM addressed this problem using an Iteration Frame Pointer (IFP) that relocated loop-variant values on each iteration. Loop-variant register operands are addressed by adding a displacement from the instruction to the IFP modulo the register file size to determine the physical location of the selected register. The IFP is decremented every loop iteration to relocate subsequent register reads and writes. This allowed each operation in the loop body to generate overlapped sequences of register lifetimes that spanned multiple loop iterations. The compiler was responsible for allocating and tracking these values as loop execution progressed.

General-purpose registers (GPRs) were designed to solve a number of problems. Loop invariant values are values that are computed prior to loop execution and repeatedly referenced on each iteration. Loop invariants are supported using GRPs that are addressed as conventional registers and do not participate in register relocation. The use of separate address and data clusters limits the flow of operands between clusters. For example, a value loaded from memory and stored within its CRM row cannot be read directly by an operation in the address cluster. To provide flexible connectivity, all units can write or read the GPRs. GPRs are typically written outside of innermost loops and allow flexible communication between clusters.

Predicated Execution

Many loops have conditionals that preclude parallelization by previous vector and multiprocessor techniques. The Cydra 5 used predicated execution to allow the execution of nested if-then-else conditionals within a loop

body. If conversion is a code transformation that uses predicates to eliminate branches. After simple if-then-else code is if-converted, a compare computes complementary Boolean predicates for then and else clauses. Operations within each clause reference a predicate operand that provides the correct clause conditional. Each predicated operation executes only if its predicate input is true otherwise it performs no action. This correctly implements conditional actions without program branches. Predicated execution also allows the nested if-conversion of conditionals within conditionals. After if-conversion, conditional, and unconditional code is uniformly scheduled within a software pipeline. The execution of multiple if-then-else conditionals can be overlapped and scheduled as if the code were unconditional code.

Specialized Loop Control

Software pipelining produces code schedules that are periodic only after a prolog period during which the software pipeline is filled and before an epilog period during with the pipeline is drained [7]. Without specialized hardware, code replication is needed for both prolog and epilog code before a periodic kernel loop is reached. Efficient support for loops with small trip counts exaggerates the complexity of code generation and scheduling as prolog code may enter epilog code without ever reaching the kernel.

The Cydra 5 combined the CRM with predicated execution and a specialized branch to greatly simplify this problem. A *brtop* loop branch operation supported software pipelines. The *brtop* was itself a pipelined branch having two branch delay slots in which additional *brtop* operations might be scheduled. The *brtop* referenced a number of registers, including: the Loop Counter (LC), Epilog Stage Counter (ESC), the iteration frame pointer (IFP), as well as the Iteration Control Register (ICR) predicate file.

The structure of a software-pipeline loop executing on the Cydra 5 is shown in Fig. 2. Each row illustrates operations executed during II cycles of machine code execution. Operations from a single iteration of the original loop body are divided into lettered stages, each stage requiring exactly II execution cycles. The vertical (time sequential) execution of code within stages ABCD and E completes a single iteration of the source

Iteration number	Loop counter	ESC	Stages executed					ICR predicates		
initial	6	4						00001		
1	6	4	e	d	c	b	A	Prolog code	00001	
2	5	4		e	d	c	B	A	00011	
3	4	4		e	d	C	B	A	00111	
4	3	4			e	D	C	B	A	01111
5	2	4	Kernel code	E	D	C	B	A	11111	
6	1	4		E	D	C	B	A	11111	
7	0	4			E	D	C	B	A	11111
	0	3			E	D	C	B	a	11110
	0	2				E	D	C	b a	11100
	0	1				E	D	c	b a	11000
		0				E	d	c	b a	10000
									00000	

Cydra 5, Fig. 2 The structure of a software-pipeline loop

loop. These loop iterations are overlapped, and each subsequent copy of the actively executed loop body is displaced downward in time by II cycles. The example illustrates the execution of seven source iterations of a loop body having five stages in its software pipeline schedule. The execution of code stages is lettered and lower case indicates that a stage is nullified (executed with predicate 0). Upper case indicates that a stage is active (executes with predicate 1).

The number of stages measures the amount overlap among successive iterations. The kernel code region is shown in gray. In this region, horizontal (simultaneous issue) code from all five stages of the loop executes every cycle. Each kernel stage corresponds to a single loop iteration at the machine instruction level that executes the steady-state pattern in the heart of highly optimized loop. This is the period during which full parallelism is achieved.

Prior to entry in the kernel, the loop is in its prolog. New iterations are initiated every II cycles, but there are an insufficient number of prior iterations to fill the pipeline. In the first row or iteration of the machine code loop, only the first stage A of the first source iteration actively executes. In the second row, B from the first source iteration and A from the second source iteration execute simultaneously. As the pipeline fills, additional stages become active. The pattern of predicates shown on the right correctly enables corresponding stage code. During the epilog, old iterations continue to finish but no new iterations are initiated. Again the epilog pattern

is controlled using the pattern of ICR stage predicates shown on the right.

The brtop operation implements all actions needed to control such loops. Each execution of the brtop decrements the iteration frame pointer in order to relocate all register values (including predicates) to allow overlap of lifetimes. The brtop operates on both loop count (LC) and epilog stage count (ESC) registers to correctly calculate the branch decision and stage predicate needed for the next II cycles. While the LC counts the number of source loop iterations, the ESC counts additional iterations needed to fill the software pipeline. A newly calculated stage predicate shifts into view from the right-hand side every II cycles. Operations within each stage reference a corresponding stage predicate. Using this architecture, a single copy of the loop body executes highly optimized software pipelines.

Cydra 5 Memory

Instead of relying on a data cache for high bandwidth, the Cydra 5's NP provided full bandwidth to DRAM-based main memory. A 64-way interleaved memory provided up to 512 MB of storage, and two access ports allowed two memory operations on every cycle. Memory module collisions, memory refresh, and other difficult-to-predict events introduced nondeterministic delays in performance. However, the numeric processor provided no register interlocks to stall dependent operations for tardy input operands, and memory operands

needed to return at a statically scheduled time. This was accomplished using the memory collating buffers and the memory latency register (MLR).

Memory collating buffers allowed operands to return from memory early and out of order within the memory subsystem while those operands are delivered on time and in order to the NP. The collating buffer holds returning memory operands until the exact delivery time. The memory latency register allowed the programming of the latency time at which results from memory were delivered. If a memory result could not be delivered within the requisite time as defined by the MLR, the NP was stalled until that operand was available. From the viewpoint of the NP, all memory operands are delivered exactly on time.

A key memory system goal was to support the efficient access to data for programs having arbitrary stride or complex scatter-gather access patterns. Conventional interleaved memories suffer a large performance loss when the stride of the access pattern and the degree of memory interleave have common factors. For a worst-case stride 64 access into a 64-way interleave memory, only a single memory module is used and memory bandwidth drops significantly. The Cydra 5 uses a pseudo random address transformation [8] that randomizes the memory access pattern so that common address patterns appear as a random pattern to the memory. The memory is designed to deliver full bandwidth for such random patterns.

The lowest possible latency for The Cydra 5 NP memory is 17 cycles. For scalar code, the MLR was set to 17 to minimize the wait time for scalar operands arriving from underutilized memory that occurs out of loop. Within loops, the this minimal MLR value leads to reduced performance as collisions from memory reference sequences resulted in delayed memory access. An optimal value for loops was selected as 26 cycles which balanced needs for high bandwidth and low latency, especially for short trip count loops.

Exception Handling

The Cydra 5's exception handling followed the philosophy of relying on software rather than hardware for complex operations that not critical to performance. The Cydra 5 provided a Program Counter History Queue (PCHQ) to record the recent history of instruction addresses. Exceptions are not generated at the time

of instruction issue but instead occur a number of cycles after issue at specific exception latencies. For example, a memory exception is reported after a failed address translation, three cycles after operation issue. Hardware directs execution a software exception handler that provided a direct indication of the unit on which the exception occurred and the cause of the exception. However, hardware does not directly indicate the actual operation that caused the exception. This operation issued a number of cycles in the past and may have been followed by one or more intervening branches. Exception-handling software processes the PCHQ and uses known exception latencies to identify the exact operation that caused the exception. If needed, software recovers input operands from the input registers for the exception causing operation. This provides all inputs needed to process exceptions in software and resume execution.

Cydra 5 Compiler Technology

A large burden was placed on the compiler to identify and exploit large amounts of parallelism. After classical optimizations, the compiler was responsible for identifying all interoperation dependences to expose parallelism as a program graph. The graph was transformed to enhance parallelism in order to providing adequate parallelism for the NP. Then the compiler's scheduler created a schedule that packed operations into instructions and time. Specialized compiler technology for software pipelining was developed to exploit the NP hardware. While the Cydra 5 compiler also utilized specialized VLIW compilation techniques for scalar code, this is beyond the scope of this discussion and can be studied in more detail in [9, 10].

The Cydra 5 software-pipelining technology demonstrated optimal sustained performance for most numeric loops. Specialized loop optimization was applied to software-pipeline loops to transform code into an optimized form for software pipeline execution. The compiler used an intermediate form to represent loops in various optimization stages. The intermediate form used the concept of Expanded Virtual Registers (EVRs) to reference operands that have been computed in the current iteration or in previous iterations that may still be available for use in the current iteration. Thus, while an operand "t" represents a temporary computed in this iteration, t [1] represents the value of the same operand

as computed in the prior loop iteration. This intermediate form was used to express a number of loop optimizations, including load/store elimination, common sub-expression elimination, and height-reducing transformations needed to produce optimized code. Each of these optimizations reused values computed from prior iterations and EVRs precisely define this operand reuse.

Optimization is followed by scheduling for software pipeline execution. Performance bounds on the maximal execution rate were calculated and used to assist in loop scheduling. For most loops, loop schedules were identified that exactly match those bounds achieving optimal program schedules for long trip count loops. The Initiation Interval II measures the minimal time delay in cycles between successive iterations in a software pipeline. The II may be limited either by insufficient hardware resources or by the latency between dependent operations in successive iterations.

The limited availability of hardware resource provides a bound known as ResMII. Assume that a loop body has a number of operations n_t of a given operation type t (e.g., four operations of type memory in the loop body). Assume that hardware provides a number f_t of function units that execute operations of this type (e.g., two memory units). For operation type t , $res_t = n_t/f_t$ provides a resource bound for that type. This bound provides a lower bound on the initiation interval and an upper bound on the throughput for this loop. In this example, a loop body having four memory operations that executes on a processor having two memory units has $\Pi \geq 2$. For multifunction VLIWs, the largest res_t over all resource types provides the worst-case ResMII bound. The initiation interval measures a number of clock cycles as an integer quantity. When the ResMII resource bound is fractional, loop unrolling can often achieve the fractional bound. For example, if the ResMII = 2.5, two unrolled loop iterations may be executed every five cycles.

Dependences between operations in adjacent iterations also limit parallelism. A bound known as RecMII is calculated directly the program intermediate form. The intermediate form represents a program graph that captures dependences both within a loop iteration and between loop iterations. Inter-iteration dependences are marked with a distance that captures the number of iterations spanned by the dependence. Legal program graphs contain cycles that describe dependence

```

initialize:
a[1]
s[1]

do N times
    a = a[1]+4
    t1 = read(a)
    s = fadd(s[1],t1)

enddo

s = s[1]

```

Cydra 5. Fig. 3 Vector sum pseudo code

```

initialize:
  a[2],a[1]
  s[3],s[2],s[1]

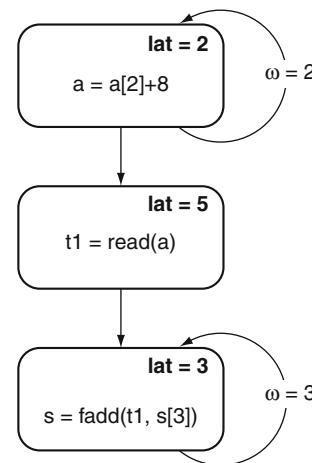
do N times
  a = a[2]+8
  t1 = read(a)
  s = fadd(s[3],t1)

enddo

u = fadd(s[1],s[2])
s = fadd(u,s[3])

```

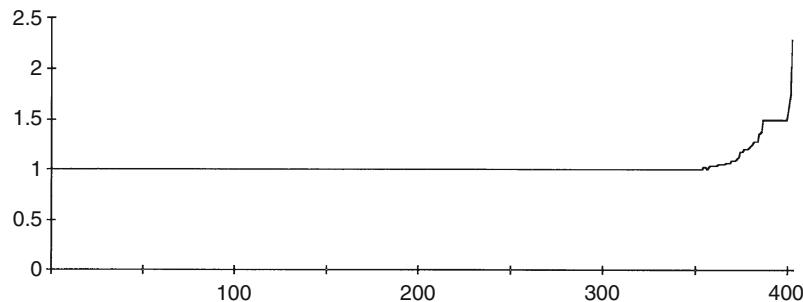
Cydra 5. Fig. 4 Height-reduced code



Cydra 5. Fig. 5 Loop graph for height-reduced code

relationships between operations within a single loop iteration as well as between iterations.

The Cydra 5 compiler applies height-reducing transformations to alleviate the performance limiting effects of inter-iteration dependences. Figure 3 illustrates vector sum pseudo code for the Cydra 5. To simplify the example, latencies of 2 for address add, and 3 for floating add replace actual Cydra 5 latencies of 3, and 4. Each operation in the loop body can be executed as a single



Cydra 5. Fig. 6 Achieved performance for over 400 loops

machine operation on the Cydra 5. The EVR references provide an iteration distance as a register reference $r[i]$ indicates the register value r from the i 'th prior loop iteration. A value produced in the current iteration $r[0]$ is abbreviated as r . In this non-height-reduced form, the floating point sum recurrence limits loop performance to one iteration every three cycles.

Figure 4 shows code after height reducing transformations [11] are applied to alleviate limitations from recurrences. First-order recurrences are replaced by second- and third-order recurrences for address and floating sums, respectively. While summation terms are reordered, a correct final value is reproduced as the same terms are finally summed. Figure 5 illustrates the program graph for the height reduced loop. Two recurrences are shown, and each limits performance by its path length divided by the iteration distance. After height reduction, a new loop iteration is initiated every cycle and, in the final loop schedule (not shown), all three operations are scheduled in a single machine instruction that executes every cycle.

The Cydra 5 loop scheduler relied on an accurate estimate of the desired II to begin the scheduling process. ResMII and RecMII bounds are calculated, and the larger of the two is used as the starting II for loop scheduling. The actual procedure to calculate RecMII is somewhat complex and beyond the scope of this discussion. After an initial II is determined, the scheduler uses a branch and bound search algorithm to explore the placement of operations within a cyclic schedule. Each newly scheduled operation is placed to satisfy both resource availability and interoperation dependences from previously scheduled operations. The scheduler backtracks when new operations cannot

be scheduled. Typically, the scheduler finds an optimal schedule in this process. However, sometimes, when all scheduling choices are exhausted or after excessive compile time is expended searching for a schedule, the II is raised to search for a lower performance loop schedule.

Figure 6 illustrates the performance of over 400 loops taken from Numerical Recipes, Linpack, and the Livermore FORTRAN Kernels. The performance each loop is plotted as a ratio of the II for the achieved loop schedule divided by the II that is estimated using the MAX(ResMII, RecMII) bound. Thus, a value of one indicates the loop scheduler has achieved optimal performance. Loops are sorted on order of increasing ratio of (achieved II)/(estimated II) and plotted. Note that over 350 of the loops achieve optimal performance. On the right, a single worst case loop having unusually complex resource patterns and dependences achieves a throughput 2.5 times slower than the desired bound.

Bibliographic Notes and Further Reading

At the time Cydra 5 was introduced, scientific computing was dominated by Cray corporation's vector processors [12]. Other key contemporary processors included the Alliant multiprocessor and the Convex vector processor. VLIW architectures were explored in research by Fischer [5], and this work resulted in the Multiflow architecture and compiler [13] that was contemporary with the Cydra 5. The Multiflow architecture relied on Trace Scheduling to identify and replicate important program code regions.



Bibliography

1. Lam M (1988) Software pipelining: an effective scheduling technique for VLIW machines. In: ACM SIGPLAN '88 conference on programming language design and implementation, Atlanta, pp 318–327
2. Rau BR (1994) Iterative modulo scheduling: an algorithm for software pipelining loops. In: Proceedings of the 27th annual symposium on microarchitecture, San Jose, pp 63–74
3. Rau BR, Glaeser CD, Greenawalt EM (1982) Architectural support for the efficient generation of code for horizontal architectures. In: Symposium on architectural support for programming languages and operating systems, Palo Alto, pp 96–99
4. Charlesworth AE (1981) An approach to scientific array processing: the architectural design of the AP-120B/FPS-164 family. Computer 4(9):18–27
5. Fisher JA (1983) Very long instruction word architectures and the ELI-512. In: Proceedings of the tenth annual symposium on computer architecture, Stockholm, pp 140–150
6. Davidson ES, Shar LE, Thomas AT, Patel JH (1975) Effective control for pipelined computers. In: Proceedings of the COMPON, IEEE, New York, pp 181–184
7. Rau BR, Schlansker MS, Tirumalai PP (1992) Code generation schema for modulo scheduled loops. In: Proceedings of the 25th annual international symposium on microarchitecture, Portland
8. Rau BR, Schlansker MS, Yen DWL (1989) The Cydra Tm 5 stride-insensitive memory system. In: Proceedings of the 1989 International Conference on Parallel Processing, The Pennsylvania State University, University Park
9. Dehnert JC, Towle RA (1993) Compiling for the Cydra 5. J Supercomput 7:181–227
10. Ellis JR (1986) Bulldog: a compiler for VLIW architectures. MIT Press, Cambridge, MA
11. Schlansker M, Kathail V (1993) Acceleration of first and higher order recurrences on processors with instruction level parallelism. In: Sixth international workshop on languages and compilers for parallel computing, Portland
12. Tang J, Davidson ES (1988) An evaluation of the Cray X-MP performance on vectorizable Livermore FORTRAN kernels. In: Proceedings of the 2nd International conference on Supercomputing, ACM, St. Malo
13. Lowney PG, Freudenberger SM, Karzes TJ, Lichtenstein WD, Nix RP, O'Donnell JS, Ruttenberg JC (1993) The multiflow trace scheduling compiler. J Supercomput 7:51–142



D

DAG Scheduling

- ▶ [Task Graph Scheduling](#)
- ▶ [Data Mining](#)

Data Analytics

- ▶ [Data Mining](#)

Data Centers

BABAK FALSAFI
Ecole Polytechnique Fédérale de Lausanne, Lausanne,
Switzerland

Synonyms

[Internet data centers](#)

Definition

A Data Center is a facility hosting large collections of servers that are physically co-located to facilitate one or more combinations of Internet connectivity, operation infrastructure (such as power delivery and cooling), management, and providing access security.

Discussion

Introduction

Large organizations (e.g., enterprises, governmental agencies, research and academic institutions) have historically used machine rooms to host and operate a collection of server- and supercomputer-class computing platforms. The machine rooms were designed to leverage the cost of building, hosting and operating a collection of high-performance computers.

The exponential growth in information technology made possible by advancements in semiconductor fabrication for the past several decades has been met or surpassed by the growth in demand for computing. The net result is that computing platforms hosted in machine rooms have become denser (e.g., have a higher computational and storage capability per occupied unit volume), and machine rooms have been growing in size, increasingly hosting large collections of server-class machines referred to as clusters or server farms.

Today, information technology is an indispensable pillar for a modern society in general and industry in particular. Companies from start-ups to multinationals all have IT departments which are large organizations with many machine room installations that provide computing as a service internally to the company. Because most of the activity within an IT department is centered around processing and storage of data belonging to an organization, machines rooms have evolved into entities that are increasingly referred to as Data Centers.

Today, Data Centers range in size from installations that have about thousands of processing cores and a few petabytes storage dissipating a few hundred kilowatts of electricity to hundreds of thousands of processing cores, hundreds of petabytes, and tens of megawatts of electricity.

Physical Layout

As in machine rooms, a Data Center typically refers to a physical site. A site can be a shipping-container-sized entity, a room, one or more floors of a building, or a number of collocated buildings. The minimal physical unit to which network connectivity, power, and cooling are delivered to is referred to by some vendors as a POD. A POD is a collection of racks of hardware that are collocated and sit adjacent to each other.

Containers are PODs that are optimized for transport and mobility and are sized exactly to be the size

of a shipping container. PODs in a room are sized for maximum energy efficiency.

Servers

The IT equipment in a Data Center consists of compute nodes or compute clusters (for processing), storage nodes or storage clusters (for storing), the network nodes (for communicating both within and to the outside), and a tape library (for backup). Compute, storage, and network hardware today appear in the form of rack-mounted platforms. Tape libraries often appear in the form of tightly integrated collection of tapes that are operated internally by robots. Nodes of the same type are often collocated due to the diverse cooling and reliability requirements of the node types. Node form factors continue to evolve based on improvements in fabrication technology, server density, and power/cooling delivery.

Hosting Infrastructure

Hosting infrastructure includes outside network connectivity, power delivery, cooling, and security (including both fire protection and secured access). The site must provide network connectivity commensurate to its size. Fortunately, with advances in communication technologies in line with improvements in processing speed and storage capacity, network bandwidth has kept pace with the demand on IT installation sites. Cooling and power delivery account for a substantial fraction of electricity in Data Centers. Power Usage Efficiency (PUE) is a metric that expresses the energy efficiency of a Data Center. PUE is the total electricity used by a site divided by the electricity used only by the IT equipment. PUEs in today's Data Centers can be as high as two indicating a 100% overhead in electricity to deliver cooling and power. Vendors are now marketing installations with PUEs as low as 1.3.

Historically, machines in a room have been air-cooled with computer room air conditioning (CRAC) units that cool the room and consequently the equipment. As Data Centers have become denser through the years and semiconductor technologies are reaching their limits in energy efficiency, designers are resorting to CRAC-less cooling technologies that are more energy-efficient. Modern PODs are airtight from all sides and have access panels/doors from the sides to

allow for serviceability. Today's low PUE installations circulate chilled water over rack tops and use forced air down to cool the racks. Modern cooling infrastructure and POD installations also obviate the need for raised floors which played a central role in air circulation in more conventional setups.

Power accounts for the second largest source of overhead after cooling. To prevent a single point of failure, all elements of the electrical system are replicated. This replication dramatically adds to the cost and electricity usage of the power delivery system. There are provisions for backup power that consist of universal power supplies and diesel generators.

Tiers

Data Centers have diverse reliability and availability requirements based on usage. As such, they are categorized into "tiers" ranging from Tier 1 with the least stringent reliability requirements to Tier 4 with the most stringent reliability requirements. Even within the same organization, applications may demand the usage of Data Centers from varying tiers. For instance, a telecommunications company may use a Tier 4 data center for high availability in their data network service and a Tier 1 data center with less stringent availability for bill processing. The various tiers' availability requirements are 99.671% for Tier 1, 99.741% for Tier 2, 99.982% for Tier 3, and 99.995% for Tier 4. The difference in the infrastructure (power delivery and cooling) cost between Tier 1 and Tier 4 can be a factor of two or higher [1].

Energy

The main scalability impediment to Data Centers in the future is energy, with both economic and environmental implications. Demand for Data Center scalability has shot up through the years, resulting in an unprecedented level of electricity usage in Data Centers. On the one hand, the demand for ubiquitous computing and data access by all computer users has skyrocketed. On the other hand, the cost of owning and operating IT equipment has caused many start-ups and smaller enterprises to buy processing and storage of data as a utility. The latter is loosely referred to as Cloud Computing. Cloud Computing is leading to unprecedented growth in server-side computing and Data Centers.

The Energy Star report to the US government indicates that Data Center electricity usage in the USA doubled from 2000 to 2006, and is slated to double again by 2011. The net result is that in 2012, operation cost in terms of electricity over the lifetime of volume servers will cost 50% more than the capital cost of purchasing the server in 2012. The carbon footprint of Data Centers in the USA in 2012 is slated to equal that of the airline industry.

While projections of conventional semiconductor fabrication technologies indicate scalability in chip integration levels and cost for another decade, chip energy efficiency has dramatically slowed down. The net result is tokened “the economic meltdown of Moore’s law” for servers and datacenters by Kenneth Brill of the Uptime Institute.

There are a number of short-term solutions to mitigate the Data Center energy efficiency problem. Idle power is quite high in servers, reaching up to 70% of peak power. Virtualization helps avoid idling by consolidating multiple servers into a single physical platform. Virtualization is quite effective in eliminating idle power, but soon reaches diminishing returns as utilization reaches 100%.

Cooling efficiency is another low hanging fruit. Modern installations use cold-isle rather than the conventional hot-isle setup where the racks are cooled inside the air-tight POD and the hot air is circulated out. Cooling efficiency can dramatically reduce electricity consumption. Improvements in overall energy consumption can come about for installations (e.g., in Europe) that tightly integrate outdoors air/water at cooler temperatures to help reduce cooling costs, and use the generated heat to warm buildings.

Microprocessors are increasingly incorporating a larger number of simpler processing cores. The latter are more energy-efficient but require an increase in the degree of parallelism in the software. A number of conventional server softwares are highly parallel and as such amenable to such organizations. Low parallelism in code and the diminishing returns on how simple cores can get will eventually limit energy efficiency in conventional microprocessor organizations.

Solid-state disks (SSD) are emerging as a cost- and energy-effective approach to augment the storage hierarchy. A layer of SSD between memory and disks can serve as an effective cache for the storage, filtering much

of the traffic to disks, requiring lower bandwidth disk configurations, and saving energy at the same time.

Future Directions

With regards to energy efficiency, there are a number of opportunities to exploit to scale Data Centers. As voltages in semiconductors chips have leveled off, while chip integration continues, designers must pursue solutions that minimize the number of joules per operation in a computing stack. To scale energy for another 10 years, the solutions must achieve two orders of magnitude of improvement in energy efficiency.

Mobile platforms historically have been designed for low power due to battery usage concerns. Servers must incorporate such technologies to minimize power without sacrificing the multitude of abstraction layers in modern server software. Specialization and vertical integration of lower-level software and hardware for common services in a server (e.g., OS, databases, web, search) are likely candidates to achieve such orders of magnitude in energy reduction.

Air cooling is highly inefficient as air is not an effective medium in absorbing heat. Conventionally, supercomputers have used liquid cooling to achieve much higher cooling efficiencies at albeit higher technology costs. Technologies, such as two-phase liquid cooling, are promising approaches to improve Data Center PUE’s to levels below 1.1.

To achieve optimal temperature and minimal cooling, thermal sensing, management, and cooling must be coordinated. Modern servers include internal knobs (e.g., voltage/frequency scaling) for thermal/power management. As servers are overprovisioned (to provide performance guarantees), modern servers do not exploit these mechanisms. Moreover, modern platforms include crude temperature sensing and management technologies that do not lend themselves well to accurate thermal management. There are proposals for integrated and accurate thermal management and cooling.

There are a number of proposals to reduce replication levels in the electrical system and power delivery while not compromising the availability. Such proposals dramatically reduce the electricity usage in the power delivery.

Related Entries

- Clusters
- Power Wall

Bibliographic Notes and Further Reading

The Datacenter as a Computer [2] is a short book that glosses over modern datacenter design issues and trends. The data center page on Wikipedia [3] includes more detailed information about conventional datacenter installations. The Uptime Institute [4] and the Energy Star [5] reports include careful analysis on energy costs and projections for future Data Center scalability. ITRS [6] includes white papers on technology projections.

Bibliography

1. Data Center Site Infrastructure Tier Standard: Topology (2010) Uptime Institute LLC
2. Barosso LA, Hölzle U (2009) The datacenter as a computer: an introduction to the design of warehouse-scale machines. Synthesis series on computer architecture. Morgan & Claypool, San Rafael
3. Wikipedia. http://en.wikipedia.org/wiki/Data_center
4. Brill K (2007) The economic meltdown of Moore's law. Uptime Institute and the Green Data Center. USENIX Keynote. http://www.usenix.org/event/lisa07/tech/brill_talk.pdf
5. US Environmental Protection Agency (2007) Energy star program, report to congress on server and datacenter energy efficiency. Public Law 109-341. <http://www.energystar.gov>
6. Yearly Reports. The international technology roadmap for semiconductors. <http://www.itrs.net/reports.html>

Data Distribution

SAMUEL MIDKIFF

Purdue University, West Lafayette, IN, USA

Definition

Data distribution refers to the process of spreading the elements of a data structure, typically an array, across two or more processors of a distributed memory machine. This is done to enable parallel computation on the distributed structure. Distributing an array across multiple processors leads directly to the problems of how to address the elements of the array within a processor, how to determine the processor on which some element of an array has been placed, and the related question of what elements a given processor owns. Topics related to distributing non-array data structures are not covered in this article. Moreover, the discussion is

restricted to arrays whose subscripts are affine expressions, and to distributions that can be described as affine expressions in one or two variables. These restrictions are important because they allow the problems above to be targeted with Diophantine equations while handling common data distributions and program idioms.

Overview

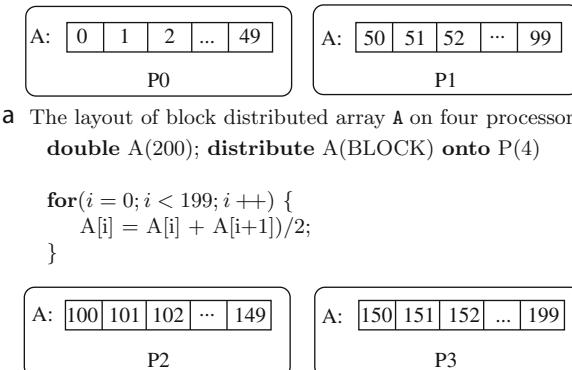
Computational Model and Notation

Figure 1a shows the array A *block* distributed across four processors. The distribution results in the block of elements 0...49 being placed on the first processor, elements 50...99 on the second processor, elements 100...149 on the third processor, and elements 150...199 on the fourth processor.

Given a set α of array elements whose values will be computed by a processor p , it is often the case that values not distributed onto p are needed to perform the computation. A popular way of determining the elements in α is the *owner computes rule* [11], i.e., each processor computes new values for the array elements that reside on it, i.e., that are *owned* by it. To compute the new values of array elements in α , it is necessary for the processor to receive from other processors the input values that reside on these processors. Thus in Fig. 1a, to compute A[49], processor p_0 must acquire the value of A[50] from processor p_1 . Because values are typically communicated via two-sided MPI primitives, it is also necessary for the processor that owns the needed values to know that it must send them. Thus, in the example, it is necessary for p_1 to send the value of A[50] to processor p_0 .

Notation Needed by Data Distribution

Let A be an array of rank D_A , that is A has D_A dimensions. For every dimension d , $0 \leq d < D_A$, the lower bound of A in dimension d will be zero, and the upper bound will be denoted U_{A_d} . The processors performing the computation can also be arranged as an array with rank D_P , using a similar notation. Individual processors will be described as elements in the processor grid, i.e., $p_{j_0, j_1, \dots, j_{D_P-1}}$ is element $P[j_0, j_1, \dots, j_{D_P-1}]$ in the processor grid P. For simplicity, in this article, one-dimensional processor grids will be assumed unless otherwise stated.



- a The layout of block distributed array A on four processors
double A(200); distribute A(BLOCK) onto P(4)

```

for(i = 0; i < 199; i++) {
    A[i] = A[i] + A[i+1])/2;
}
  
```

- b A loop performing a computation on A that is not distributed.

Data Distribution. Fig. 1 An example illustrating data distribution and related problems

References to arrays are typically contained in loop nests. The loops within a nest are subscripted by index variables named i_1, i_2, \dots, i_n , where the i subscript is the nest level of the loop, i.e., i_1 is the outermost loop. The index variable for an entire loop nest is called I , i.e., $I = [i_1, i_2, \dots, i_n]$. For each dimension d of array A, there is an affine index or subscript function $\sigma_{A_d}(I) = c_1 * i_1 + c_2 * i_2 + \dots + c_n * i_n + c_0$.

Upper (lower) bounds will be referred to as $U(L)$, respectively, with a subscript indicating the program structure (e.g., a loop, or a data array or processor grid) and the dimension whose bounds are being represented. Thus, U_{i_1} is the upper bound of the loop whose index variable is i_1 , L_{A_d} is the lower bound of dimension d of an array, and L_{A_d, p_j} is the lower bound of dimension d of the portion of the array A on processor p_j . The number of elements in an array (processor grid) dimension is $U_{A_d} + 1$ ($U_{P,d} + 1$).

The *block size* B_s is the number of elements of a block of an array dimension that is either block or block-cyclic distributed. As with upper and lower bounds, subscripts can be used to further identify the block being described. For each distribution, there is an affine function $\Delta_{A_d}(\cdot)$ in one or two variables that describes the distribution of elements of dimension d of array A onto a processor grid. A function $\sigma(\cdot)$ is an affine subscript function for some dimension of an array reference. Subscripts on σ (and other quantities) may be dropped if the meaning is clear from the context.

Local and Global Spaces

The literature often refers to the *local* and *global* spaces of an array. In the example of Fig. 1, the global space of the array contains all of the elements of the array, and valid index values are 0...199. The local space varies with each processor. Thus the local space of the array on p_0, p_1, p_2 , and p_3 is 0...24. In languages like Fortran 90 that support nonzero lower bounds and array sections, the distributed array elements on some processor can be addressed in the global space. In languages like C, C++, and Java, they must be accessed in the local space, and a translation analogous to what Fortran 90 does going from a reference in global space to accessing a locally distributed element must be performed by the programmer. For simplicity, in this article, it is assumed that all accesses are in global space, i.e., $A[25]$ is used to access the first element of the A array on processor p_1 in Fig. 1a.

Data Distributions

This section describes what elements of an array are placed on what processors, and how this placement can be represented by an affine function in one or two variables.

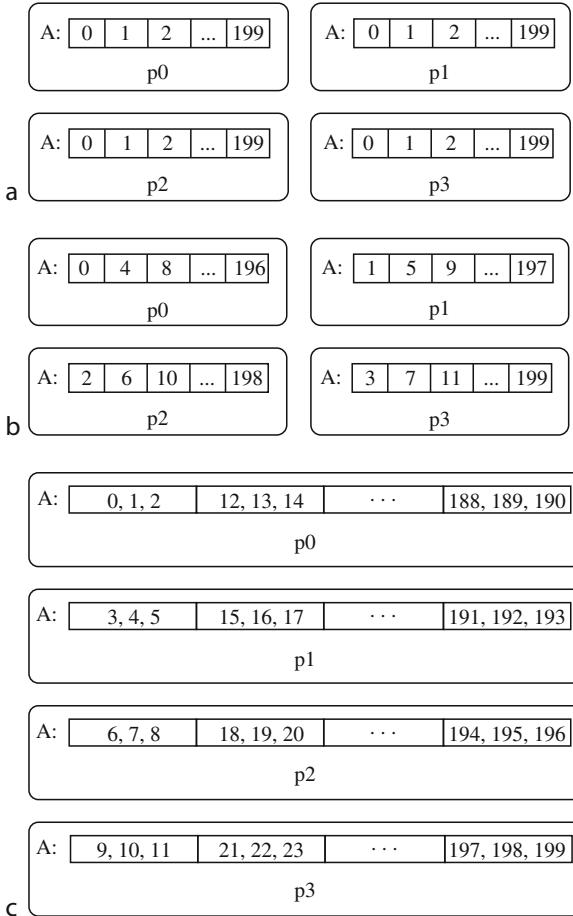
After discussing the simple one-dimensional case, the case of multidimensional data arrays being distributed onto multidimensional processor grids will be considered.

Replication of Data

The simplest form of data distribution is to *replicate* the structure across the different processors [8]. Thus, a copy of the distributed array is placed on each processor as shown in Fig. 2a. The function $\Delta(e) = 0 \leq e < U_A$ describes the elements of A contained on each processor.

Block Distribution

With a block distribution, *blocks* of contiguous array elements are distributed onto each processor [8], as shown in Fig. 1a. Given an array A of 100 elements block distributed onto four processors, the first processor would receive elements $A[0], A[1], \dots, A[24]$, the second processor would receive elements $A[25], A[26], \dots, A[49]$, and so forth. This distribution can be defined more formally. Given the $U_P + 1$ processors onto which an array dimension is distributed, the



Data Distribution. Fig. 2 Examples of replicated, cyclic, and block-cyclic distributions. (a) The layout of a replicated array A on four processors. (b) The layout of a cyclically distributed array A on four processors. (c) The layout of a block-cyclically distributed array A on four processors

number of elements on each processor (i.e., the block size for each processor) is $B_s = \lceil \frac{U_A+1}{U_p+1} \rceil$. Processor p_j will get the j th block, and thus the lowest element of A on p_j is $L_{A_d, p_j} = j \cdot B_s$. The highest element on p_j will be one less than the start of the block on the next processor, or $(j+1) \cdot B_s - 1$. To ensure that only elements in the declared array are distributed, the upper bound on p_j must be less than the maximum element in the array, and thus $U_{A_d, p_j} = \min(U_{A_d}, (j+1) \cdot B_s - 1)$. To determine what processor p_j contains, or owns, some element $e = A[\sigma_{A_d}(I)]$, it is only necessary to determine which

block of length B_s the element is in. The block containing element e is $\lfloor \frac{e+1}{B_s} \rfloor$. Finally, $\Delta_{A_d}(e) = L_{A_d, p_j} \leq e \leq U_{A_d, p_j}$.

Cyclic Distribution

With a cyclic distribution, elements of the array are distributed in a round-robin fashion as shown in Fig. 2b. Each processor p_j receives elements

$$A[j], A[j + U_p + 1], A[j + 2 \cdot (U_p + 1)], \dots \\ A[j + U_{A_d, p_j} \cdot (U_p + 1)],$$

where U_{A_d, p_j} , the number of elements distributed to processor p_j , is $\lceil \frac{U_A+1-j}{U_p+1} \rceil$. An affine function describing the elements of A that reside on some p_j is $\Delta_{A_d}(p_j) = (U_p + 1) \cdot (k - 1) + j, 1 \leq k \leq \lceil \frac{U_A+1-j}{U_p+1} \rceil$. The processor that owns element $A[\sigma_{A_d}(I)]$ is given by $\sigma_{A_d}(I) \bmod (U_p + 1)$.

Block-Cyclic

The block-cyclic (or *cyclic(Bs)*) distribution combines features of both block and cyclic distributions. Whereas the cyclic distribution (which is a block-cyclic distribution with $B_s = 1$) spreads single elements of an array across the processors in a round-robin fashion, the block-cyclic distribution spreads blocks of contiguous array elements across the processor grid. With block-cyclic, B_s is the programmer specified size of a block, and $num_{Bs} = \lceil \frac{U_A+1}{B_s} \rceil$ is the number of blocks in the array. To find the elements contained by processor p_j , it is necessary to first find the blocks that are contained by p_j . A processor p_j contains $num_{Bs, p_j} = \lceil \frac{num_{Bs}-j}{U_p+1} \rceil$ blocks. Note that the formula for the number of blocks is similar to that for the number of elements on a processor with the cyclic distribution. Number the b blocks as $1, 2, \dots, num_{Bs, p_j}$. Then the first element in block b of array A_d is $L_{A_d, b, p_j} = B_s \cdot j + B_s \cdot (U_p + 1) \cdot (b - 1)$. Thus, the set of doubles

$$\{ [max(L_{A_d}, L_{A_d, b, p_j}) : min(L_{A_d}, L_{A_d, b, p_j} + Bs - 1)] | \\ 0 \leq b \leq num_{Bs, p_j} \}$$

describe the elements of the A on processor p_j .

The elements contained on a processor p_j for the replicated, cyclic, and block distributions can be described as an affine function in one variable because the distance between the e 'th and $e + 1$ 'th element of an array residing on p_j , in the global space of A , is constant and independent of the value of e . For block-cyclic, this

is not true – if elements e and $e + 1$ are within a block, the distance between them is one, and if e and $e + 1$ are in different blocks, the distance is at least $(U_P + 1) \cdot Bs$. This means that the distribution cannot be described as an affine function in one variable, a situation that caused much difficulty in handling block cyclic distributions in HPF [5, 9] in the early 1990s. However, once it is realized that the distribution can be described as an affine function in two variables [7, 10, 12], it is possible to manipulate these distributions as described above. One variable, b , moves from block to block, and the other, e , traverses the elements within a block.

Distributing Data Along Multiple Dimensions

The distribution of multidimensional array A onto multidimensional processor grid P is now described. There exist three cases: (1) the array rank is greater than the processor rank; (2) the array and processor rank are equal; and (3) the processor rank is greater than the array rank.

In the first case, D_P dimensions of the array are distributed onto the processor grid, and $D_P - D_A$ dimensions are replicated, as shown in Fig. 3a. In the second case shown in Fig. 3b, $D_A = D_P$ dimensions of the array are distributed onto the processor grid. Figure 3b shows array rows (columns) distributed onto processor rows (columns), and Fig. 3c shows array columns (rows) distributed onto processor rows (columns). In the third case, D_A dimensions of the array are distributed onto D_P dimensions of the processor grid, and the distributed array is spread (replicated) across the remaining $D_P - D_A$ dimensions of the processor grid, as seen in Fig. 3d.

The computations required for determining what elements along a dimension a processor might own can be computed independently for every dimension. Given an element of an array, it resides on a processor if the index for that element along every dimension of the array resides on the processor.

Trade-Offs of Different Distributions

When a value is needed on every processor, and the computation of the value costs significantly less than communicating it (e.g., via a *broadcast* or complicated point-to-point messaging), the best solution is often to compute the value on every processor. This requires replicating the data across the processors.

a	A[0:3, 0:24]	A[4:7, 0:24]	A[8:11, 0:24]	A[12:15, 0:24]
	A[0:3, 0:3]	A[4:7, 0:3]	A[8:11, 0:3]	A[12:15, 0:3]
	A[0:3, 4:7]	A[4:7, 4:7]	A[8:11, 4:7]	A[12:15, 4:7]
	A[0:3, 8:11]	A[4:7, 8:11]	A[8:11, 8:11]	A[12:15, 8:11]
b	A[0:3, 12:15]	A[4:7, 12:15]	A[8:11, 12:15]	A[12:15, 12:15]
	A[0:3, 0:3]	A[0:3, 4:7]	A[0:3, 8:11]	A[0:3, 12:15]
	A[4:7, 0:3]	A[4:7, 4:7]	A[4:7, 8:11]	A[4:7, 12:15]
	A[8:11, 0:3]	A[8:11, 4:7]	A[8:11, 8:11]	A[8:11, 12:15]
c	A[15:12, 0:3]	A[12:15, 4:17]	A[12:15, 8:11]	A[12:15, 12:15]
	A[0:3]	A[4:7]	A[8:11]	A[12:15]
	A[0:3]	A[4:7]	A[8:11]	A[12:15]
	A[0:3]	A[4:7]	A[8:11]	A[12:15]
d	A[0:3]	A[4:7]	A[8:11]	A[12:15]

Data Distribution. Fig. 3 Examples of multidimensional distributions. (a) Distribution of an array onto a processor grid with fewer dimensions. (b) Distribution of an array onto a processor grid with the same number of dimensions. (c) Distribution of rows (columns) of an array onto columns (rows) of a processor grid. (d) Distribution of an array onto a processor grid with a larger number of dimensions

A cyclic distribution is often used in situations where the amount of work being done on each processor may change from iteration to iteration, as is the case with triangular loops and some of sparse matrices. Using a cyclic distribution spreads consecutive iterations across the processors and leads to better load balancing at the cost of increased communication when adjacent array elements (in the global space) are needed. A block distribution can lead to a greater load imbalance but less communication. The block-cyclic distribution is a hybrid of the two with the benefits of both at the cost of more complex code to access elements of the array and to determine loop bounds.

Iteration Space Partitioning

After data has been distributed onto processors, it is necessary to partition, or shrink, the iteration space of loops that execute on each processor to only access the elements of the array that are owned by the processor. To partition the iteration space for a processor p_j , information is needed about the distribution, original loop bounds, and the subscript that is accessing the array. For the distribution, the information required is the first and last elements of the array that are distributed

onto the processor (U_{A,p_j}, L_{A,p_j}) , the block-size B_s , the number of processors $(U_p + 1)$, the affine function Δ describing elements of A that are present on a processor, and the type of the distribution. For simplicity, the present discussion will restrict itself to subscript expressions that are affine functions in one variable, i , i.e., $e = \sigma(i)$. This restriction is later lifted, and the general case of multiple references in a loop is also discussed.

Partitioning on Subscript Expressions in a Single Variable

Partitioning the loop bounds logically proceeds in two steps. First, the bounds are initially shrunk to encompass the range of elements that are owned by the processor. Next, the elements resident on a processor that are accessed by the subscript are identified, and the range of the index variable is modified to only access those elements.

Initial bounds partitioning. The initial partitioning gives lower and upper bounds of the loop index i on processor p_j for some reference $A[\sigma(I)]$ as

$$L_{i,p_j,\sigma}^{\text{init}} = \max(L_i, \min(\sigma^{-1}(L_{A,p_j}), \sigma^{-1}(U_{A,p_j})))$$

and

$$U_{i,p_j,\sigma}^{\text{init}} = \min(U_i, \max(\sigma^{-1}(L_{A,p_j}), \sigma^{-1}(U_{A,p_j}))),$$

respectively. The function σ^{-1} is the inverse of the subscript function, i.e., $e = \sigma(\sigma^{-1}(e))$. Thus the lower bound of the partitioned loop i must be at least as great as the least of the lower and upper bounds of the original loop. Moreover, for a monotonically increasing subscript value, it should be no less than the value needed to access the lowest numbered element of the array partition owned by the processor. The explanation of the expression for monotonically decreasing subscript values and for the initial loop upper bound is similar.

Finding elements on p_j accessed by $\sigma(i)$. The technique above is not sufficient to find the bounds of the loop. Why this is true can be seen in the example of Fig. 2a. In the example, some processors only contain odd elements of A, and some only contain even elements. If the array is accessed with $\sigma(i) = 2 \cdot i$, the processors containing only odd elements of A will not access any elements.

To determine precisely which elements are accessed, it is necessary to intersect the set of elements accessed by the subscript for each reference, as specified by σ and

the bounds computed above, and the list of elements on the processor, specified by Δ and its bounds. This is done by solving the Diophantine equation $\sigma = \Delta$. If there is no solution to the equation, then no elements of A accessed by the subscript are owned by the processor. If there exist solutions, then for each variable x_1, x_2, \dots, x_n there is a parametric equation $x = \kappa_1 + \kappa_2 \cdot t$ [1]. For block and cyclic distributions, the variable i used in $\sigma(i)$ is the loop index, and the variable (k for cyclic, b for block) for Δ specifies the different elements. Solving $L_{i,p_j,\sigma}^{\text{init}} \leq \kappa_1 + \kappa_2 \cdot t \leq U_{i,p_j,\sigma}^{\text{init}}$ gives the range of the t needed to generate the needed values of the x that corresponds to the index variable i . Plugging these values of t into the formula above gives the range of i values, and therefore the bounds of i for the loop on each processor.

For block-cyclic distributions, the same technique is used, but the Δ function has two unknowns, and consequently the i loop must be logically replaced by a pair of nested loops that iterate over the elements of the array.

Partitioning on Subscript Expressions in Many Variables

It is often the case that a subscript function will be a function of two or more index variables in a loop nest. Consider a loop nest of depth n , and the analysis of a subscript expression

$$\begin{aligned} \sigma(I) &= c_1 \cdot i_1 + c_2 \cdot i_2 + \dots + c_{q-1} \cdot i_{q-1} \\ &\quad + c_q \cdot i_q + c_{q+1} \cdot i_{q+1} + \dots c_n \cdot i_n + c_0 \end{aligned}$$

to partition the bounds of the i_q loop. At runtime, when the i_q loop begins to execute, values of the i_1, i_2, \dots, i_{q-1} loop indices are known, and the sum of products of these and their coefficients can be folded into the constant c_0 , giving $c'_0 = c_0 + \sum_{j=1}^{q-1} c_j \cdot i_j$. For the index variables i_{q+1}, \dots, i_n , the maximum and minimum values of the function

$$\sigma'(i_{q+1}, \dots, i_n) = c_{q+1} \cdot i_{q+1} + \dots c_n \cdot i_n$$

in the iteration space of the i_{q+1}, \dots, i_n index variables can be found. The minimum and maximum values that σ' can have are found for each processor by evaluating it at upper and lower bounds of the iteration space on that processor, and intersecting it with the array elements owned by the processor. Once these bounds are determined, the lower bound can replace the terms

i_{q+1}, \dots, i_n in σ to find the lower bound as described in section “Partitioning on Subscripts Expressions in Single Variables”, and the upper bound can be used to replace the terms to find the upper bound.

Finding Iteration Spaces for Loops with Multiple References

Let $L_{A_1,i}, L_{A_2,i}, \dots, L_{A_r,i}$ be the lower bounds on i for the r array references, $U_{A_1,i}, U_{A_2,i}, \dots, U_{A_r,i}$ be the upper bounds, and $S_{A_1,i}, S_{A_2,i}, \dots, S_{A_r,i}$ be the strides. (The strides result from the solutions of the parametric equations described in section “Partitioning on Subscripts Expressions in Single Variables”.) The final iteration space must contain all of the iterations needed by all of the references. Thus, the lower bound of the resulting loop is $\max(L_i, \min(L_{A_1,i}, L_{A_2,i}, \dots, L_{A_r,i}))$, the upper bound is $\min(U_i, \max(U_{A_1,i}, U_{A_2,i}, \dots, U_{A_r,i}))$, and the stride is $\gcd(S_{A_1,i}, S_{A_2,i}, \dots, S_{A_r,i})$. Because the iteration space contains, in general, more iterations than needed by any single reference, each computed reference must be guarded by an *if* statement that checks if the current value of i is within the upper and lower bound for that reference, and if the stride is a divisor of the stride for a reference, as shown in Fig. 4.

Communication Sets

If cross-iteration dependences exist in the program, communication among processors is necessary to transfer data that is needed by processor p_r but is owned by another processor p_s . With the bounds of the partitioned loop on some processor p_r , the elements of a read array reference accessed by p_r can be determined and expressed as an affine function. The affine function describing the read elements is simply the subscript expression σ for the reference, bounded by the upper and lower bounds of the partitioned loop. By intersecting this affine function with the affine functions Δ describing elements of the array owned by each processor, the elements that must be sent from each processor to other processors can be determined. This intersection is the solution of the Diophantine equation $\sigma = \Delta$.

Naively, this intersection must be performed by every processor. However, compiler analyses can determine communication patterns for pairs of references [3, 8]. These patterns will often limit the number of processors for which communication sets must be computed.

```

for (i = max(L_A, min(L_{A_1,i}, L_{A_2,i}, ..., L_{A_r,i})),
      i ≤ min(max(U_{A_1,i}, U_{A_2,i}, ..., U_{A_r,i})),
      i+ = gcd(S_{A_1,i}, S_{A_2,i}, ..., S_{A_r,i}) {
  if (i ∈ max(L_A, L_{A_1,i}) : min(U_A, U_{A_1,i}) : S_{A_1}) {
    A_1 = ...
  }
  if (i ∈ max(L_A, L_{A_2,i}) : min(U_A, U_{A_2,i}) : S_{A_2}) {
    A_2 = ...
  }
  ...
  if (i ∈ max(L_A, L_{A_r,i}) : min(U_A, U_{A_r,i}) : S_{A_r}) {
    A_r = ...
  }
}

```

Data Distribution. Fig. 4 Example of code needed with multiple computed references to distributed arrays in a loop

Bibliography

1. Banerjee U (1993) Loop transformations for restructuring compilers: the foundations. Kluwer, Norwell
2. Banerjee UK (1996) Dependence analysis. Kluwer, Norwell
3. Gupta M, Banerjee P (1992) A methodology for high-level synthesis of communication on multicomputers. In: ICS ’92: Proceedings of the 6th international conference on supercomputing, ACM Press, New York, pp 357–367
4. Gupta M, Midkiff S, Schonberg E, Seshadri V, Shields D, Wang K-Y, Ching W-M, Ngo T (1995) An HPF compiler for the IBM SP2. In: Supercomputing ’95: proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM), ACM, New York, p 71
5. Kennedy K, Koelbel C, Zima HP (2007) The rise and fall of High Performance Fortran: an historical object lesson. In: Proceedings of the third ACM SIGPLAN history of programming languages conference (HOPL-III), pp 1–22
6. Kennedy K, Kremer U (1998) Automatic data layout for distributed-memory machines. ACM Trans Program Lang Syst 20(4):869–916
7. Kennedy K, Nedeljkovic N, Sethi A (1996) Communication generation for cyclic(k) distributions. Kluwer, Troy, New York, pp 185–197
8. Koelbel C, Mehrotra P (1991) Compiling global name-space parallel loops for distributed execution. IEEE Trans Parallel Distrib Syst 2(4):440–451
9. Koelbel CH, Loveman D, Schreiber RS (1993) The High Performance Fortran handbook. MIT Press, Cambridge
10. Midkiff SP (1995) Local iteration set computation for block-cyclic distributions. ICPP 2:77–84
11. Rogers A, Pingali K (1989) Process decomposition through locality of reference. In: Proceedings of the 1989 ACM conference on programming language design and implementation, Portland, OR, pp 69–80

12. Wang L, Stichnoth JM, Chatterjee S (1996) Runtime performance of parallel array assignment: an empirical study. In: Supercomputing '96: proceedings of the 1996 ACM/IEEE conference on supercomputing (CDROM), IEEE Computer Society, Washington, DC, p 4

Data Flow Computer Architecture

JACK B. DENNIS

Massachusetts Institute of Technology, Cambridge, MA, USA

Definition

Data Flow Computer Architecture is the study of special and general purpose computer designs in which performance of an operation on data is triggered by the presence of data items.

Discussion

Introduction

This article discusses several forms of data flow architecture that have been studied in university research groups and industrial laboratories beginning around 1974 [8]. The architectures covered all use some form of data flow graph programming model to enable the exploitation of parallelism.

In its original form, data flow architecture envisioned single instructions operating on individual data elements, integers or floating point numbers for example, as the units of computation, and this has characterized most of the design proposals and projects. Other designs have used blocks of instructions or modules of code as the independently executed unit.

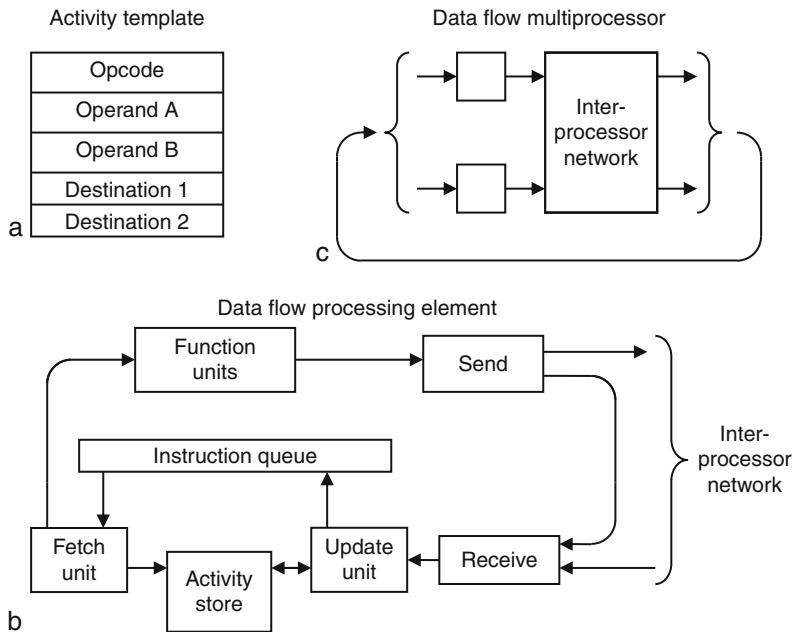
Two major lines of development of data flow architecture have been pursued. In *static* data flow machines, the hardware holds a single unchanging representation of a data flow graph loaded prior to program execution. In *dynamic* data flow, means are incorporated so that computation proceeds as though copies of a data flow graph were generated during program execution, providing direct support of arbitrarily nested function activations. A general form of dynamic data flow also permits many instances of a loop body to have effectively separate data flow representations for each cycle of loop execution so that overlapped execution may occur.

The following sections discuss static and dynamic data flow architecture, then data structures in data flow computers, and multithreading architectures inspired by data flow principles.

Static Data Flow Architecture

The basic scheme of a static data flow multiprocessor is illustrated in Fig. 1. A data flow program is stored in the machine as a collection of Activity Templates divided among the several processing elements and held in the Activity Store of each. The operand fields of Activity Templates are placeholders for operand values expected as results of executing other instructions; destination fields specify the target Activity Template, which may reside locally or in a remote processor, and the role that the result plays at the target. The Instruction Queue contains Activity Store addresses of instructions, for which operands needed for execution have been placed in the corresponding Activity Template, and delivers them to the Fetch Unit. In turn, the Fetch Unit accesses the completed Activity Template and forwards it to a Function Unit appropriate for the requested operation. When a result is obtained, the Function Unit constructs one or more Result Packets, each consisting of a copy of the result value and one destination address from the Activity Template, and passes these to the Send Unit. The Send Unit either directs the packet through the Interprocessor Network to the Update Unit of a remote processing element, or delivers the packet to the local Update Unit through the Receive Unit. The Update Unit enters the result in an operand field of the target Activity Template, and checks whether all needed operands are present. If so, it enters the address of the Activity Template in the Instruction Queue. Conditional and iteration program elements may be implemented using relational instructions that produce Boolean control packets; the Update Unit may use such a control value to choose which of two operands a Function Unit should use. Reference [9] gives details of one proposal.

Although implementations of static data flow have a variety of forms, the circular pipeline structure evident in Fig. 1 is a common feature, and supports the continuous processing of several instructions simultaneously. Static data flow concepts are well-suited to signal processing applications and to scientific computation using linear algebra methods, especially when array sizes are known at the start of computation. Several industrial



Data Flow Computer Architecture. Fig. 1 Scheme of a static data flow multiprocessor

projects have constructed static data flow processors, and the principles have found use in signal processing applications.

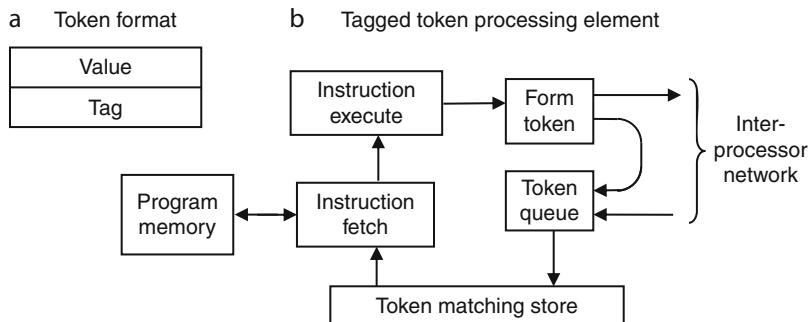
Dynamic Data Flow Architecture

In a dynamic data flow computer, the values passed between actors of a data flow program graph are tagged to indicate which instance of a variable is represented. To handle arbitrary nests of function calls and unknown loop iteration counts, the number of distinct value instances existing in any snapshot of an ongoing computation is unknown until after the computation has begun. Hence, the information in the tag of a value cannot be coded in the structure of the (finite) graph. The tag of a value must define to which function activation the value pertains, and to which cycle within each nested loop of the function body the value applies. The semantics of an execution model known as the *unraveling interpreter* were published in 1977 [1, 12] and supplied the initial impetus for dynamic data flow.

Figure 2 shows the general scheme of a dynamic data flow multiprocessor. The dynamic data flow scheme has similarities to the static scheme: tokens carrying data values flow in a circular pipeline structure. Differences

lie in the way instructions are stored and activated. In dynamic data flow, operands cannot be held with the instruction that operates on them. Instead, a special component, called the Token Matching Store, is used to hold operand values that must wait for a second operand to arrive before an operation on the pair may be performed. Then the Matching Store forwards the pair of values to Instruction Fetch, which accesses the instruction and passes the combination to Instruction Execute. The Form Token unit creates a token using the computed result value, and the operand tag and destination address passed through from Instruction Fetch. Tokens are either entered in the local Token Queue or dispatched to a remote processor according to their tag values.

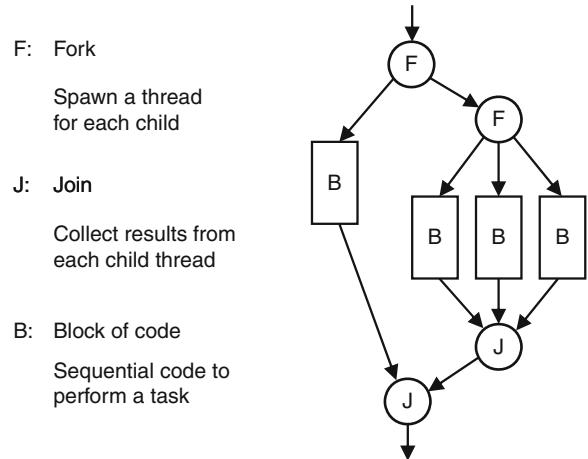
A pioneering implementation of dynamic data flow was built at Manchester University, England [13], and it used a hardware associative memory for the Token Matching Store. Several variations on the scheme of Fig. 2 have been developed at research institutions, some with industrial support. The variations are mainly concerned with replacing the associative memory used in the Manchester University machine with innovative schemes that are substantially more cost-effective.



Data Flow Computer Architecture. Fig. 2 Scheme of a dynamic data flow multiprocessor

Data Structures in Data Flow

Performing computations involving structured data such as arrays of values presents a challenge for data flow architecture because it seems wasteful to carry a complete data structure from a producer actor to a consumer. On the other hand, if the data structure is held separately from the data flow graph, then hazards can arise from races between conflicting memory operations. One solution to this quandary is to use special data structures known as *I-Structures* for which a single write order sets a value and several synchronized reads may safely access it [2]. Other approaches involve partial *copy-on-write* operations, and special data parallel treatment of array construction operations.



Data Flow Computer Architecture. Fig. 3 Multithread tasking scheme inspired by data flow concepts

Data Flow Ideas in Multithreading

Several workers familiar with data flow principles have employed data flow concepts in the construction of systems able to multithread large numbers of concurrent activities with a guarantee of determinate execution. The principles used are illustrated in Fig. 3, where the links indicate flow of control (threads). Here four code blocks are arranged to execute concurrently through the use of **fork** and **join** operators. It is expected that a computer system will schedule threads to execute the code blocks as short sequential program segments. In execution, the code blocks may have internal variables, but all shared data is read-only. At a **fork** operator, the parent thread makes specific data objects accessible to each child task. At a **join**-operator result values produced by each child are combined and made available to the continuation thread. Several variations of this scheme have been implemented for shared-memory multiprocessor systems, and provide a guarantee of determinate

behavior so long as the restriction of code segments to read-only shared data is honored.

Future Prospects

Currently, increased attention is being given to multithread computer architectures inspired by data flow principles to yield higher performance by dynamically scheduling sequences of instructions as independent units of computation [17]. It is known how to build static dataflow computers and streaming processors that can perform well. The extension to data structures and dynamic control structure, such as recursive functions, has remained a challenge for demonstrating levels of performance that would attract widespread interest.

Because dataflow graphs are a determinate representation for parallel programs that do not make essential use of nondeterminacy, it is feasible to build

computers using data flow principles that can provide the user with a guarantee of determinate operation, while achieving highly parallel computation. This was accomplished in the Barton/Davis DDM1 machine [6, 7]. However, reluctance to adopt the functional programming style favored by data flow architecture has limited the prospects for this achievement.

Related Entries

- ▶ [Data Flow Graphs](#)
- ▶ [Determinacy](#)
- ▶ [Functional Languages](#)
- ▶ [Multi-Threaded Processors](#)

Bibliographic Notes and Further Reading

Data flow computer architecture was inspired by the prospect of using data flow graphs as a programming model, thereby permitting exploitation of parallelism to be readily accomplished. The first proposals for the hardware organization of dataflow computers were published in 1974 [8] and 1975 [9], originating in the Computation Structures Group of MITs Project MAC. The earliest working model of a computer using data flow principles was the DDM1 designed by Al Davis and Robert Barton at the University of Utah, built at the Burroughs Corporation and described in 1978 [6, 7]. It was a very innovative machine, supporting a dynamic programming model including a tree-structured heap for memory objects. It relied on the serial processing of text packets representing instructions and data.

Static data flow architectures were developed as a direct implementation in hardware of the data flow graph program model [8, 9]. Experimental static data flow computers were built by Texas Instruments [4], the ESL division of TRW [16], Hughes [26], and in France [3].

In Japan, NEC conducted two significant static data flow projects. The NEDIPS processor was developed in the Radio division for processing radio astronomy data [24]. The second project developed a data flow processing chip that could be used in cascade to implement signal processing functions and was envisioned to be the basis for an advanced photo copying machine that could scale and rotate digitized images [10, 18]. Many other Japanese manufacturers also built experimental data flow machines; Veen [27] provides details.

The best known early project to build a machine using the tagged token principle was the Manchester University data flow computer that introduced the waiting-matching store [13]. Since then the principle has been developed to introduce more efficient realizations in projects in Japan and at MIT in the USA. The most advanced and successful of these are the Sigma 1 project at the Electrotechnical Laboratory in Japan [14, 15], and the Monsoon project at the MIT Laboratory for Computer Science [20, 21]. These projects have demonstrated the ability of data flow architecture to exploit parallelism in important codes for scientific computation.

Multithreading projects inspired by data flow concepts include the Threaded Abstract Machine [5], and others described in [11]. A recent revival of interest in applying dataflow principles in computer architecture has led to projects at University of Texas, Austin [22] and the University of Washington, Seattle [19].

The 1991 volume edited by Gaudiot and Bic [11] includes chapters written by authors personally involved in many of the data flow research efforts at its time of publication. Good treatments of multithreaded computer architectures and their relation to data flow research may be found in [17]. The papers collected in [25] describe early work on advanced programming models in the context of sequential computers that foresees later work on dynamic data flow. The book by Sharp [23] provides a summary of approaches to data flow computer architecture and includes a good bibliography of early work in the field, and the Veen survey [27] explains details of the many dataflow hardware projects of the late 1970s and 1980s.

Bibliography

1. Arvind, Gostelow KP (1982) The U-interpreter. IEEE Computer February:42–49
2. Arvind, Nikhil RS, Pingali KK (1989) I-structures: data structures for parallel computing. ACM Trans Program Lang Syst 11(4): 598–632
3. Comte D, Hifdi N, Syre J (1980) The data driven LAU multiprocessor system: results and perspectives. In: Proceedings of IFIP congress 1980, Tokyo, Japan, Oct 1980, pp 175–180
4. Cornish M (1979) The TI data flow architectures: the power of concurrency for avionics. In: Proceedings of the third conference on digital avionics systems. IEEE, New York, pp 19–25
5. Culler DE, Sah A, Schausler KE, von Eicken T, Wawrzynek J (1991) Fine-grain parallelism with minimal hardware support: a compiler-controlled threaded abstract machine. In: Proceedings of the fourth international conference on architectural support for

- programming languages and operating systems. ACM, New York, pp 64–175
6. Davis AL (1978) The architecture and system method of DDM1: a recursively structured data driven machine. In: Fifth international symposium on computer architecture, pp 210–215
 7. Davis AL (1979) DDM1. In: AFIPS conference proceedings 49, pp 1079–1086
 8. Dennis JB, Misumas DP (1974) A computer architecture for highly parallel signal processing. In: Proceedings of the ACM national conference, New York, NY. ACM, New York, pp 402–409
 9. Dennis JB, Misunas DP (1975) A preliminary architecture for a basic data-flow processor. In: Proceedings of the second annual symposium on computer architecture. ACM, pp 126–132. See also the retrospective in Sohi GS (ed) (1998) 25 years of the international symposia on computer architecture ACM, pp 2–4
 10. NEC Electronics, Inc. (December 1985) μ PD7281: image pipelined processor. Preliminary data sheet. NEC Electronics, Inc, Mountain View, CA
 11. Gaudiot J-L, Bic L (1991) Advanced topics in data-flow computing. Prentice Hall, New York
 12. Gostelow KP, Arvind (1977) A computer capable of exchanging processors for time. In: Information processing '77. North Holland, New York
 13. Gurd JR, Kirkham CC, Watson I, January (1985) The Manchester dataflow prototype computer. Commun ACM 28(1):34–52
 14. Hiraki K, Shimada T, Nishida K (1984) Hardware design of the SIGMA-1: a data-flow computer for scientific applications. In: International conference on parallel processing, Bellaire, MI. IEEE, pp 851–855
 15. Hiraki K, Sekiguchi S, Shimada T (1991) Status Report of SIGMA-1: a data-flow supercomputer. In: Advanced topics in data-flow computing. Prentice Hall, New York, pp 207–223
 16. Hoganauer EB, Newbold RF, Inn YJ (1982) DSSP: a data flow computer for signal processing. In: International conference on parallel processing, Bellaire, MI. IEEE, pp 126–133
 17. Iannucci RA (ed), Gao GR, Halstead RH, Smith B (1994) Multithreaded computer architecture: a summary of the state of the art. International series in engineering and computer science, Springer
 18. Kurokawa H, Matsumoto K, Temma T, Iwashita M, Nukiyama T (1983) The architecture and performance of image pipeline processor. In: VLSI '83: VLSI design of digital systems: proceedings of the IFIP TC 10/WG 10.5 international conference on very large scale integration, Trondheim, Norway. IFFIP/Elsevier, pp 275–284
 19. Mercaldi M, Swanson S, Petersen A, Putnam A, Schwerin A, Oskin M, Eggars S (2006) Instruction scheduling for a tiled dataflow architecture. In: Architectural support for programming languages and operating systems. ACM, New York, pp 141–150
 20. Papadopoulos GM, Culler DE (1990) Monsoon: an explicit token-store architecture. Seventeenth international symposium on computer architecture, IEEE Computer Society/ACM, New York, pp 82–91. See also the retrospective in Sohi GS (ed) (1998) 25 years of the international symposia on computer architecture. ACM, pp 74–76
 21. Papadopoulos GM, Traub KR (1991) Multithreading: a revisionist view of dataflow architectures. Proceedings of the 18th international symposium on computer architecture, ACM, New York, pp 342–351
 22. Sankaralingam K, Nagarajan R, Gratz P, Desikan R, Gulati D, Hanson H, Kim C, Liu H, Ranganathan N, Sethumadhavan S, Sharif S, Shivakumar P, Yoder W, McDonald R, Keckler SW, Burger DC (2006) Distributed microarchitectural protocols in the TRIPS prototype processor. In: Thirty-ninth international symposium on microarchitecture, IEEE, Washington, DC, pp 480–491
 23. Sharp JA (1985) Data flow computing. Wiley, New York
 24. Temma T, Mizoguchi M, Hanaki S (1983) Template-controlled image processor TIP-1 performance evaluation. In: Proceedings of the IEEE, CPVR - Computer Vision and Pattern Recognition. IEEE
 25. Tou J, Wegner P (eds) (February 1971) Data structures in programming languages. ACM SIGPLAN Notices 6:171–190. See especially the papers by Wegner, Johnston, Berry and Organick
 26. Vedder R, Campbell M, Tucker G (1985) The Hughes data flow multiprocessor. In: Proceedings of the fifth international conference on distributed computing, Denver. IEEE, pp 2–9
 27. Veen AH (1986) Dataflow machine architecture. ACM Comput Surv 18(4):365–396

Data Flow Graphs

JACK B. DENNIS

Massachusetts Institute of Technology, Cambridge, MA, USA

Synonyms

Program graphs

Definition

A *data flow graph* is a graph model for computer programs that expresses possibilities for concurrent execution of program parts. In a data flow graph, nodes, called *actors*, represent operations (functions) and predicates to be applied to data objects, and arcs represent channels for data objects to move from a producing actor to a consuming actor. In this way, control and data aspects of a program are represented in one integrated model. When data objects are available at input ports of an actor and certain conditions are satisfied, the actor is said to be *enabled*. Because the set of enabled actors may be chosen to fire simultaneously, or sequentially in any order, data flow models expose much of the

parallelism available in the computation represented, and this parallelism may be exploited in an implementation of the model.

Discussion

Introduction

Although many versions of data flow graphs (DFGs) have been studied in the literature, they share some significant common features.

1. A DFG is a directed graph representation in which an arc is a path over which data are passed from a producing node to a consuming node.
2. Dynamically, a node of a DFG acts (by *firing*, as in a Petri net) by accepting one or more data items from its inputs, performing some computation, and delivering resulting data items to its outputs.
3. Action by a node is triggered by the presence of input data.

The study of data flow graphs has focused mainly on three well-defined formal models: the static data flow model, dynamic data flow, and synchronous data flow. The various models differ in such aspects as how many data items are permitted to occupy an arc, whether actors are permitted to have internal state, and other details.

Basic forms of static DFGs and their properties are described below, including determinacy; this is followed by a discussion of dynamic data flow and the synchronous model. Uses of DFGs are summarized.

Basic Simple DFGs

Figure 1 shows a basic simple static data flow graph, which is an acyclic directed graph (DAG) in which the nodes are drawn as circles and represent *actors* that perform functional operations on a set of input values (possibly empty) to produce a set of one or more output values. The arcs of the graph are called *links*, and convey values from an output port of one actor to an input port of another, or from an input port of the DFG to an actor input, or from an actor output to an output port of the DFG.

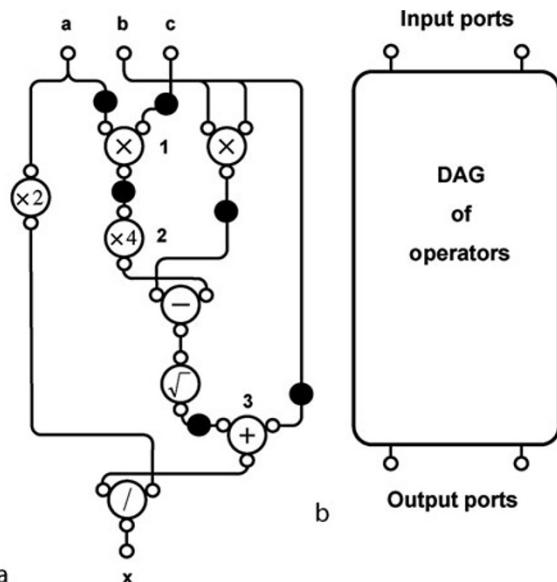
To discuss the semantics of a DFG, it is useful to associate tokens with arcs of the graph. Each token carries a value, which may be a scalar value such as an integer, or, in some studies, a data structure. For most

work with static data flow graphs, links are permitted to hold just one token. For simple static DFGs, each arc holds either a single token or is empty. The firing rule for advancing the state of a DFG is:

Firing rule: An actor of a DFG is *enabled* if a token is present on each input link of the actor, and no token is present on any output link. An enabled actor may be *fired* by removing one token from each input link and placing a token on each output link. The values held by the output tokens are the result of applying the operator of the actor to the set of values held by the input tokens.

Asynchronous firing of actors is permitted, and the time taken for any actor to fire may be arbitrary, so several actors in a DFG may be firing simultaneously. In **Fig. 1**, a state of execution is shown wherein actors 2 and 3 are both enabled and may be chosen to fire in either order, or to fire simultaneously. After actor 2 has fired, actor 1 will be enabled because its output link will then be empty.

The input ports of a DFG may be regarded as special actors that place tokens on the input links of the graph when values are supplied to the ports from the external environment of the DFG. Similarly, output ports make the values held by tokens on output links available to the environment and remove the tokens when the values are taken.



Data Flow Graphs. Fig. 1 Simple static data flow graphs

A DFG is activated by supplying data values at each input port, and terminates when values are available at each output port. A DFG is said to be *well-behaved* if its activity terminates following each presentation of values.

The DFG in Fig. 1a represents the standard formula for computing one of the roots of the second-degree polynomial, $a + bx + cx^2$:

$$x = \frac{b + \sqrt{b^2 - 4ac}}{2a}$$

This DFG is well-behaved, as are all simple static DFGs if their actors have operators that are total functions. Also, simple DFGs can represent pipelined computation, that is, more than one set of input values may be accepted by the DFG before its activity has terminated for the earlier set of input values. The pipelined operation of DGFs is closely related to use of coroutines as a program structure for parallel computing. Some workers with DFGs view each arc as a FIFO queue that can hold either a bounded or unbounded number of value-carrying tokens. This may be shown on the graph by permitting multiple tokens on any arc, with the stipulation that order is preserved: tokens are removed from the arc by a consuming actor in the same order as they were placed by the producing actor.

A simple data flow graph is *determinate*. Determinacy is an important concept in parallel computing. It captures the idea that a system can have repeatable behavior while at the same time being nondeterministic in its operation. The property of being well-behaved guarantees that the value produced at each output port is a fixed composition of the functions defined by the actors of the DFG. Basic simple DFGs are determinate whether or not they are well-behaved.

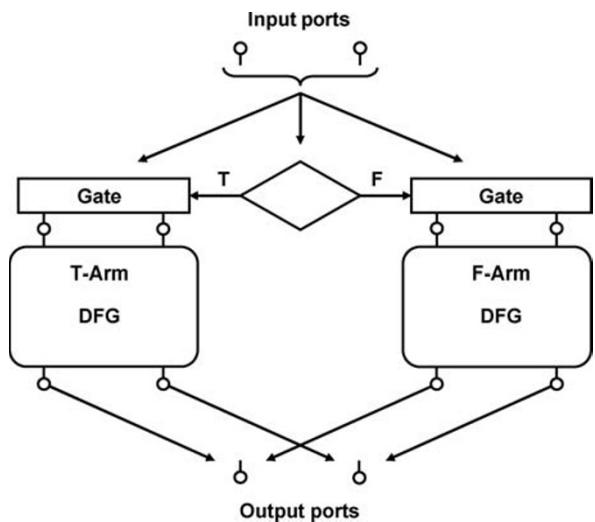
Compound Simple DFGs

A complete system for describing computations requires means for describing conditionals and iteration. In simple DFGs, these features are provided using DFG structures motivated by the principles of structured programming advocated by Dijkstra and Hoare that have had a strong influence on modern programming language design.

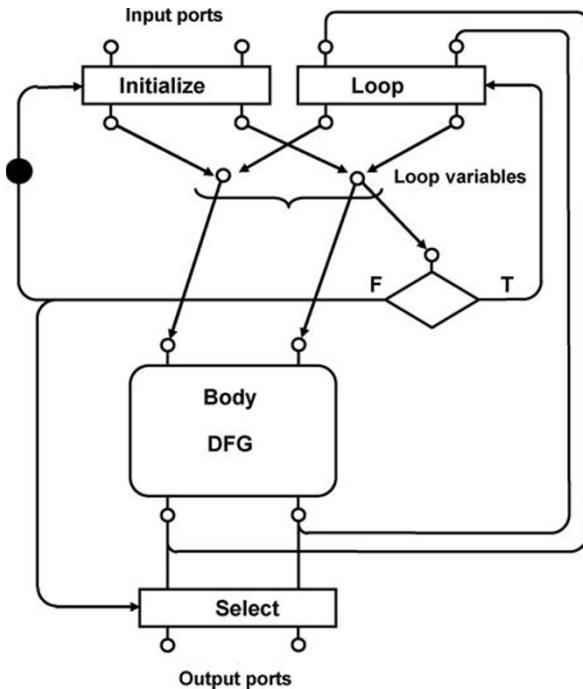
For ease of exposition, the manner of expressing conditional and iteration expressions as DFGs used here

is adapted from their use in several compilers for functional programming languages. Figure 2 illustrates a typical way of representing an *if...then...else* construction as a data flow graph. The diamond-shaped *decider* actor accepts a token carrying a boolean value from one of the input ports and generates *event signals* (tokens) that activate gating structures, shown as rectangular boxes, that select input values for passing to the two component DFGs: the T-Arm activated for a true decision and the F-Arm for false. Any combination of input ports may be selected as inputs to the arm graphs. However, the output ports of the arm graphs must be in one-to-one correspondence with output ports of the conditional graph so that output values are defined for every possible execution. A conditional DFG is determinate, and will be well-behaved if the two arm graphs are well-behaved.

Figure 3 illustrates one way of representing an iterative loop as a data flow graph, in this case a computation of the form **while...do...**. The body of the iteration is represented by a DFG that takes input values from a set of loop variables and produces output values that provide updated definitions for the loop variables. Here, the gating structures serve to provide initial values for the loop variables and then to redefine them from the outputs of the loop body for each cycle. The decider uses the value of a boolean loop variable to control the gating.



Data Flow Graphs. Fig. 2 A conditional DFG



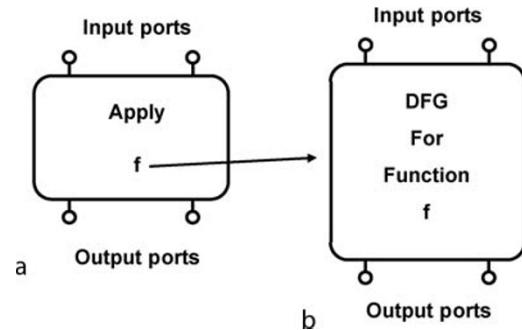
Data Flow Graphs. Fig. 3 Iteration as a data flow graph

Note that the initial state of this graph must have a token carrying an event signal for gating initial values to the loop variables. If the body DFG is well-behaved, then so will the compound DFG, provided the iteration terminates. Note that the compound DFG is determinate even if it might not terminate for some inputs.

The Apply Actor

In comparison with programming languages, the DFG constructions introduced so far are limited. There is no means for expressing function invocation, and no means for building and analyzing data structures. For function invocation, DFGs use the **apply** actor which has the semantics shown in Fig. 4. The Apply actor identifies a separate DFG that defines a computation to be performed whenever the Apply actor is executed. It is executed by copying values from the input ports of the apply actor to the input ports of the specified DFG. When the function body terminates, the values at its output ports are transferred to the output links of the Apply actor.

There are two ways of elaborating the semantics of the Apply actor: by substitution and by using *tagged*



Data Flow Graphs. Fig. 4 The apply actor

(colored) tokens. For substitution, the meaning of a DFG containing apply actors is the simple DFG obtained by copying the specified DFG in place of each apply actor. Of course, this reaches limitations for recursive functions where a potentially unbounded cascade of copying is necessary.

The second approach is to avoid the copying of graphs by permitting multiple tokens with distinct tags to be present on each DFG link. The tags are chosen so that all tokens that carry values associated with a specific activation of a function graph have identical tags. The use of tagged tokens also permits iteration to be represented in a way that permits actors from more than one cycle of an iteration to be active concurrently. The *tagged-token model* of DFG semantics has been the inspiration for at least two projects for building computers using data flow principles, and constitutes the essence of *dynamic data flow*.

An important and useful property of data flow graphs is that they not only express the dependence relations among operations of a computation, but also provide a recipe for execution – how to perform the computation – specifically the firing rule given above. The firing rule specifies a *data-driven* order of performance, that is, each operator acts just when all data it needs have arrived. In this scheme, the firing of an Apply actor, causing execution of the body of the applied function, is delayed until all inputs (function arguments) are present (evaluated). A consequence is that if the applied function does not make use of all of its arguments, unnecessary computation will be performed, and execution of such a program might not terminate if evaluation of an unneeded argument executes an endless loop.

This issue can be avoided by using *demand-driven* evaluation wherein a data flow actor is executed only when its result is needed. It is known how to convert a data flow graph into a new graph for which data-driven execution performs the actors of the original graph exactly as for demand-driven execution.

Data Structures

The simplest way to include data structures in a data flow model is to use **read** and **write** actors that access a memory that is not part of the DFG. The read operation accesses a specified location in the memory, and the write operation updates a memory location with a new value. The drawback of this approach is that the DFG is no longer necessarily determinate because read and write operations for the same memory location may be concurrent in the DFG.

A better way to introduce data structures is by permitting data objects, representing array or record values, to be carried by tokens in a DFG. Three new actor types are added: **create**, **append**, and **select**. The create actor, when fired, produces a null data structure having no components – representing an empty record or an array with no defined elements. The append actor performs the operation

$$\text{object} \leftarrow \text{append } (\text{object}, \text{index}, \text{element})$$

where the object produced has a component named *index* with *element* as its value. (The element could be a structured value such as a row of a matrix or a scalar value.) The object operated on by an append actor may be a null object from a create actor, or a structure from another append actor with one or more defined components. The select actor

$$\text{element} \leftarrow \text{select } (\text{object}, \text{index})$$

yields the component identified by *index* in the data structure represented by *object*. DFGs constructed using the select and append actors have functional (determinate) semantics as in pure Lisp. Other work on data flow models has treated arrays of values as sequences of tokens passed between actors.

Control Flow and Data Flow

A subject of importance for compiler design is the choice of an intermediate representation (IR) on which program transformations may be performed. Data flow

graphs are an attractive IR because they both represent dependence information and provide a complete executable semantics of the computation. Therefore, being able to use this form for programs defined by traditional imperative languages would be highly beneficial. It is straightforward to represent imperative programs by control flow graphs and means for converting control flow representations to data flow graphs are known. Hence, it is feasible to employ data flow graphs as an IR for optimizing transformations in compilers for imperative languages.

Synchronous Data Flow

In the synchronous data flow (SDF) model, arcs of a DFG are permitted to hold multiple tokens, and a generalized form of firing rule is used. Moreover, the firing of actors happens at definite points in time, as if at periodic cycles defined by a system clock. For each actor, the designer of the model may specify a number of tokens required to be present at each input port, how many are removed when the actor fires, and how many tokens are placed on output arcs. These numbers do not change during operation of the model. The links in an SDF model are treated as FIFO buffers, and their sizes, along with the numbers of tokens placed or removed by the firing of nodes must be chosen carefully to avoid deadlock. Conditional constructions and while loops are not permitted in SDFs as these would negate the property that a SDF graph describes a system with a constant rate of operation. The effect of a conditional may be achieved by incorporating the conditional within a node, such that the duration of its firing is invariant. Feedback paths may be present in a SDF graph and are important to represent continuous processes that have internal state using only state-free (functional) operators.

The synchronous data flow (SDF) model is intended as a representation of digital signal processing (DSP) computations that continue in execution for an unbounded time interval, consuming a stream of input data items and producing an output stream. A feature of the SDF model is that DSP systems with components operating at different clock rates can be accurately modeled.

Applications

The early work on data flow models inspired several attempts in universities and industry to build practical

computers embodying data flow concepts. See the article on ►Data Flow Computer Architecture in this encyclopedia. The programming languages Id, Val, Sisal, and Ph are functional programming languages specifically designed to permit natural expression of the parallelism exposed in data flow models. Applications in signal processing include the system modeling tool LabVIEW and work on synthesizing digital signal processing systems, as in the Ptolemy project at UC Berkeley. An important use of data flow graphs is as an intermediate form for programs processed by a compiler written in either a functional or an imperative programming language.

Related Entries

- Actors
- CSP (Communicating Sequential Processes)
- Data Flow Computer Architecture
- Dependences
- Functional Languages
- Message Passing Interface (MPI)
- Petri Nets

Bibliographic Notes and Further Reading

Precursors to data flow models may be found in simulation models for industrial systems, engineering descriptions of the processing of information packets, especially for communications systems, as reviewed in the survey by Davis and Keller [6]. However, the first works using the data-driven concept as a precise representation for computer programs and systems were published in the 1960s, independently by Karp and Miller at the IBM T. J. Watson Research Center [10], by Rodriguez at MIT [16] and by Duane Adams at Stanford University [1]. These early models were “static” in that they did not embody a natural way of describing arbitrary nests of possibly recursive function definitions. The 1980 paper of Dennis [8] includes an introduction to static DFGs.

The Graph/Heap model for dynamic data flow using colored tokens was described by Dennis in 1974 [7]. The work of Arvind, Gostelow and Plouffe in 1978 [3] formulated the dynamic data flow model in the form of the “unraveling interpreter” using tagged tokens. The semantics of dynamic data flow computations by graph copying was described by Dennis in 1974. The

synchronous data flow model was introduced and its properties studied in the 1987 paper of Edward Lee [11].

In 1982 the IEEE journal *Computer* published a special issue devoted to concepts and research on data flow models [2]. It includes an introduction by editors Tilak Agerwala and Arvind, a paper on data flow languages by William Ackerman, the survey of the data flow program graphs by Alan Davis and Robert Keller, a description of the unraveling interpreter by Arvind and Kim P. Gostelow, and a description of the Manchester tagged token data flow computer by Ian Watson and John Gurd. A “Second Opinion on Data Flow” by Gajski, Padua, Kuck, and Kuhn completes the issue. The book by Sharp [17] provides a later review of the major forms of data flow program models and the types of computer program execution models inspired by them.

Several programming languages have been developed or inspired by data flow concepts, including Val [12], Sisal [13], Lucid [4] and pH [14]. The use of data flow graphs as a program representation in Sisal compilers has been described by Dennis [9] and by Skedzielewski and Glauert [18], although there have been other unreported uses. The expression of demand-driven computations as demand-driven DFGs was described by Pingali and Arvind [15], and the use of DFGs in representing computations written in imperative languages is discussed by Beck, Johnson and Pingali [5].

For a discussion of the influence of data flow concepts on computer architecture and programming languages, see the encyclopedia entries on Data Flow Architecture and Parallel Functional Languages.

Bibliography

1. Adams DA (1970) A model for parallel computations. In: Hobbs LC (ed) Parallel processor systems, technologies and applications, Spartan Books, New York, pp 311–333
2. Agerwala T, Arvind A (eds) (1982) IEEE Computer 15, 2. Special issue devoted to data flow
3. Arvind A, Gostelow KP, Plouffe W (1978) An asynchronous programming language and computing machine. Technical report 114a, Department of Information and Computer Science, University of California, Irvine
4. Ashcroft EA, Wade WW (1977) Lucid, a nonprocedural language with iteration. Commun ACM 20(7):519–526
5. Beck M, Johnson R, Pingali K (1981) From control flow to dataflow. ACM Trans Progr Lang Syst 12:118–129
6. Davis AL, Keller RM (1982) Dataflow program graphs. IEEE Comput 15(2):42–49

7. Dennis JB (1974) First version of a dataflow procedure language. In: Lecture notes in computer science LCNS 19: programming symposium, Springer, Berlin, pp 362–376
8. Dennis JB (1980) Data flow supercomputers. IEEE Comput 13(11):48–56
9. Dennis JB (1989) The paradigm compiler: mapping a functional language for the Connection Machine. In: Simon H (ed) Scientific applications of the connection machine, World Scientific, Singapore, pp 301–315
10. Karp RM, Miller RE (1966) Properties of a model for parallel computations: determinacy, termination, queuing. SIAM J Appl Math 14(6):1390–1411
11. Lee EA, Messerschmitt DG (1987) Synchronous data flow. Proc IEEE 75(9):1235–1245
12. McGraw JR (1982) The VAL language: description and analysis. ACM T Progr Lang Syst 4(1):44–82
13. McGraw JR, Skedzielewski SS, Allan R, Oldehoeft R, Glauert J, Kirkham C, Noyce B, Thomas R (1985) SISAL: streams and iteration in a single assignment language: reference manual version 1.2. Technical report M-146, Rev. 1. Lawrence Livermore National Laboratory, Livermore, CA
14. Nikhil R, Arvind A (2001) Implicit parallel programming in Ph. Morgan Kaufman, San Francisco, CA
15. Pingali K, Arvind A (1986, 1987) Efficient demand-driven evaluation. ACM T Progr Lang Syst 7(2):311–333 (Part 1), and 8(1): 109–139 (Part 2)
16. Rodriguez JE (1969) A graph model for parallel computations. MIT technical report ESL-R-398, and MAC-TR-64, Cambridge, MA
17. Sharp JA (1985) Data flow computing. Ellis Horwood, Chichester
18. Skedzielewski S, Glauert J (1985) IF1 – an intermediate form for applicative languages. Technical report M-170, Lawrence Livermore National Laboratory, Livermore, CA

Data Mining

AMOL GHOTING

IBM Thomas. J. Watson Research Center, Yorktown Heights, NY, USA

Synonyms

Data analytics; Knowledge discovery

Definition

Data mining, also popularly referred to as knowledge discovery from data, is the automated or convenient extraction of patterns representing knowledge implicitly stored or catchable in large data sets, data warehouses, the Web, and other massive information

repositories or data streams. Parallel data mining refers to the use of parallel computing to reduce time to execute a data-mining algorithm.

Discussion

Introduction

Advances in data collection and storage technologies have allowed organizations to gather, collect, and distribute increasing amounts of data. Spurred by these advances, the field of data mining has emerged, merging ideas from statistics, machine learning, databases, and high performance computing. The main challenge in data mining is to extract knowledge and insight from massive data sets in a fast and efficient manner. This process is iterative in nature and involves a human in the loop. Therefore, to facilitate effective data understanding and knowledge discovery, it is imperative to minimize execution time of a data-mining query. Parallel data mining refers to the use of parallel computing to reduce the time required to execute a data-mining query.

Data mining commonly involves four classes of tasks: association rule mining, classification, clustering, and regression. These classes will be described next together with a brief summary of parallel algorithms for each task.

Association Rule Mining

Association rule mining is a popular method for discovering interesting relations between variables in large databases. Agrawal et al. introduced association rules [4] for discovering patterns involving products in transactional data. An example of transactional data is point-of-sale (POS) data that is produced in supermarkets, which lists the items purchased by each customer during a visit. An association rule found in such data could indicate that if a customer purchased items A and B together, the customer is likely to also purchase item C. An example of an association rule is $(\text{milk}, \text{sugar}) \rightarrow \text{cereal} (90\%)$, which means that if a customer buys milk and sugar together, he or she is likely to also buy cereal 90% of the time. Note that the antecedent of the rule can contain more than two items. Such rules are often used to enable various marketing activities such as promotional pricing and product placements. In addition to the above example from market basket

analysis, association rules are also employed in many other application areas, including web usage mining, intrusion detection, and bioinformatics.

Finding association rules requires determining all frequent patterns in a transactional data set. A post-processing step can then find all association rules using the set of frequent patterns. The frequent pattern mining problem was first formulated by Agrawal et al. [4]. Briefly, the problem description is as follows: Let $I = \{i_1, i_2, \dots, i_n\}$ be a set of n items, and let $D = \{T_1, T_2, \dots, T_m\}$ be a set of m transactions, where each transaction T_i is a subset of I . An itemset $i \subseteq I$ of size k is known as a k -itemset. The *support* of i is $\sum_{j=1}^m (1 : i \subseteq T_j)$, or informally speaking, the number of transactions in D that have i as a subset. The frequent pattern mining problem is to find all $i \in D$ that have *support* greater than a minimum support value, *minsupp*. The frequent pattern mining step is the most time-consuming step in association rule mining, and much work has been done to make it efficient.

Agrawal et al. [2] presented *Apriori*, the first efficient algorithm to solve the frequent pattern mining problem. The *Apriori* algorithm traverses the itemset search space in a breadth-first manner. At level k , candidate itemsets of size k are generated by using frequent itemsets of size $k - 1$. These candidates are then validated against the database (usually by a full scan) to obtain frequent itemsets of size k . *Apriori*-based algorithms speed up the computation by using the anti-monotone property and by keeping some auxiliary state information. The anti-monotone property states that if a size k -itemset is not frequent, then any size $(k + 1)$ -itemset containing it cannot be frequent.

Agrawal and Shafer [3] were the first to address the challenge of frequent pattern mining on shared-nothing architectures, proposing Count Distribution (CD) and Data Distribution (DD). Both CD and DD are parallelizations of the *Apriori* algorithm.

CD reduces execution time by parallelizing the data set scans. The data set is partitioned across all nodes, and each node scans its local data set to obtain frequencies of candidates. The frequency information is then accumulated to determine the global frequencies of the candidate itemsets. Once the global frequencies are obtained, infrequent itemsets are pruned, and each node generates candidates for the next level. Candidate generation is sequential, and not parallel, as each node

generates the same set of candidate itemsets. This process repeats until a level is reached where there are no more candidate itemsets. CD thus reduces the communication between processors at the cost of performing redundant computations (in the form of candidate generation) in parallel. However, it scales poorly as the number of candidates increases.

DD distributes the data set to each node. At the start of each level, each node takes turn broadcasting its local data set to every other node, and each node then generates a mutually disjoint partition of candidates. Furthermore, frequent itemsets are communicated so that each processor can asynchronously generate its share of candidates for the next level. DD overcomes the bottleneck of serial candidate generation by partitioning generation amongst the processors. However, it incurs very high communication overhead when broadcasting the entire data set, which is especially costly in case of large data sets. In their study, the authors find the costs of communication outweigh the improvements to parallel candidate generation. In both CD and DD, processors need to be synchronized at the end of each step, which can potentially cause processors to idle if the data is skewed.

Han et al. [14] proposed Intelligent Data Distribution (IDD) and Hybrid Distribution (HD), which build upon CD and DD. IDD reduces the communication overhead by distributing the data using a ring all-to-all broadcast algorithm. In the original DD algorithm, each node broadcasts its data set, which results in unnecessary communication. Although an improvement, IDD still suffers from the high overhead of transferring the entire data set at each level. The authors noted that a weakness in CD is that the global candidate list may exceed the size of available main memory for a single node. To address this problem while minimizing communication and computation overheads, HD combines the advantages of both the CD and IDD algorithms by dynamically grouping processors and partitioning the candidate set accordingly to maintain good load balance. Each group is large enough to hold the candidate list in main memory and executes the IDD algorithm. Communication across groups is performed as in CD.

Parthasarathy et al. considered the parallelization of the *Apriori* algorithm on shared memory systems [23]. They showed that the hash-based data structures used by *Apriori* during the candidate generation and

counting phases result in poor data locality and false sharing on shared-memory systems. To address these issues, the authors proposed several memory placement techniques to improve performance.

Unlike the above approaches that traverse the search space in a breadth first manner, *Eclat* [28] is an algorithm that traverses the search space in a depth first manner. The algorithm uses itemset clustering techniques to approximate the set of potentially maximal frequent itemsets. A maximal frequent itemset is a frequent itemset that cannot have a frequent superset itemset. Once these sets have been identified, the algorithm makes use of efficient depth-first traversal techniques to generate frequent itemsets contained in each cluster. The advantages of this approach compared with the Apriori-style approaches are improved locality and a natural structure for parallelization – each cluster can be processed independently. The structure of this algorithm naturally lends itself to both shared-memory and distributed-memory parallelizations.

Han et al. proposed the *FPGrowth* [15] algorithm that improves performance through the use of a smart data structure known as the *FP-tree* and by employing a projected data set during the depth-first traversal of the search space. However, while delivering excellent sequential performance, a limitation to parallelizing this algorithm is its reliance on a dynamic, pointer-based data structure and the fact that the data structure can potentially be out-of-core for very large data sets.

Cong et al. [10] proposed a sampling-based framework for parallel itemset mining using FPGrowth on shared-nothing systems. They first selectively sample the transactions by discarding a fraction of the most frequent items from each transaction. Based on the mining time on the reduced data set, the computation is divided into tasks needing roughly the same amount of execution time. In practice, they found that sample mining times were quite representative of actual mining times. Using these timing results, tasks could be assigned to machines statically, while affording excellent load balancing. A drawback of their approach, particularly when mining very large dense data sets is that it assumes that the FP-tree fits in core, and that the data is present on a single central node, which might not be true in many real world scenarios. To address the limitation of the approach by Cong et al., Buehrer et al. presented a parallel algorithm [8] based on FPGrowth –

the salient features of the algorithm include a serialization and merging strategy for computing global trees from local trees, efficient tree pruning for better use of available memory space, and mechanisms to efficiently handle out-of-core data sets and data structures.

Ghoting et al. [12, 13] presented a cache-conscious approach for the effective parallelization of FPGrowth on shared-memory systems. They showed that poor cache performance limits scalability on shared-memory systems and demonstrated that through a careful architecture conscious redesign these bottlenecks can be removed, affording significant improvements. The authors developed smart data placement and computation restructuring techniques to improve both spatial as well as temporal locality during execution.

Classification

Data classification is a supervised learning procedure that given a training data set consisting of data points and their corresponding labels, learns an unknown function that is capable of mapping the input data elements to their corresponding labels based on some characteristic inherent in the input data set. For instance, consider the problem of filtering email spam. Here, one has some representation of an email and a label indicating whether the email is “Spam” or “Non-Spam.” The goal of data classification would be to learn a function that can label an email as “Spam” or “Non-Spam.” This function can then be applied to new data items, in this case, new email messages, to classify them as “Spam” or “Non-Spam.” A variety of data classification algorithms are used in data mining. Examples of classifiers are neural networks (multilayer perceptrons), support vector machines, k-nearest neighbor classifier, naive Bayes, and decision trees. These algorithms differ in the nature of functions they use to approximate the target function. Describing all these algorithms and their parallelizations is beyond the scope of this entry. For many of these algorithms, computation is structured and often specified using linear algebra, allowing for a parallel implementation using popular libraries such as LAPACK [20]. Due to the unstructured nature of its computations, decision trees are described – a family of extremely popular and scalable classifications algorithms.

Decision trees approximate the target function as a tree. Each interior node in this tree corresponds to one

of the input variables; there are edges to children for each of the possible values of that input variable. Each leaf represents a value of the target variable given the values of the input variables represented by the path from the root to that leaf. Decision trees are popular as they can be used to generate a set of classification rules (each path from the root to the leaf represents a classification rule) that are easy to interpret. An example of such a classification rule is : if temperature >90, humidity <70, and outlook = sunshine, then weather is good for playing tennis. A decision tree is typically learnt using a recursive procedure. During each step in this recursion, one determines an (attribute,value) pair that can be best used to split a data set. Here, “best” is defined by how well the (attribute,value) splits the data set into subsets that have similar values on the target label. Finding the best split involves sorting each attribute on its value so as to estimate the distribution of the target label for the potential splits. Starting with the input data set, the process is repeated on each derived subset until it is determined that splitting no longer provides any improvement to the predictions.

SLIQ [21] was one of the first scalable algorithms for decision tree induction. The approach first sorts each attribute in the data set separately. Each attribute is then maintained as a list of sorted values together with the corresponding record identifier. Presorting means that attributes do not need to be sorted through each step in the recursion. The tree is built level-by-level (or bread-first), each level requiring a scan of the input data set. Unfortunately, SLIQ uses a memory-resident data structure that stores the class labels of each record. This data structure limits the size of the data sets SLIQ can handle.

Shafer et al. [24] presented a more memory-efficient version of SLIQ called SPRINT and also gave parallel versions of SPRINT for shared-nothing systems. This algorithm maintains the class label along with the record identifier for each value in the attribute lists. The data set is first partitioned across all processors in a row-wise fashion. Then, a global sort operation is performed at the end of which each processor has an equal and contiguous portion of each attribute list. When determining split points, each processor determines the best split points for all the records that were assigned to it. This is followed by an all-to-all broadcast operation

to determine the best global split point. Once this decision has been made, the attribute lists of the splitting attribute can be partitioned easily. In order to split other lists, a hash table is formed to maintain a mapping of record identifiers to nodes (of the decision tree in the next level), which needs to be communicated to all the other processors from the processor that produced the winning split point. The other processors then use the hash table to split their own attribute lists. The size of this hash table is proportional to the number of records at the current node, which means that for the root of the decision tree, the size of the hash table is proportional to the total number of records in the training set. ScalParC [19] improves on SPRINT by using a distributed hash table that need not be locally constructed and thus eliminates the bottleneck that the hash table presented for SPRINT.

The above mentioned approaches build a decision tree level-by-level and thus exploit data parallelism. However, disjoint sub-trees of a decision tree being independent, an alternative strategy would be to employ task parallelism. It is recognized that data parallelism is more efficient when processing nodes near the root of the tree, while task parallelism is more efficient when processing nodes near the leaf of the tree. This is the basis of the pClouds approach [25] for decision tree construction on shared-nothing systems. Similar strategies have also been utilized for decision tree construction on shared-memory systems [29].

Clustering

Data clustering, or simply clustering, is the assignment of a set of observations into subsets (called clusters) so that observations in the same cluster are similar in some sense. It is a form of unsupervised learning. Data clustering is vastly popular and has several real-world applications. One example is the assignment of customers into segments or groups based on some customer traits (e.g., spending habits) to better understand groups as well as the relationships between groups. Another example is the grouping of web pages into genres to automatically categorize web pages. One of the most important steps in clustering is to define a distance measure between points to measure similarity. The nature of the distance measure dictates the shape of the discovered clusters as well as the quality of the clustering. Clustering algorithms can be roughly

put into three categories – partitional, hierarchical, and density-based algorithms.

Partitional algorithms typically determine all clusters at once. An example of a partitional clustering algorithm is the popular *kMeans* clustering algorithm [16]. Given a data set and the desired number of centers (k), the algorithm first generates k random cluster centers. Next, each point in the data set is assigned to its nearest cluster center, where “nearest” is defined with respect to one of the distance measures. Finally, the new cluster centers are recomputed as the means of the points that were previously assigned to the centers. These steps are repeated until some convergence criterion is met (usually that the cluster centers haven’t changed). The main advantage of the *kMeans* algorithm is that it is easy to implement and parallelize, and thus can be used to process large data sets. Its parallelizations on shared-memory and shared-nothing systems are very similar. The task of assigning data points to one of k centers is the most time-consuming step and is easy to parallelize. Each processor is responsible for assigning a portion of the data points to one of k centers. Each processor maintains arrays for the means of all points and number of points that were assigned to the centers. These arrays are then summed up and used to find new k centers.

Hierarchical algorithms find successive clusters using previously established clusters [17]. These algorithms usually are either agglomerative (“bottom-up”) or divisive (“top-down”). Agglomerative algorithms begin with each data point as a separate cluster and merge them into successively larger clusters. Divisive algorithms begin with the whole data set (as one cluster) and proceed to divide it into successively smaller clusters. Most hierarchical clustering algorithms employ a greedy approach to divide or merge clusters for efficiency reasons, i.e., during each iteration they either divide (“top-down”) or merge (“bottom-up”) clusters that are deemed to provide the best clustering. During each iteration, a distance measure between not just a pair of points but groups of points needs to be defined. Due to a wide range of such possible distance functions, a variety of hierarchical clustering variants exist today. All these algorithms present worse than linear scalability, making hierarchical clustering computationally demanding. The reader can look at the work by Olson [22] for an analysis of parallel hierarchical clustering on various parallel abstraction machines. Due to

the sequential nature of the computations, few practical parallel algorithms for hierarchical clustering exist today.

Density-based clustering algorithms attempt to cluster data points based on their local densities [11]. Such clustering algorithms are capable of discovering clusters with arbitrary shapes. Clusters are groups of points that have dense neighborhoods and are connected in some sense. The primary computation performed in density-based clustering is the nearest neighbor search, and such clustering algorithms have been parallelized on shared-nothing systems by using a distributed index structure [26].

Regression

Regression analysis is used to model the relationship between a target variable and a set of predictor variables. This relationship is expressed as a function that predicts the target variable using the predictor variables as inputs. Regression analysis is widely used for prediction and forecasting. Regression is similar to classification in that both approaches attempt to predict a target variable. Unlike classification, where the target variable is categorical, regression attempts to predict a continuous variable. A variety of regression algorithms are used in data mining. Techniques for classification such as neural networks (multilayer perceptrons) support vector machines, k -nearest neighbors, and decision trees have been adapted to the problem of regression, and their parallelizations are structurally similar. Parametric approaches to regression such as linear regression and logistic regression are often specified using matrix operations, and they can be implemented in parallel using popular libraries such as BLAS [5] and LINPACK [20].

Bibliographic Notes and Further Reading

The earlier portion of this entry concentrated on association rule mining. Over the past decade, a variety of extensions to association rules such as sequential rules and graph patterns have emerged. The reader can look at the book by Zaki and Ho [27] and papers by Buehrer et al. [6] for their parallelizations.

Until this last decade, most data sets that were used in data mining could be processed on a single machine. With the advent of the world wide web and advances

in data collection technologies, institutions are increasingly seeing the need to implement parallel data-mining algorithms, and an emerging research area is that of building infrastructures that are specifically aimed at implementing parallel data mining algorithms. Technologies such as the IBM Parallel Machine Learning Toolbox [18] and the rapidly growing Hadoop Apache project [1] are indicators of this trend.

Another recent research direction in parallel data-mining is that of executing data-mining algorithms on emerging multicore architectures such as the commodity GPU [9] and the STI Cell processor [7]. Due to the nature of these architectures, researchers have targeted data mining algorithms that have structured computation and data access patterns.

Bibliography

1. Apache Hadoop. <http://apache.hadoop.org>
2. Agrawal R, Srikant R (1994) Fast algorithms for mining association rules in large databases. In: International conference on very large databases, Santiago, pp 487–499
3. Agrawal R, Shafer JC (1996) Parallel mining of association rules. *IEEE T Knowl Data Eng* 8(6):962–969
4. Agrawal R, Imielinski T, Swami AN (1993) Mining association rules between sets of items in large databases. In: ACM international conference on management of data, Washington, pp 207–216
5. BLAS. <http://www.netlib.org/blas>
6. Buehrer G, Parthasarathy S, Chen Y (2006) Adaptive parallel graph mining for CMP architectures. In: Sixth international conference on data mining ICDM'06, Hong Kong, pp 97–106
7. Buehrer G, Parthasarathy S, Ghoting A (2006) Out-of-core frequent pattern mining on a commodity PC. In: International conference on knowledge discovery and data mining, Philadelphia, pp 86–95
8. Buehrer G, Parthasarathy S, Goyder M (2008) Data mining on the cell broadband engine. In: Proceedings of the 22nd annual international conference on supercomputing, Island of Kos, pp 26–35. ACM, New York
9. Catanzaro B, Sundaram N, Keutzer K (2008) Fast support vector machine training and classification on graphics processors. In: Proceedings of the 25th international conference on machine learning, Helsinki, pp 104–111. ACM, New York
10. Cong S, Han J, Hoeftlinger J, Padua DA (2005) A sampling-based framework for parallel data mining. In: International conference on principles and practice of parallel programming, Chicago, pp 255–265
11. Ester M, Kriegel H, Sander J, Xu X (1996) A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proceedings of international conference on knowledge discovery and data mining, Portland, vol 96, pp 226–231
12. Ghoting A, Buehrer G, Parthasarathy S, Kim D, Nguyen A, Chen Y, Dubey P (2005) Cache-conscious frequent pattern mining on a modern processor. In: Proceedings of the 31st international conference on very large data bases, Trondheim, pp 588
13. Ghoting A, Buehrer G, Parthasarathy S, Kim D, Nguyen AD, Chen Y-K, Dubey P (2007) Cache-conscious frequent pattern mining on modern and emerging processors. *VLDB J* 16(1):77–96
14. Han E-H, Karypis G, Kumar V (2000) Scalable parallel data mining for association rules. *IEEE T Knowl Data Eng* 12(3):377–352
15. Han J, Pei J, Yin Y (2000) Mining frequent patterns without candidate generation. In: International conference on management of data, Dallas, pp 1–12
16. Hartigan J (1975) Clustering algorithms. Wiley, New York
17. Johnson S (1967) Hierarchical clustering schemes. *Psychometrika* 32(3):241–254
18. I. P. M. L. Toolbox. <http://www.alphaworks.ibm.com/tech/pml>
19. Joshi M, Karypis G, Kumar V (1998) ScalParC: a new scalable and efficient parallel classification algorithm for mining large datasets. In: IPPS '98: proceedings of the 12th international parallel processing symposium on international parallel processing symposium, Orlando. IEEE Computer Society, Washington, DC, pp 573
20. LAPACK. <http://www.netlib.org/lapack>
21. Mehta M, Agrawal R, Rissanen J (1996) SLIQ: a fast scalable classifier for data mining. In: Advances in database technology EDBT'96, Avignon, pp 18–32
22. Olson C (1994) Parallel hierarchical clustering. Technical report, University of California, Berkeley
23. Parthasarathy S, Zaki MJ, Li W (1998) Memory placement techniques for parallel association mining. In: International conference on knowledge discovery and data mining, New York, pp 304–308
24. Shafer JC, Agrawal R, Mehta M (1996) Sprint: a scalable parallel classifier for data mining. In: VLDB, Mumbai, pp 544–555
25. Sreenivas M, Alsabti K, Ranka S (1999) Parallel out-of-core divide-and-conquer techniques with application to classification trees. In: Proceedings of the 13th international symposium on parallel processing and the 10th symposium on parallel and distributed processing, San Juan, pp 555–562. IEEE Computer Society, Los Alamitos
26. Xu X, Jäger J, Kriegel H (2002) A fast parallel clustering algorithm for large spatial databases. In: Guo Y, Grossman R (eds) High performance data mining, Kluwer, New York, pp 263–290
27. Zaki M, Ho C (2000) Large-scale parallel data mining. Springer, New York
28. Zaki MJ, Parthasarathy S, Ogihsara M, Li W (1997) New algorithms for fast discovery of association rules. In: KDD, Newport Beach, pp 283–286
29. Zaki M, Ho C, Agrawal R (1999) Parallel classification for data mining on shared memory multiprocessors. In: ICDE '99: proceedings of the 15th international conference on data engineering, Sydney. IEEE Computer Society, Washington, DC

Data Race Detection

- ▶ [Intel Parallel Inspector](#)
- ▶ [Race Detection Techniques](#)

Data Starvation Crisis

- ▶ [Memory Wall](#)

Dataflow Supercomputer

- ▶ [SIGMA-1](#)

Data-Parallel Execution Extensions

- ▶ [Vector Extensions, Instruction-Set Architecture \(ISA\)](#)

Deadlock Detection

- ▶ [Deadlocks](#)
- ▶ [Intel Parallel Inspector](#)

Deadlocks

Roy H. CAMPBELL
 University of Illinois at Urbana-Champaign, Urbana,
 IL, USA

Synonyms

[Deadlock detection](#); [Gridlock](#); [Hang](#); [Impass](#); [Stalemate](#)

Definition

A deadlock is a condition that may happen in a system composed of multiple processes that can access shared resources. A deadlock is said to occur when two or more processes are waiting for each other to release a resource. None of the processes can make any progress.

Discussion

Deadlocks are a problem in parallel computing systems because of the use of software or hardware synchronization resources or locks to provide mutual exclusion for shared data and process coordination.

In general, a deadlock has four necessary conditions:

1. A resource cannot be used by more than one process at a time, commonly called *mutual exclusion*.
2. A process using a resource may request another resource, a *hold and wait* condition.
3. A *no preemption* condition applies to the resources held by a process, and they cannot be released without an action of the process.
4. Two or more processes are waiting for each other to release a resource in a *circular wait* or chain of dependency.

In parallel programming, resources can have many different forms. Some examples are:

1. Memory that must be updated as if by an atomic or indivisible operation
2. Devices like tape drives that can only be used by one process at a time
3. Synchronization primitives that coordinate access to the use of resources
4. Messages that must be sent and received

Example

Consider two sets of memories M1 and M2. If process P1 has exclusive access to the contents of M1 and needs exclusive access to M2 to complete its operations but process P2 has exclusive access to the contents of M2 and needs exclusive access to M1 to complete its own operations, then this is a deadlock. An I/O device like a tape drive may be considered as a substitute for the memory.

In shared memory parallel programming scenarios, semaphores can be used to coordinate access to a resource. Each memory location (or tape drive) would be associated with a semaphore. An operation P() would represent a request to have access to the resource and an operation V() would represent its release. So a program for process P1 might be written:

```
P1: S1.P(); /* get exclusive access to memory M1 */
          /* ... */
S2.P(); /* get exclusive access to memory M2 */
```

```
/* update M1 and M2 */
S2.V();/* release exclusive access to memory M2 */
S1.V();/* release exclusive access to memory M1 */
```

Semaphore S1 is associated with access to M1 and S2 to M2. The value of a semaphore represents whether its associated resource is available. In this case, each semaphore is given an initial value of 1 for its associated memory location. A semaphore value of 0 represents that a process has exclusive access to memory M1. The operation P() decreases the value of a semaphore, while the operation V() increases it. If the semaphore is 0, a P() operation makes the requesting process wait for the semaphore before continuing.

Process P2 is written in a similar manner:

```
P2: S2.P();/* get exclusive access to memory M2 */
S1.P();/* get exclusive access to memory M1 */
/* update M1 and M2 */
S1.V();/* release exclusive access to memory M1 */
S2.V();/* release exclusive access to memory M2. */
```

Then the two processes P1 and P2 deadlock if process P1 executes its first semaphore operation S1.P() and process P2 also executes S2.P().

Coordination of processes can create a deadlock as when process P1 waits for a message from process P2 before sending a message to P2 and process P2 either never sends a message to P1 or P2 waits for a message from P1 first and then sends a message to P1.

Each example satisfies the four deadlock conditions and may be extended to more processes and resources.

Deadlock Detection, Prevention, and Avoidance

The set of four deadlock conditions are necessary and sufficient to create a deadlock. Detecting, prohibiting, or avoiding one of the conditions is thus a basis for deadlock detection, prevention, or avoidance [10]. *Deadlock detection* requires analyzing whether the four conditions apply to a set of processes and resources. *Prevention* ensures that one or more of the conditions cannot hold. *Avoidance* analyzes whether granting a resource request could eventually lead to a deadlock and blocks the process from making that request if such a possibility exists. In turn:

1. Allowing a resource to be used by more than one process at a time removes *mutual exclusion*. Thus,

for example, making memory access read-only eliminates the need for mutual exclusion. Similarly, devices may be shared as in if a tape device is used in append only mode. Non-blocking synchronization primitives may be used that return an error code if a resource is in use. Messages may be sent in asynchronous fashion so that a message receive does not block a process or synchronization primitives.

Programming synchronization and coordination solutions without blocking primitives use “non-blocking synchronization algorithms.” However, such algorithms can suffer from livelock, a similar problem to deadlock in which processes make no progress but are never blocked.

2. The *hold and wait* condition can be eliminated by forcing a process either to acquire all the resources it needs in one operation or to release all the resources it holds when it acquires more resources. Job control in early IBM 360 systems imposed such conditions. In general, this is inefficient as it leads to a process holding more resources than it needs or to an undue number of acquire and release operations.
3. Allowing resources to be removed from a process eliminates the *no preemption* condition. For example, processes often require additional physical memory and virtual memory algorithms allow the physical memory of another process to be released in order to prevent a deadlock arising from memory allocation needs. Another approach often used to preempt a resource is to roll the process holding the resource back to a previous state prior to the time it acquired the resource. This leads to optimistic concurrency control, lock-free, wound-and-wait, and wait-free algorithms for resource allocation. In general, preemption of resources can lead to inefficiencies and thrashing.
4. Ordering the acquisition of resources eliminates the *circular wait* condition and can be implemented in a number of ways. For example, the address of a resource can be used to create an order in which resources are acquired or the resources can be organized into a hierarchy.

As an example above of ordering resources, if the address of M1 is less than M2, then this can be used to impose an ordering on the semaphores such that both

process 1 and process 2 always perform a P.S1 before P.S2. This requires rewriting process P2 as:

```
P2: S1.P();/* get exclusive access to memory M1
    S2.P();/* get exclusive access to memory M2
    /* update M1 and M2
    S2.V();/* release exclusive access to memory M2
    S1.V();/* release exclusive access to memory M1
```

Now, whichever process performs S1.P() first can perform the S2.P() and continue until it can release its access to M1 and M2 and perform S2.V() and S1.V().

Detection

Given an arbitrary parallel program, detecting whether that program will deadlock is undecidable except in very specific circumstances, in the same manner as the halting problem is undecidable. There is much literature concerning the undecidability of deadlocks. Brand and Zafiroplou [2] describe the computational complexity of deadlock detection in communicating state machines, and Gold [8] describes the computational complexity of deadlock avoidance algorithms. However, deadlocks are detectable at runtime. Since the processes involved in a deadlock are waiting, deadlock detection must be performed by a monitoring algorithm that tracks whether the four conditions have been met for the given set of processes and resources. Deadlock detection can be achieved by building a resource allocation graph [12] that represents the state of the system in which a deadlock is observed as a cycle in the graph. Detection algorithms have a runtime complexity of at least $O(M \times N)$ or $O(M^3)$, where M number of resources and N is the number of processes (Holt 1979). Kim and Koh found a complexity of $O(M \times N)$ using trees.

Avoidance

Deadlock detection works well for processes that are already deadlocked. It is possible, under certain restrictions, to determine whether allowing a process to acquire a resource might eventually lead to deadlock. If a deadlock is unavoidable, the process and its request can be suspended pending the release of other resources, leading to a variety of avoidance algorithms.

The banker's algorithm [9] is one such avoidance technique, and it works with a given number of processes and resources, each resource being of a particular type. It imposes the restriction that every process declares the maximum number of resources it might need of a particular type. When a resource is requested, the algorithm determines whether to allocate that resource or suspend the process based on if that allocation will keep the system in a *safe* state.

A safe state is one in which there is some sequence of allocations of resources to processes such that each process can eventually acquire its maximum number of resources needed so that it may complete and release those resources. This algorithm is conservative in that it avoids situations that are unsafe, but may not lead to deadlock. For example, a process may decide not to request its maximum number of resources but the banker's algorithm has no way of accommodating this information.

If it is inexpensive to kill and restart a process, then another avoidance approach is to allow processes to continue until a request is made that will cause a deadlock. Every process is given an age using a timestamp. In the *Wait/Die* algorithm, if the process requesting the resource is older than the process holding the resource, it waits. Otherwise, the process dies. In the *Wound/Wait* algorithm, if the process requesting the resource is younger than the process holding the resource, it waits. Otherwise, the process dies. In both algorithms, the aging scheme ensures that progress is made by preferentially killing the younger process.

Distributed Deadlock

The problems of deadlock extend to distributed systems. The solutions are similar, but require distributed implementations. Distributed deadlock detection solutions decompose the global resource allocation graph into local wait-for graphs that are then combined globally to detect cycles indicating circular wait using graph reduction [3]. An alternative approach to detecting the cycles is to trace the edge of the wait-for graphs from one node to another to determine if it forms a cycle.

Discussion

In practice, deadlocks have not proven to be as difficult an issue as at first thought. The cost of restarting a process or system when a deadlock is discovered is often not too great compared with the cost incurred from an implementation of deadlock prevention or avoidance.

Bibliography

1. Belik F (1990) An efficient deadlock avoidance technique. *IEEE Trans Computers* 39(7):882–888
2. Brand D, Zafiropulo P (1981) On communicating finite-state machines. *IBM Res Report RZ 1053 (#37725)*
3. Bracha G, Toeg S (1987) Distributed deadlock detection. *Distrib Comput* 2:127–138
4. Coffman E, Elphick M, Shoshani A (1971) System deadlocks. *Comput Surveys* 3:67–78
5. Dijkstra E (1965) Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, The Netherlands
6. Dijkstra E (1977) The mathematics behind the banker's algorithm. In: Dijkstra EW, published as pages 308–312 of Edsger W. Dijkstra, Selected writings on computing: a personal perspective. Springer, Berlin
7. Ezpeleta J, Tricas F, Garcia V, Colom J (2002) A banker's solution for deadlock avoidance in FMS with flexible routing and multiresource states. *IEEE Trans Robot Autom* 18(4):621–625
8. Gold E (1978) Deadlock prediction: easy and difficult cases. *SIAM J Comput* 7(3):320–336
9. Habermann A (1969) Prevention of system deadlocks. *Commun ACM* 12(7):373–377, 385
10. Havender J (1968) Avoiding deadlock in multitasking systems. *IBM Syst J* 7(2):74–84
11. Holt R (1971) Comments on prevention of system deadlocks. *Commun ACM* 14(1):36–38
12. Holt R (1972) Some deadlock properties of computer systems. *ACM Comput Surveys* 4:179–196
13. Kim J, Koh K (1991) An O(1) time deadlock detection scheme in single unit and single request multiprocess system. In: IEEE TENCON'91, New Delhi, India. pp 219–223
14. Lang S (1999) An extended banker's algorithm for deadlock avoidance. *IEEE Trans Softw Eng* 25(3):428–432
15. Lee J, Mooney V (2005) Hardware-software partitioning of operating systems: Focus on deadlock detection and avoidance. In: IEE proceedings on computers and digital techniques
16. Leibfried T (1989) A deadlock detection and recovery algorithm using the formalism of a directed graph matrix. *Operating Syst Rev* 45–55
17. Shoshani A, Coffman E (1970) Detection, prevention and recovery from deadlocks in multiprocess, multiple resource systems. In: 4th annual Princeton conference on information sciences and system, Princeton, NJ

Debugging

CHRISTOF KLAUSECKER, DIETER KRAZLMÜLLER
Ludwig-Maximilians-Universität München, Munich,
Germany

D

Definition

Debugging and testing are integral parts of the software development process, which provides a model for the development of software products from the initial planning to deployment and maintenance. However, debugging and testing are, although closely related, two completely different tasks which must be clearly distinguished. Software testing is used to verify the reliability and functionality of software products and individual components. This is accomplished by checking their adherence to specified requirements, thus allowing an assessment of the quality of software. If any discrepancies occur during the testing phase, debugging is used to track the reasons why the software fails and to correct the mistakes. Concurrency in applications adds additional error sources and effects, making debugging parallel programs a challenging task. As such, parallel program debugging has to deal with increased complexity, large amounts of data, and effects like race conditions and deadlocks.

Discussion

Overview

Debugging of parallel programs is a complex topic. Therefore, Section “Bugs” starts by clarifying the meaning and origin of the term “bug” itself. Section “The Debugging Process” describes a commonly used debugging approach and the difficulties posed by parallel applications. Section “Breakpointing in Parallel Programs” presents the purpose of breakpointing, one of the essential features provided by interactive debuggers, together with different breakpoint types for concurrent programs. Debuggers, that is, the tools commonly used to track down bugs, will be discussed and classified in Sect. “Parallel Debugging Tools”. Current issues and future directions will be presented in Sect. “Future Directions”.

Bugs

A “bug,” which in computing can occur in hardware or software, causes unexpected or unintended behavior that diverges from the product’s specification. A common belief is that the term originated from the finding of a real living bug. While the moth that was found in one of the relays of the Mark II computer [11] may have actually been the first real bug to be found in a computer, the term “bug” was already used at the end of the nineteenth century to describe technical glitches [5].

Definition 1 Bug

A bug is the commonly used term for an error with originally unknown location and reason [18].

According to Definition 1, the location of a bug is typically initially unknown. Its causes are small mistakes, design errors, or hard- and software failures. Their effects, however, can vary greatly in their extent. Bugs crashing a program immediately may seem to be the more severe ones, but they are also more likely to get noticed, and thus usually have a limited life span. The serious bugs are often those having initially only limited effects, or those occurring only sporadically, because they are located in seldom used functions or appear only under special circumstances – for instance only when running a parallel program with a higher number of processes or with specific input data. Such bugs remain undetected for a long time, often until an application is used in production. A product containing bugs is not only annoying for the users, but may also cause consequences such as financial loss and bad publicity for the developer.

While bugs can be avoided to a certain extent by carefully planning and designing using the established software development processes and by practicing good code style, the more complex a program gets, the more likely it is to contain bugs. It is practically impossible to write larger pieces of bug-free code. The reason for this is that software is written by humans, who are likely to make “dumb mistakes” [8]. Parallel programs are especially error-prone since they are often complex due to their concurrent nature and the resulting necessary communication and synchronization steps. As a consequence, software should be tested extensively for unintended behavior and upon detection the causes need to be found and the errors corrected.

The above given definition of a bug stresses the fact that its location is initially unknown, so in order to fix it, it needs to be tracked down first. The process of finding and fixing bugs is called debugging and is usually supported by corresponding tools.

The Debugging Process

As stated before, the best approach is to avoid producing bugs in the first place, which is unfortunately rarely possible. When a programmer encounters bugs, the goal of the debugging process (Definition 2) is to reduce the number of bugs, and consequently make an application execute and behave according to its specification.

Definition 2 Debugging

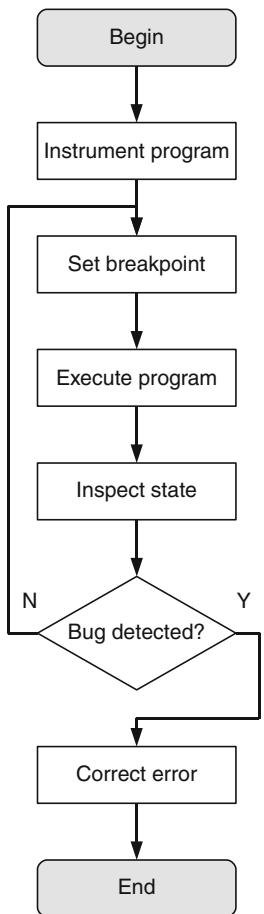
Debugging [22] is the process of locating, analyzing, and correcting suspected errors.

According to the definition of a bug (Definition 1), its location is initially unknown; therefore, to remove it, it needs to be located. In practice approximately 95% of debugging time is spent on locating the origin [25].

The typically applied debugging strategy (in sequential and in parallel case) is the cyclic debugging approach [22] as shown in Fig. 1. The underlying principle is as follows: After a bug has been detected, the developer reruns the program under control of a debugger to narrow down the location of the bug. The same input data is used to reproduce the same faulty behavior. The developer uses breakpoints to stop a program at execution states where the origin of errors is suspected or where additional knowledge about the program’s behavior can be obtained. If the information gathered during a cycle is not enough to isolate the bug, the procedure starts again by setting new breakpoints.

A different approach is reverse debugging, which based on recorded information, enables to execute a target program in reverse to a limited extent. This functionality has been recently introduced in some debuggers [6, 30].

The cyclic debugging approach implies that a program behaves in the same way every time it is executed. While this is often the case for sequential programs, it is not necessarily directly applicable to parallel programs. Due to the introduction of concurrency, a whole new class of problems and error sources appears. The reasons



Debugging. Fig. 1 Cyclic Debugging [16]

why parallel debugging differs from sequential debugging [16] can be summarized as follows:

- **Increased complexity**

Because of the multiple processes or threads involved and the interactions between them, parallel applications can quickly become complex. Therefore, it is hard to understand their behavior and to locate bugs by only using tools which operate on source code level.

- **Amount of debugging data**

The amount of information which accumulates during debugging parallel applications is significantly larger in comparison to sequential applications. Thus, it can get difficult for the programmer to find the proverbial needle in the haystack, especially if haystacks are increasing.

- **Additional anomalous effects**

Due to parallelism and communication, error sources which do not exist in sequential applications are introduced. Those additional anomalous effects include serious problems like deadlocks and nondeterminism, or related events, where concurrent execution of processes and their interaction is involved.

- **Scalability**

When the limit of frequency scaling was reached roughly in the year 2005, a switch from single- to multi-core architectures took place. This development influenced not only the desktop market, but also made high-end parallel computers with higher core counts possible and necessary – a fact posing scalability problems not only for applications, but also for the debugging process and the associated tools (which usually are parallel programs themselves).

A main source of errors in parallel programs is nondeterministic behavior. This poses serious problems for the cyclic debugging approach introduced above, because nondeterministic applications do not necessarily produce the same outcome or execution path for the same input data on repeated runs. This so-called irreproducibility effect sometimes renders reruns of an application to narrow down the location of a bug useless.

There are various reasons for nondeterministic behavior of an application – a common and usually intended source is, for example, a random number generator. An often unintended effect is a race condition. Resulting from varying execution speeds of the parallel processes or network jitter, if two processes send a message to a third one, the outcome can vary depending on which message arrives first.

A possible solution for the elimination of the irreproducibility effect are mechanisms such as instant replay [19]. During a so-called record phase, information from the initial program run, containing ordering information for critical events, is gathered. Afterward, this information is used to replay the previously observed behavior, thus ensuring that the same execution path is used for repeated application runs.

Breakpointing in Parallel Programs

One of the essential functions used during cyclic debugging is breakpointing – the process of setting breakpoints. A breakpoint can be defined as follows:

Definition 3 Breakpoint

A *breakpoint* [31] is a controlled way to force a program to stop its execution. Breakpointing may occur on software interrupt calls, on calls to a program subfunction, or on selected points within the program.

Breakpoints are set intentionally to interrupt a program's execution at a state interesting to the developer. During this interruption, the environment and the state of the application can be arbitrarily inspected and even modified. Furthermore, it is possible to set additional breakpoints for future interruptions.

There are two common types of breakpoints: instruction breakpoints and data breakpoints. The former offer the possibility to interrupt the execution of an application before a specified instruction is reached. The latter, also called watchpoints, allow to stop when a memory address is accessed, or a specified value is assigned to a variable. In other words, instruction breakpoints enable a control-flow-oriented debugging strategy, while watchpoints allow data-oriented debugging [4]. Additionally, some debuggers allow to specify conditions for breakpoints. In the context of breakpointing, a condition is a boolean expression which is evaluated every time the breakpoint is reached. Only if the expression is true the application is stopped. The so-called ignore count is a special case of breakpoint condition, which only stops the application when a

breakpoint has been hit a certain, previously specified, number of times [30].

Setting breakpoints in sequential applications is straightforward: When a breakpoint is reached and the conditions are met, the program is stopped. Afterward, when the programmer finished inspecting and modifying the application's state, the application can be resumed. When it comes to parallel programs consisting of multiple processes, breakpointing gets more complicated since the programmer may want to inspect the state and variables of multiple processes. The question arising is: which of the processes to stop and where.

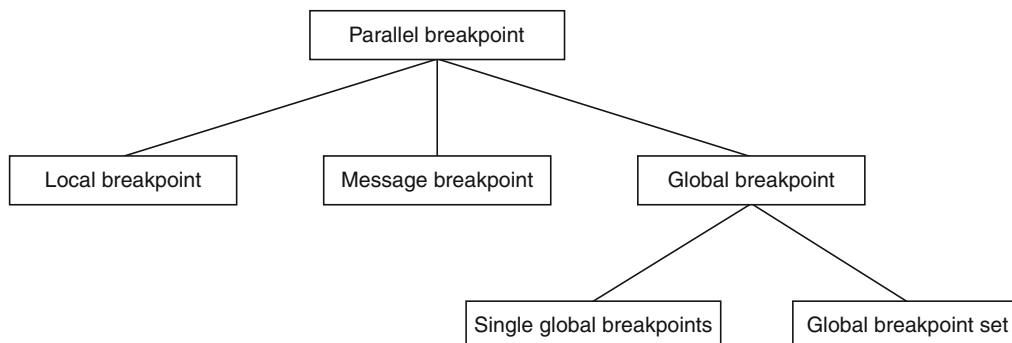
Parallel debuggers typically allow to identify individual processes of a parallel application using numbers reflecting the process number assigned by the parallel runtime environment. Breakpoints for parallel applications can be divided into three main classes (Fig. 2) [10], which can be described as follows:

- **Local breakpoints**

A local breakpoint is only applied to a single process, all other processes are not affected and will continue their execution. This is identical to the sequential case. However, in parallel programs, there are usually relations and dependencies to other processes.

- **Message breakpoints**

A message breakpoint stops all processes involved in a single communication event. In point-to-point communication this typically involves two processes. However, in case of collective operations any number of processes can be part of a message breakpoint.



Debugging. Fig. 2 Classification of parallel breakpoints [10]

- **Global breakpoints**

A global breakpoint involves multiple processes and its classification can be further divided into a single global breakpoint and a global breakpoint set.

- **Single global breakpoint**

A single global breakpoint has exactly one owner process, the respective process on which it was hit. This owner process is stopped at the location or state defined by the breakpoint. The other processes of the parallel application are stopped immediately afterward. They are, however, stopped at undefined states, because of factors like varying execution speed.

- **Global breakpoint set**

In contrast to the single global breakpoint, the global breakpoint set consists of multiple local breakpoints. For this reason, when the global breakpoint is reached, the involved processes are stopped at defined states. However, to accomplish this, each local breakpoint needs to be set at a specific location.

Breakpointing is an important concept in the process of pinpointing the origin of a bug and therefore a central aspect of the debugging process.

Parallel Debugging Tools

While for relatively short and simple programs it may be sufficient to carefully read the code or to insert print statements for locating mistakes, it is not adequate for complex programs like large-scale parallel scientific simulations. As mentioned above parallel applications are not only more error-prone, but locating the error source is also more challenging. For this reason, more sophisticated tools than the so-called printf debugger are required.

There is a variety of tools available for debugging parallel applications. Debugging tools, which intend to provide useful knowledge about a program's execution and the occurring program state changes [34] to the developer, can be divided into three major classes:

- Run-time checking tools
- Program visualization tools
- Traditional debuggers

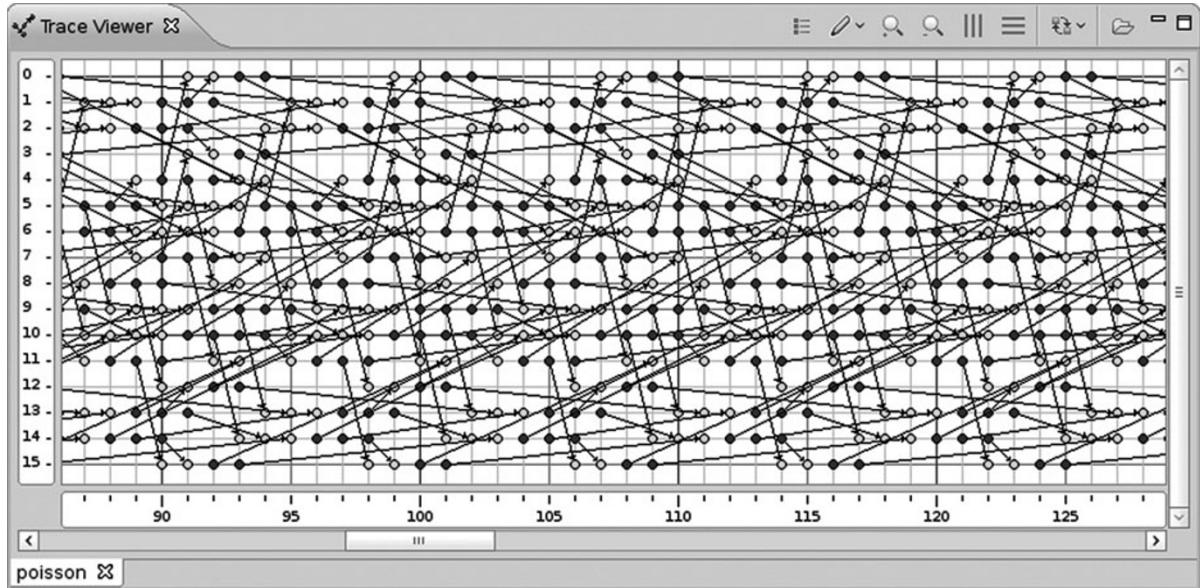
Run-time checking tools provide automated mechanisms to check for errors during the execution of a

program. These tools are typically designed for certain programming interfaces or specific error classes. The used parallel programming interface often adds considerable complexity to the development of concurrent applications. A common example is the Message Passing Interface [23, 24] (MPI), which provides developers with great freedom but also leaves much room for mistakes. As a consequence run-time tools checking the correct usage of MPI have been developed. Prominent examples for this class of tools are Marmot [15], Umpire [33] and their successor MUST [7], which detect invalid arguments for MPI function calls, race conditions, and deadlocks, but also portability issues between different MPI implementations. Another example class of run-time tools are so-called memory checkers. The Intel Parallel Inspector [9] or the Valgrind tool suite [29], help detecting potential memory bugs, like usage of uninitialized memory, memory leaks, and allocation/deallocation errors.

Program visualization tools provide developers with a way to cope with the increased complexity of parallel applications resulting from communication between concurrent processes. Depicting application flow and relations of parallel processes as space-time diagram, like, for instance, the VAMPIR [26] performance analysis tool does, can aid program understanding, and help to detect errors in an application's communication behavior.

The Trace Viewer [12] is a visualization tool based on the space-time diagram and focused on debugging. It implements the event graph paradigm [16], and enables to visualize and analyze communication of message passing programs based on traces [13] recorded during run-time. Figure 3 illustrates the dependencies and the message flow of a parallel message passing program. The vertices represent the events which are located at specific points in time and highlight state changes of a process. The edges represent the continuous transition from one state to another. This includes not only local events, but also the communication flow between processes.

Traditional debuggers are tools allowing to look at and to analyze other programs interactively while they are running [34]. As such, these debuggers support programmers in searching for the reasons for an application's abnormal and unexpected behavior and can be defined as follows:



Debugging. Fig. 3 Event graph visualization [12]

Definition 4 Debugger

A debugger [28] is a tool to help track down, isolate, and remove bugs from software programs.

Rosenberg further proposes several principles a good debugger should at least try to adhere to [28]. These principles can be summarized as following:

- Observing an application with a debugger inevitably changes the debuggee's behavior (sometimes called Heisenberg principle [20]). A debugger, however, should minimize its intrusion on the debuggee.
- The second principle asserts that a debugger must provide truthful information. It is very important that the debugger can be trusted and that it does not lead the programmer into a wrong direction.
- Another prerequisite for a good debugger is that it presents the provided debugging information to the user, together with information where in the execution the program is and how it got there.
- The final principle states the unfortunate fact that debuggers are usually not as technologically advanced as the programmer would need them to be.

The functionality provided by traditional interactive debuggers allow to start and stop an application under their control [34]. Furthermore, it is possible to attach

a debugger to already running applications, but also to disconnect it, thus ending the influence of the debugger on the application. After inspection, the debugger can, on behalf of the user, signal the program to continue, to be aborted, or even to perform a single instruction or execute a source code statement – enabling to trace the execution path of an application. As such, debuggers can be classified into machine-level debuggers and source-level debuggers. As the name already indicates, the former operate on a very low level, providing only assembly information. Source-level debuggers need the debuggee to contain additional symbolic information, hence they are also called symbolic debuggers. This additional information can be added during the compilation of the application, usually by adding a specific compiler flag. Source-level debuggers provide, in addition to the assembly view, means to track the execution progress in the original source code.

Concerning support for parallelism, one needs to distinguish between thread and process level parallelism. While most of today's interactive debuggers support threaded applications, only few provide support for multiple processes. However, there are both open source and commercial debuggers with support for process level parallelism available; for example, MPI programs. The most widespread parallel debuggers include Allinea's DDT [1] and Rogue Wave's Totalview [32],

both commercial products. Well-known open source solutions include the Eclipse Parallel Tools Platform (PTP) [27], and the g-Eclipse [12] debugging functionality. Additionally, several vendor-specific debuggers, usually distributed together with and optimized for certain hardware, are available.

The listed debuggers provide functionality similar to single process debuggers, some of them are actually merely a scalable frontend to multiple instances of a single process debugger, like the popular open source GNU Project Debugger [30] (GDB). Parallel debuggers, however, allow to perform debugging operations like breakpointing, stopping, stepping, and variable and array inspection not only on individual processes or threads but on defined process groups or the whole parallel application. They provide comparison means to analyze differences in the content of variables and in program states between processes of a parallel application.

Parallelizing applications usually requires to partition the data to be processed. Array inspection and visualization tools (Fig. 4) provide useful insight into distributed multidimensional arrays, and assist in the search for data distribution errors [17]. Additionally,

support for filtering and gathering statistical information on data array values, a useful feature for locating bugs in computation, is provided by the aforementioned parallel debuggers.

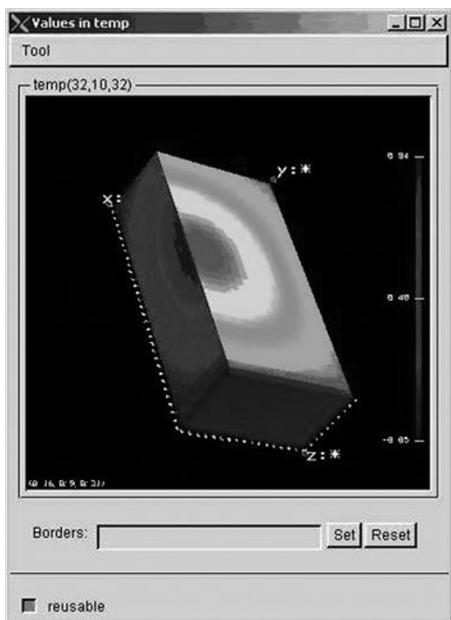
Future Directions

The future of parallel program debugging is clearly connected to the evolution of parallel computing itself. To exploit future computational resources, applications will have to adapt to hierarchical hardware structures, as well as utilize accelerators. Developers will have to incorporate a variety of programming models simultaneously into their applications, and the mixture of these models will make the already demanding task of developing parallel applications even harder and more error-prone. With multi- and many-core architectures coming to the desktop, parallel program debugging will also become more important to a wider audience. Faced with such complexities, debugging will become a major problem, in particular at scale. This poses scalability problems, not only for tools launching and internal communication [21], but also for the analysis and presentation of the debugging information.

Current approaches include using filtering tree-based communication structures [2] for scalable tool communication, however, even if the debugging tools scale, the developer, who has to sift through the data manually, might not. Visualization of program runs with more than a few thousand processes is also not an option without further abstraction techniques, like for instance pattern matching algorithms supporting communication analysis and debugging [14].

Another issue will be that the traditional interactive debugging approach with several reruns is hardly applicable, if not impossible, on large program runs, since it is unmanageable and simply too expensive in terms of time, power, and ultimately money. The Stack Trace Analysis Tool [3] (STAT) tries to address this problem using a lightweight approach for sampling and merging stack traces, with the goal of providing a preselection of processes for further analysis using a traditional interactive debugger.

The focus of future debugging tools must lie on algorithms that automatically analyze the gathered data. Only this will make debugging tools ready for upcoming computing systems by preprocessing and



Debugging. Fig. 4 Array visualization using the MAD environment [17]

abstracting the presented information, assisting and guiding the developer through the error detection and error removal processes.

Related Entries

- [Checkpointing](#)
- [Deadlocks](#)
- [Intel® Parallel Studio](#)
- [Parallel Tools Platform](#)
- [Race Conditions](#)
- [Tracing](#)

Bibliographic Notes and Further Reading

Further general information on debugging can be found in [5], [28] and [31]. Approaches for debugging and visualization of parallel message-passing applications are presented in [16]. A documentation and introduction to using the widespread GNU Project Debugger is freely available [30]. Information on specific parallel debuggers can be found in [1], [27] and [32].

Bibliography

1. Allinea DDT (2010) <http://www.allinea.com>. Accessed 15 Nov 2010
2. Arnold DC, Pack GD, Miller BP (2006) Tree-based overlay networks for scalable applications. International Parallel and Distributed Processing Symposium, Long Beach, CA
3. Arnold DC, Ahn DH, de Supinski BR, Lee GL, Miller BP, Schulz M (2007) Stack trace analysis for large scale debugging. International Parallel and Distributed Processing Symposium, Long Beach, CA
4. Copperman M, Thomas J (1995) Poor man's watchpoints. SIGPLAN Notices 30(1):37–44
5. DiMarzio JF (2007) The debugger's handbook. Auerbach Publications. Boston, MA
6. Gottbrath C (2008) Reverse debugging with the TotalView debugger. In: Cray Users Group Conference Proceedings, Helsinki, Finland
7. Hilbrich T, Schulz M, de Supinski BR, Müller MS (2009) MUST: a scalable approach to runtime error detection in MPI programs. In: Müller M, Resch M, Schulz A, Nagel W (eds) Tools for high performance computing, proceedings of the 3rd international workshop on parallel tools for high performance computing, Dresden, September 2009. ZIH, Springer, Berlin
8. Hovemeyer D, Pugh W (2004) Finding bugs is easy. SIGPLAN Notices 39(12):92–106
9. Intel Parallel Inspector (2010) <http://software.intel.com/en-us/articles/intel-parallel-inspector/>. Accessed 15 Nov 2010
10. Kacsuk P (2000) Systematic macrostep debugging of message passing parallel programs. Future Gen Comput Sys 16(6):609–624
11. Kidwell PA (1998) Stalking the elusive computer bug. IEEE Ann Hist Comput 20(4):5–9
12. Klausecker C, Köckerbauer T, Preissl R, Kranzmüller D (2008) Debugging MPI programs on the grid using g-eclipse. In: Resch MM, Keller R, Himmler V, Krammer B, Schulz A (eds) Tools for high performance computing, proceedings of the 2nd international workshop on parallel tools for high performance computing, Stuttgart, July 2008. HLRS, Springer, Berlin
13. Knüpfer A, Brendel R, Brunst H, Mix H, Nagel WE (2006) Introducing the open trace format (OTF). In: Vassil A, van Albada G, Sloot P, Dongarra J (eds) Computational science ICCS 2006, vol 3992 of lecture notes in computer science. Springer, Berlin, pp 526–533
14. Köckerbauer T, Klausecker C, Kranzmüller D (2010) Scalable parallel debugging with g-eclipse. In: Müller MS, Resch MM, Schulz A, Nagel WE (eds) Tools for high performance computing 2009. Springer, Berlin, pp 115–123
15. Krammer B, Bidmon K, Müller MS, Resch MM (2004) MAR-MOT: an MPI analysis and checking tool. In: Joubert GR, Nagel WE, Peters FJ, Walter WV (eds) Parallel computing - software technology, algorithms, architectures and applications, volume 13 of Advances in parallel computing, North-Holland, Amsterdam, pp 493–500
16. Kranzmüller D (2000) Event graph analysis for debugging massively parallel programs. PhD thesis, GUP Linz, Joh. Kepler University Linz, Austria. <http://www.mnm-eam.org/~kranzlm/documents/phd.pdf>. Accessed 15 Nov 2010
17. Kranzmüller D, Rimmac A (2003) Parallel program debugging with MAD: a practical approach. In: Proceedings of the 2003 international conference on computational science, ICCS'03, Springer, Berlin, pp 201–210
18. Krawczyk H, Wiszniewski B (1998) Analysis and testing of distributed software applications. Taylor & Francis, Bristol
19. LeBlanc TJ, Mellor-Crummey JM (1987) Debugging parallel programs with instant replay. IEEE Trans Comput 36:471–482
20. LeDoux CH, Parker DS (1985) Saving traces for Ada debugging. In: Proceedings of the 1985 annual ACM SIGAda international conference on Ada, SIGAda '85. Cambridge University Press, New York, pp 97–108
21. Lee GL, Ahn DH, Arnold DC, de Supinski BR, Legembre M, Miller BP, Schulz M, Liblit B (2008) Lessons learned at 208K: towards debugging millions of cores. SC2008, Austin
22. McDowell CE, Helmbold DP (1989) Debugging concurrent programs. ACM Comput Surv 21(4):593–622
23. Message Passing Interface Forum: MPI: a message-passing interface standard (1995) <http://www mpi-forum.org/docs/mpi-10.ps>. Accessed 15 Nov 2010
24. Message Passing Interface Forum: MPI-2: extensions to the message-passing interface (1997) <http://www mpi-forum.org/docs/mpi-20.ps>. Accessed 15 Nov 2010
25. Myers GJ, Sandler C, Badgett T, Thomas TM (2004) The art of software testing, 2nd edn. Wiley, New Jersey
26. Nagel WE, Arnold A, Weber M, Hoppe HC, Solchenbach K (1996) VAMPIR: visualization and analysis of MPI resources. Supercomputer 12:69–80

27. PTP – Parallel Tools Platform (2010) <http://www.eclipse.org/ptp>. Accessed 15 Nov 2010
28. Rosenberg JB (1996) How debuggers work: algorithms, data structures, and architecture. Wiley, New York
29. Seward J, Nethercote N, Weidendorfer J (2008) Valgrind 3.3 - advanced debugging and profiling for GNU/Linux applications. Network Theory Ltd, Bristol
30. Stallman R, Pesch R, Shebs S et al (2010) Debugging with GDB, 9th edn. Free software foundation, Boston
31. Stitt M (1992) Debugging – creative techniques and tools for software repair. Wiley Professional Computing Series. Wiley, New York
32. TotalView (2010) <http://www.totalviewtech.com>. Accessed 15 Nov 2010
33. Vetter JS, de Supinski BR (2000) Dynamic software testing of MPI applications with umpire. In: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM), Supercomputing '00. IEEE Computer Society, Washington
34. Von Kaenel PA (1987) A debugger tutorial. SIGCSE Bulletin 19(4):40–44

DEC Alpha

JOEL EMER, TRYGGVE FOSSUM
Intel Corporation, Hudson, MA, USA

Synonyms

Microprocessors

Definition

The DEC Alpha was an architecture and line of microprocessors created by Digital Equipment Corporation (DEC). These processors, which began shipping in the early 1990s, were among the most innovative and highest performing of their era. The 64-bit Alpha processors were the successor to DEC's highly successful VAX product line and ran VAX legacy code, though translation, on new incarnations of the company's VMS operating system. Alphas also ran an OSF1-derived version of Unix called Tru64, and Microsoft Windows NT ran natively on the Alphas with support for x86 programs via binary translation.

Background for Alpha

The Alpha architecture arose inside DEC in the late 1980s out of recognized challenges with the company's then-current VAX product line. VAX design projects

were facing difficulties in meeting their schedules and the designs were becoming overly complex for the relatively small DEC design teams. As a result VAX designs, which had had their heyday during the 1980s, were starting to lag behind their competitors on cost and performance. But the VMS operating system and a substantial software ecosystem, which ran on VAXes, were very important to DEC, representing a large customer base and a significant revenue stream.

The VAX architecture had been developed around a philosophy that was premised on code being written in assembly language. In the late 1970s when the VAX architecture was being specified, most large software systems were written in assembly language to achieve adequate efficiency. The VAX instruction set made this easier by having instructions which performed sophisticated operations and provided a universally available set of addressing modes for each operand of the instruction. Improved compilers, however, undermined the assembly language premise on which the VAX was based. In this evolution, the VAX was at a disadvantage because the VAX instructions were over-specified, and it was difficult for compilers to map programming language idioms into the available instructions. Even worse, the VAX instructions provided semantics beyond that needed by the compiler but which had to be paid for in the implementation. This resulted in designs that were more complex and slower than needed.

The VAX architecture with its sophisticated instructions and powerful addressing modes became known as a *complex instruction set computer* or CISC-style architecture. But the 1980s also had seen the beginning of the *reduced instruction set computer* or RISC revolution. RISC architectures were based on the notion that a computer architecture should be good target for compilers by providing exactly the primitive operations that a compiler needs and that also map naturally to simple and efficient hardware implementations. The RISC work was spearheaded by research at IBM, Stanford, and Berkeley, who each developed simple, powerful processor chips quickly. This work led eventually to IBM's PowerPC processors, spawned MIPS as a company, and SPARC as the basis for Sun's computer line.

The attraction of RISC-based architectures was not lost on DEC's engineers. As a result, there were several early RISC projects at DEC. These included Titan,

a RISC-based research project; SAFE, a RISC architecture explored within the development organization; and Prism, which formed a RISC-style architectural basis for an entire software system that was an evolutionary reimplementation of the VMS operating system. DEC also built and marketed systems based on the MIPS architecture. Those systems used MIPS designed chips, but there were MIPS processor designs in flight at DEC when all RISC development efforts were redirected toward Alpha.

A flaw common to all of DEC's initial RISC-style design efforts was that they were not believed to be able to preserve the VAX/VMS-based market, so a task force was formed to address that deficiency. Thus, the primary charter of the Extended VAX, or EVAX, taskforce was to propose a means of modernizing the VAX architecture in a way that preserved the software base in VMS and in DEC's Ultrix (Unix) market. Modernization, of course, implied that it had to gain the efficiency/complexity benefits of the competing RISC architectures and thus this group was often referred to as the RISCy VAX taskforce.

The RISCy VAX taskforce considered a number of design alternatives. Since a key criterion was preserving software compatibility, a number of the alternatives also preserved the VAX architecture directly. For example, one alternative was to have a *heterogeneous design* in which new systems would have a small VAX core running the OS and legacy applications, with a more modern RISC core running performance critical applications, while sharing virtual memory. While such a design preserved compatibility, it added to a RISC design the cost and complexity of a VAX design whose performance would still be a critical component of overall performance.

To mitigate some of the cost of multiple heterogeneous cores, another proposal examined *multi-execution path cores*. In this variation a single core could execute both VAX and RISC instructions by providing two front ends: one interpreting VAX instructions, and another dealing with the new RISC ISA. This was how VAX evolved from the PDP-11, but in that case the microcode complexity was already being paid for, which was something not needed for a pure RISC core.

Some years earlier DEC had already made a simplification of the VAX architecture through the removal some of its most complex operations. This

simplification facilitated the creation of single-chip-microprocessor VAXes. These were the highly successful MicroVAXes. Since reducing the ISA had been successful before, why not try it again?

The *fixed length instruction VAX subset* proposal suggested carefully selecting and implementing only a subset of VAX instructions, all fixed length and with other RISC-like attributes. Note, however, that limiting instructions to only 32-bits (like other RISC architectures) would have resulted in a load with an inadequate 8-bit offset, so a small number of different length instructions were included. Using such a subset architecture, DEC could design a simpler, faster VAX. As with MicroVAX, this scheme dealt well with backward compatibility, but would require some form of emulation for forward compatibility of instructions no longer implemented in hardware. Unfortunately, analysis showed that a number of factors, such as the small architectural register file set and the implementation complexity of multiple instruction sizes, would destine this proposal for subpar performance.

A final hardware solution proposed a hardware unit that would do *hardware instruction translation* of VAX instructions into RISC instructions and then execute those instructions in a pure RISC-style back end. This strategy was approximated in some VAX designs, leaving much of the CISC operations to the front end for simple instructions, while more complex instructions were handled in microcode. Such a scheme was used in the mid-1990s by Intel to implement out-of-order execution in its Pentium Pro processor and has been the basis of many subsequent Intel processors.

After discussing this set of less than perfect hardware solutions to the problem, it was decided to save the hardware of the hardware-based instruction translation scheme and leave it to software to facilitate the migration from VAX to Alpha. Through a combination of interpretation, static and dynamic binary translation, compilation of Macro32 assembly language using a new Macro32 "compiler," and recompiling of higher-level language programs, software would solve the problem. Relegating migration to software freed the architecture team to focus on what they believed to be the best RISC architecture ever without significant legacy constraints.

To make migration to Alpha easier, a few hardware hooks were architected in for VAX support, but

not many. Several were easy to incorporate such as identical protection rings, memory protection codes, and interrupt priority levels. The VAX page size of 512 bytes was already resulting in less efficient page translation and complicated first level cache design, so was a sticky point. Therefore, it was decided to immediately start aligning VAX pages on 8 KB boundaries in the linker two years before Alphas were to start shipping to allow Alpha to be architected with a minimum page size of 8 KB.

Many compatibility issues were also handled with special code called PALcode. A program enters PALcode with a special instruction that dispatches into the entry point for an implementation-specific PALcode routine. PALcode, which looks like regular instructions, but with extra privileges, some implementation-specific instructions, and special registers, could perform special operations needed for various system activities but also could be used to support complex VAX operations. Special instructions could even be used to bracket sequences of operations that were to be executed atomically.

PALcode also provided an easy place to provide various other system functions and implementation-specific functions. This use is similar to Itanium's PAL code.

Alpha Architecture

Like all RISC architectures, the Alpha architecture was designed with a set of simple register-based operations and separate instructions to move memory values to and from the registers. As with many other RISC architectures, the instructions are of a fixed length of 32 bits. Alpha instructions came in just four forms:

- Register-to-register operations
- Memory operations
- Branches
- Dispatches to PALcode

The Alpha instructions were also designed to map efficiently to a hardware implementation and hence were extremely regular. To keep them regular, the operands are in fixed locations. Furthermore, instructions have at most two sources operands and no more than one destination. This greatly simplified the register file, operand tracking tables, and pipeline organization, since there never was a need to access, track, or transport more

than two input values and one output value for an instruction.

A recurring issue related to the two-input/one-output principle was the proposal for adding a Floating Multiply and Add instruction (FMA). FMA allows for more efficient implementation of its constituent operations and saves instruction issues for an operation sequence that occurs frequently in some codes. But FMA also requires a minimum of three sources and a destination. Thus it violated the basic design principles and was left out of the architecture.

With 32-bit instructions, providing a reasonable number of operations and no more than three register operands, the number of architectural registers is restricted to 32. However, by using separate 32-entry register files for integer and floating point instructions, Alpha gained an extra implicit register specifier bit, resulting in a total of 64 architectural registers for computation.

The majority of Alpha architectural state is the 64 integer and floating point registers. Two of these registers, R31 and F31, were hardwired to zero. Thus reads of these registers always returned a zero and writes were ignored. Allowing writes to these registers did, however, allow for special instruction semantics. For example, loads to R31 served as a memory prefetch. Additional architectural state consisted of shadow registers for integer registers that appeared in kernel mode. These saved the costs of explicit register saving by providing some scratch space on entry to kernel routines.

Another piece of architectural state is a floating point mode control register. This register was a compromise that involved saving the use of precious bits in each floating point instruction to specify the mode, e.g., rounding mode, and having library routines inherit a mode from the caller (with unpredictable accuracy results) or check/save/restore the mode (at a significant performance penalty for small routines).

The program counter (PC) is also architectural state, but unlike on the VAX the PC does not appear as a numbered general purpose register. Alphas also have a status register that holds information about such things as the current and previous ring level and interrupt priority level. But in contrast to the VAX the Alpha architecture does not have an explicit register to hold condition codes. This both saves on the bottleneck of having a single register receiving results from many

instructions and preserves the principle that an instruction only reads two registers and writes one register thus simplifying dependency tracking.

Avoid Complex Hardware Features

In a reaction to the complexity of VAX instructions, the Alpha architecture was designed to avoid features that would result in complex hardware. For an operation to be included in the Alpha ISA as an instruction, it had to be shown to occur frequently in important programs, that it could not be done effectively by sequences of other instructions, and that it fit the Alpha model of simple hardware implementation. Integer multiply was debated, but was finally included. Integer divide was found to be a rare operation, often dividing by a constant, equivalent to a handful of shifts and adds, and hence not included.

A significant, and maybe the most controversial decision based on complexity versus frequency of occurrence, was the decision to leave out load and store operations for less than 32-bit values (longwords). This meant that 8-bit values (bytes) and 16-bit values (words) were not accessible except as part of a longword. Stores of a byte required a read, modify, write sequence. The statistics showed byte memory operations to be rare, and that the shifts involved in lining up an individual byte in a 64-bit datapath would slow down loads and stores in general. Byte access can also complicate the implementation of error checking, for example, via ECC. The absence of 8 and 16 bit load/store support turned out to be more costly than had originally been envisioned. The compiler's lack of ability to determine memory aliasing of the bytes being gratuitously read (and re-written) near a byte of interest was a major impediment to code reordering. Thus instructions to load and store both bytes and words were added in the shrink of the second generation Alpha, EV56.

Floating point square root instructions and support for video encoding were added in the third generation Alpha, EV6. Square root was chosen because it was a common operation and could be executed by a small tweak to the floating point divide hardware. Video and multimedia support had emerged as performance-critical applications. A few, simple instructions turned out to make a big difference for performance.

In order to simplify exception handling hardware, arithmetic exceptions are architected to be imprecise in

Alpha. This lets later instructions complete before overflow and underflow conditions are known. To give software the opportunity to rationally respond to such conditions, a barrier instruction was provided that would force instruction issue to wait for all pending exceptions to be resolved. Thus software could achieve the effect of precise exceptions by placing a barrier instruction at the end of an idempotent sequence of instructions that followed any instruction that might cause an exception. This approach simplifies hardware, but does complicate software design. It is interesting to note that with out-of-order execution, register renaming made precise exceptions practically free.

The Alpha design teams also considered hardware support for speculative operations. This tends to create some complexities, mainly around exceptions. But there were some benefits from being able to move operations past branches (control speculation), and to a lesser degree from moving loads past stores (data speculation). Ultimately, it was found that most of these benefits could be achieved without hardware support, but by dealing with spurious exceptions in software. Such a framework was developed in Tru64 UNIX.

Hardware Implementations

The Alpha architecture was implemented in a series of hardware designs. The principal numeric designation for these designs was 21< n >64, where the 21 stood for the twenty-first century (since Alpha was touted to be the architecture for the next 25 years), the 64 indicated that this was a 64-bit architecture and < n > indicated the generation of the design starting with 0.

Internally, the Alpha designs were referred to by code names in the form EV< n >. The EV prosaically stood for Extended VAX, although it has been suggested that EV stood for other more colorful things. The < n > stood for the DEC CMOS process generation, thus EV4 (which was sold as the 21064) was implemented in CMOS4, an 0.7 μ process. When a design was largely remapped into a later process the process generation of the new process was appended to the code name. Thus, EV45 was a shrink (with some design changes) of the EV4 design into the CMOS5 process. Unfortunately, as the goal of having a complete new design in each process generation proved unsustainable, the < n > became simply a monotonically increasing designation of new designs. Still the EV nomenclature conveys more

DEC Alpha. Table 1 Alpha timeline

Year	Processor	Process	Frequency	Features
	EV3	1.0 μ		Software development vehicle
1992	EV4	0.7 μ	200 MHz	Two-wide superscalar
1995	EV5	0.5 μ	350 MHz	Four-wide superscalar
1998	EV6	0.35 μ	600 MHz	Out-of-order, Multimedia
2004	EV7	0.18 μ	1.15 GHz	Integrated system functions
	EV8	0.13 μ	1.8 GHz	Eight-wide superscalar, SMT
	EV9	0.09 μ		Vector processing

information about a design, so this article will typically use it for describing the various Alpha implementations (Table 1).

EV3: Software Development Vehicle

To assist software development and allow exploration of high-speed circuit techniques, a non-commercialized implementation of the Alpha architecture called EV3 was developed. Following the naming convention described above, EV3 was implemented in DEC CMOS3 process, which was a 1.0 μ process. The EV3 design largely matched the first commercially sold Alpha, EV4, but in order to fit in a single chip in the less spacious CMOS3 process, some nonessential functions were left out or reduced in size. For example, floating point operations were emulated in software. The caches were also only 1 KB. EV3 was built into the Alpha Development Unit (ADU), which was available for early software development. EV3 proved very valuable in the successful launch of Alpha as the fastest microprocessor in the industry.

EV4: First Commercially Available Design

The first commercially available Alpha microprocessor was EV4. It was launched at an astonishing clock frequency of 150 MHz, in DEC's CMOS4 0.75 μ process at a time when the industry standard was well below 100 MHz. The functionality, frequency, and overall

performance of EV4 made it an eye opener for the industry. It started an era of rapid improvements in CMOS circuit speed. An updated version of EV4 ran at 200 MHz.

Overall, the software transition from VAX to Alpha went smoothly due to the careful preparation of tools to assist in porting both OS functions and applications. Several systems were designed around EV4, including an Alpha PC, multiple workstations, and low and high end servers.

Both UNIX and VMS were supported. Introducing a 64-bit architecture turned out to be complex. When VAX was introduced as a 32-bit architecture 15 years earlier, the market had clearly outgrown 16 bits. The need for 64 bits was less pressing. DEC decided on a split strategy: 32-bit VMS, and 64-bit UNIX. Pioneering 64 bits was heavy lifting for UNIX, as many applications did not port easily from 32 bits. This had a mixed impact on Alpha as reduced application availability clearly slowed Alpha acceptance. The larger memory footprint due to the increased size of address pointers also had a detrimental impact on some applications. On the other hand, a 64-bit address space proved a valuable asset in some application domains, such as databases, where Alpha did shine. Alpha's pioneering porting efforts also clearly aided today's 64-bit architectures by getting applications ready.

Considering the classic formula that expresses processor performance as a function of frequency of operation, cycles per instruction and number of instructions, the Alpha architecture was most focused on facilitating high-frequency operation with instructions that required few cycles to execute. Designs exhibiting such characteristics were dubbed "speed demons" and the EV4 exhibited these characteristics through a design philosophy of providing speed through high frequency implementation in a simple, relatively shallow pipeline.

Although the EV4 frequency of operation was a function of many components, considerable effort was expended on creating a 64-bit adder that could be cycled as fast as possible and was intended to set the frequency of the design. Thus, other less frequent operations, such as shifts, were allowed to take multiple cycles. Considerable design effort was also expended on the high frequency clocking network to minimize clock skew across the chip – as a consequence approximately half the power of the design went into the clock network.

High frequency operation was also facilitated by avoiding stage-to-stage flow control in the main pipeline. This eliminated complicated cascading stall signals and reduced delay in each stage of the pipeline and thus allowed for more computation logic in each stage.

To control the pipeline, an end-to-end flow control was implemented in which instructions in the early stages of the pipeline always flow systolically and an execution unit stage would notice the need for a stall. But rather than stall the immediately preceding stage of the pipeline the execution unit would redirect the front of the pipeline to stall and buffer up the next few instructions coming down the pipeline and replay them when the stall condition had been resolved. This buffer was referred to as a “skid buffer.”

Since the design of the Alpha architecture and the EV3/4 micro-architecture were designed contemporaneously, a number of micro-architectural considerations entered into the architecture. This included the placement of operand fields in the instruction and assignment of operation codes that were easy to map into a programmable logic array or PLA. The instruction set was also divided into classes that used disjoint sets of function units to facilitate the creation of superscalar designs that could issue multiple instructions simultaneously. Superscalar implementation was also simplified through the absence of a MIPS-style branch delay slot or a VAX-style condition code register.

EV4 was a superscalar design, issuing up to two instructions per cycle. There were no redundant functional units, so the two instructions issued had to be a combination of single load/store, integer, branch, or floating point instructions. The chip had an 8 KB instruction cache and an 8 KB data cache on die, with support for a board level cache.

A challenge in accommodating specific micro-architectural optimizations into the specification of an architecture is the impact such features have on future micro-architectural innovations. The Alpha’s conditional move instruction (CMOV) is an interesting case in this regard.

The CMOV instruction takes a source operand and a condition, and copies the source to the destination if the condition is true. In an in-order design, this seems to fit the two sources and a destination model.

It does, however, have the implicit assumption that the destination register is always physically in the same location. But even in an in-order pipeline the destination *value* of a CMOV is not always in the same place when compared to other arithmetic operations. Therefore CMOV instruction results cannot be bypassed and have a latency greater than that of other simple arithmetic operations.

In an out-of-order pipeline, the assumptions on which the CMOV instruction was premised are even less true. In an out-of-order design with register renaming, the newly assigned destination register does not automatically contain the old value when the condition is false. Thus the old value in effect becomes a third source, violating a key Alpha design principle. This became an issue in EV6 and EV8, the out-of-order Alpha designs. Rather than require costly extra register ports and other changes to the execution pipeline, the CMOV instruction was broken into two separate instructions in the instruction fetch stage.

Despite these implementation challenges the CMOV instruction was found valuable by software. Since the CMOV instruction can be viewed as a specific case of a predicated operation, since a little predication was found to be valuable the team considered adding more predication. At the time, there were several papers published describing significant IPC improvements from predication all instructions. Additional support for predicated operations proposed for inclusion in the architecture. Within the Alpha team, performance studies of such more extensive predication were conducted, but found only negligible benefits beyond what was obtained with the CMOV instruction alone.

Finally, application performance depends on more than just the microprocessor. The system included industry standard memory chips and IO components. The raw execution advantage of EV4 was somewhat diluted by these system components. Not until EV6 did Alpha microprocessors incorporate major advances in memory system, followed by the innovative system design in EV7.

EV45: Compaction of EV4 into .5µ CMOS Technology

Initially, the Alpha design teams maintained a cadence of scaling the previous design into a new process,

followed by a new design in the same process. EV45 was the CMOS5 version of the EV4 design. A similar strategy at Intel is referred to as the Tick-Tock model. EV45 kept the dual issue pipeline of EV4, but the increased size of first level cache from 16 KB and the clock frequency to 275 MHz.

EV5: The Next Generation

EV5 was a major new Alpha design. It was four-wide superscalar, but only if two of these were floating point instructions, an addition and a multiplication. For integer operations, it was two-wide issue. EV5 did relax some of the integer issue rules from EV4, by allowing two load operations or one store through an L1 cache which was implemented as two copies of the data to allow parallel reads. Stores wrote to both copies.

EV5 was an ambitious design. With a total transistor budget of 9.3 million devices, it crammed a lot of functionality into the chip. This was partly made possible by relying on the compiler to schedule code for optimal execution. Multi issue works only when independent instructions are aligned on natural boundaries. Studies showed that more complex issue logic might improve performance in cycles per instruction, but would create tighter timing constraints and make it more difficult for the compiler to schedule for multiple issue. EV5 also relied on the compiler to schedule for resource conflicts. For example, a store followed by a load instruction to the same address in the following cycle causes a replay trap. EV5 took the Alpha philosophy of executing good code very well, while sometimes tripping over legal, but less than optimal code.

EV5 coincided with DEC doing research in VLIW compiler technology. In some ways, it was useful to view EV5 as a VLIW. Having the compiler consider low-level code scheduling had a big impact on performance.

The cache hierarchy was revamped from EV4. The first level caches, I and D, remained at 8 KB each, but they were backed up by an on-die 96 KB, three-way set associative, unified L2 cache. It continued to support an off chip cache as a third level.

EV6: Compacting EV5 into 0.25 μ Technology

The successor, EV56, was possibly the biggest success for the Alpha product line. Re-implementing the EV5

design in the 35 μ CMOS 6 process resulted in a dramatic frequency improvement from 300 to 666 MHz. With the smaller feature size, came smaller die size and lower power, making it easier to manufacture and design into systems. Compilers were getting better at dealing with the idiosyncrasies of the microarchitecture. A fairly major change in EV56 was the addition of byte and word access to memory. This change did not have a negative impact on the cycle time, as had been feared. It had a positive impact on porting application, which often accessed memory in small chunks, and the previously required costly read-modify-write sequences.

EV6: Speed Demon and Brainiac

EV6 was the third major Alpha design. EV6 attempted to maintain the high frequency design of its predecessors. Like EV5, it was superscalar with a peak performance of four instructions per cycle. But the EV6 team aimed to not only preserve those “speed demon” characteristics of the previous designs but also provide sophisticated microarchitectural features of the so-called “brainiacs” of that era.

There were several important microarchitecture features included in EV6. Most striking was the ability to execute instructions out of order. Instructions were fetched, destination registers were renamed, and source registers mapped. They were then placed in an instruction queue, and issued based on when their source operands were ready.

With a deeper pipeline and more operations in flight, the branch predictor became more important. EV6 had a tournament branch predictor, with a global (path history) and local (history of this branch) predictor dueling. A Chooser picked the winner based on the path history. EV6’s branch predictor was far larger and more sophisticated than any of its contemporaries, yet in the design post mortem it was felt that even more size and effort should have been devoted to it.

The complex EV6 branch predictor took multiple cycles to make a prediction and calculate an expected next address. Since the machine needed to fetch a block of instructions each cycle EV6 incorporated a structure referred to as a ‘line predictor’ that could quickly predict a new next instruction block every cycle. This eliminated bubbles in the pipeline waiting for the branch predictor. If the branch predictor disagreed with the

line predictor, then a quick redirect of the front end occurred. Of course, if the branch predictor was found to disagree with the actual instruction during execution then a later redirect of the front end would occur.

EV6 could issue four integer operations in a single cycle, with up to two Load or Store instructions. A remarkable feature was the ability to double pump the first level cache. This allowed for a fully dual ported cache, without bank conflicts. This double pumping mechanism also allowed for reading dirty data out of the cache in the same cycle a block was filled. The virtually indexed cache had a three cycle load to use latency. It was backed up with an off-chip cache with a dedicated interface, eliminating conflicts with traffic to main memory and IO.

To keep the instructions flowing, EV6 relied on speculative execution. It would let issue-ready loads issue before earlier stores, and replay if the addresses conflicted. To prevent future conflicts, a PC-indexed table kept track of past problems and kept loads in order when necessary.

Another interesting issue-related mechanism existed for optimizing the issue of instructions dependent on a value from a load instruction. While the latency of most instructions is deterministic, the latency of a load depends on whether it hits or not. To rendezvous with a hit optimally requires issuing before the machine knows if the load will hit in the first level cache. Since guessing a load is going to hit and having it miss resulted in a costly replay, EV6 had a predictor for the behavior of future loads controlling this load-hit speculation.

The EV6 team set out to combine the raw execution rate of the processor with a matching memory and system interface. EV6 introduced several new instructions to the Alpha architecture. Besides Square Root, and multimedia operations, there were new versions of the memory prefetch instruction, making writes and streaming references more efficient.

For EV6, the Tsunami chipset made big advances in improving memory access. Wide interfaces and point-to-point connections resulted in several Gigabytes per second of memory bandwidth. This design style foreshadowed the link-based systems that followed.

The EV6 design was compacted into new processes: EV67 in 0.25μ , and EV68 in 0.18μ . It was also used as the compute engine in EV7.

EV7: Integrating System Functions

Alpha processors achieved good performance on benchmarks like SPEC, both integer and floating point. In larger applications, some of the processor performance advantage was lost in time spent outside the processor chip itself. With the extra transistors available through the new 0.18μ process, it became practical to integrate the traditional functions of the system chip, such as the memory controller, the network switch, and the network router. In addition, EV7 added a 1.75 MB on die second level cache, and a direct interface to an IO chip. This made for an almost glueless multiprocessor system.

The memory was based on RAMBUS technology, with eight channels connected directly to the CPU chip. This was a further improvement in memory latency and bandwidth, up to 12.8 GB/s. The memory channels could operate in RAID mode, XOR'ing in a redundant channel for high reliability. To keep the caches coherent, there was an in-memory directory which kept track of which cache lines were present in a cache somewhere in the shared memory system. This scheme is relatively simple, but has the drawback of requiring memory bandwidth for directory updates. To reduce this effect, EV7 did not use the directory for local cache accesses. This works very well for programs with good locality. Remote references had to snoop the local cache for possible hits.

The core design itself was an EV6 with only minor changes. The EV7 systems pioneered a new trend in system design. With point-to-point socket connections through high bandwidth links, the performance scaled very well up to 32 sockets with shared, cache coherent memory. This was partly due to a high ratio of memory and link bandwidth (6 GB/s per link) relative to individual socket performance. The plan was to follow up this breakthrough in system design with a new core, EV8, with increased core performance. In practice, the industry trend became multi-core processors, with high aggregate performance, putting pressure on the system functions such as memory and link bandwidth.

EV7 moved the board level cache onto the die. The resulting cache was smaller than the 4 MB cache used in many EV6 systems. Sometimes this impacted performance negatively, but with the higher memory bandwidth and shorter latencies, most applications achieved

good performance improvement with EV7 systems. EV7 showed the importance of overall system design in achieving robust application performance. Even though EV7 entered the market when Alpha investments were being reduced, it had a long product life due to its scalable system design.

EV8: Araña

EV8 was aimed at pushing Alpha processors to ever higher single stream performance. The Alpha architecture goal of enabling superscalar implementations with simple instruction decode and dependence checking would be tested as the EV8 goal was to fortify Alpha's position as the world's highest performing computer architecture. The total transistor budget was now several hundred million transistors and the goal for EV8 was to make good use of them all. The path-finding phase aimed at developing features that would increase the IPC, with relatively little concern for power and area. Performance modeling was aimed at finding the limits for instruction level parallelism. The design could issue eight instructions per cycle, with a wide range of functional units to support an assortment of instruction mixes.

Keeping an eight-wide issue unit busy is a big challenge, which EV8 designers tried hard to meet. The front end could fetch up to 16 instructions per cycle in two chunks of eight aligned instructions, possibly with a taken branch instruction between them. An extensive branch predictor could make up to sixteen predictions in a cycle. A large instruction window tried to feed the functional units with issue ready instructions, usually issued out of order.

One of the messy problems with out-of-order execution results from address conflicts when loads are hoisted above stores. The EV8 team invented the "store set" mechanism, which associated set of loads with sets of potentially conflicting stores, and used the spare register number in the micro-architectural implementation of these operations to encode the dependency.

With performance modeling, it became clear that many interesting applications would not significantly benefit from eight-wide instruction issue. The amount of ILP was just not there, so the architects added "simultaneous multi-threading" (SMT) to the design. Four threads could execute simultaneously in the pipeline,

competing for resources and issue in the same cycle. With multiple threads the instruction window was much more likely to contain issue ready instructions. This was an early implementation of SMT, breaking new ground while solving some interesting problems. A few new instructions were added to support SMT. To keep idling threads from consuming resources, the instruction pair of ARM and QUIESCE set up a memory address to be monitored for updates and allowed a thread to suspend itself while waiting for a write to that location.

With SMT, it became much more practical to utilize CPU resources. Performance scaled well up to four threads. For high IPC applications, there was a dropoff in improvement from three to four threads, but for others, such as transaction processing, the improvement was almost linear with thread count. SMT is an efficient latency hiding technique. While one thread waits for memory, other threads are free to use all the pipeline resources. The extra threads can, of course, put pressure on the shared resources, and will especially increase the cache miss rate due to supporting multiple programs when there is little locality. For EV8, the SMT design became a careful balancing act between multi-threaded performance and the cost of supporting the extra threads in area and power.

There were two principal points in the pipeline with arbitration between threads. Every cycle, a thread selector would pick a thread to fetch instructions from, based on which thread had the fewest instructions active in the early stages of the pipeline. This allowed threads with high IPC to move right along, while allowing slower threads access without clogging up pipeline resources. Another thread selector picked threads for mapping registers based on the register renaming scheme. Once instructions had been mapped, they were put into an instruction issue window shared by all the threads. From there they were selected for issue based on age among issue ready instructions.

While SMT required dedicated architectural register state for each thread, the renaming registers required to fill the instruction pipeline was shared among the four threads. Wherever possible, pipeline structures and caches were shared dynamically between the four threads for optimal throughput performance. Only a few safeguards had to be part of the allocation to prevent

threads from being starved for resources. Overall, SMT proved to more than double through-put performance with less than 10% increase in hardware resources.

EV9: Vectors and Multicore

Although EV8 never shipped, research work was in progress on a variety of successor designs. One alternative, called EV9, was targeted at high-performance technical computing and added an integrated vector unit. Following the naming scheme begun with EV8, which was called araña (spider in Spanish) alluding to both the eight-way issue and SMT threads, EV9 was also called tarantula. The Cray X2 continued this naming tradition with a code name of “black widow.”

Another trajectory examined putting a large number of smaller processors on a ring-based interconnect. This work was pioneered by DEC’s research lab in Palo Alto, code-named Piranha project. The initial design suggested that replacing a single Alpha core optimized for single stream performance with eight simpler cores could improve overall performance on applications such as online transaction processing.

Alpha Systems

Alpha processors were the compute engines in a long array of computer systems. DEC produced systems ranging from PC’s and workstations to high end server systems. TurboLaser was a bus based system with more than 2 GB/s of bus bandwidth used to connect multiple processors with memory and IO. Wildfire was the first link-based system, connecting groups of four processor blocks with a two-level switch interconnect. Marvel was an almost glueless MP system design for EV7 and EV8, using a 2D Torus. The network router and switches were integrated into the processor chips along with the RAMBUS memory controllers.

Alphas were integrated into Cray’s T3D and T3E designs. These systems combined the fast Alpha microprocessor with an external memory system. The nodes were connected with a 3D toroidal mesh network. The ASCI Q system at Los Alamos consisted of a network of a thousand four-socket SMP systems based on 1.25 GHz EV68 processors connected by network switches.

Bibliography

1. Robert M (1993) Supnik, Digital’s Alpha project. *Commun ACM* 36(2):30–32
2. Alpha Architecture Reference Manual (1992) In: R.L. Sites (ed) Digital Press, Bedford
3. Bhandarkar DP (1995) Alpha architecture and implementations. Digital Press, Bedford
4. Sites RL, Chernoff A, Kirk MB, Marks MP, Robinson SG (1993) Binary translation. *Commun ACM* 36(2):69–81
5. Chernoff A, Herdeg M, Hookway R, Reeve C, Rubin N, Tye T, Yadavalli SB, Yates J (1998) FX!32: a profile-directed binary translator. *IEEE Micro* 18(2):56–64
6. McLellan E (1993) The alpha AXP architecture and 21064 processor. *IEEE Micro* 13:36–47
7. Dobberpuhl DW, Witek RT et al (1992) A 200-MHz 64-bit dual-issue CMOS microprocessor. *Digital Tech J* 4(4):35–50
8. Edmondson JH, Rubinfeld P, Preston R, Rajagopalan V (1995) Superscalar instruction execution in the 21164 alpha microprocessor. *IEEE Micro* 15(2):33–43
9. Gronowski P, Bowhill WJ, Donchin DR, Blake-Campos RP, Carlson DA, Equi ER, Loughlin BJ, Mehta S et al (1996) A 433-MHz 64-b quad-issue RISC microprocessor. *IEEE J Solid-State Circuits* 31(11):1687–1696
10. Gwennap L (1999) Brainiacs, speed demons and farewell. *Microprocessor Report* 13(17):1–2
11. Kessler RE, McLellan EJ, Webb DA (1998) The alpha 21264 microprocessor architecture. In: Proceedings of the international conference on computer design: VLSI in computers and processors, Austin, pp 90–95
12. Kessler RE, McLellan EJ, Webb DA (1998) The alpha 21264 microprocessor architecture. In: Proceedings of the international conference on computer design: VLSI in computers and processors, Austin, pp 90–95
13. Kessler RE (1999) The alpha 21264 microprocessor. *IEEE Micro* 19(2):24–36
14. Bannon P et al (2001) Alpha 21364: A Single-Chip Shared Memory Multiprocessor. In: Government Microcircuits Applications Conf. 2001: 334–337
15. Mukherjee SS, Bannon P, Lang S, Spink A, Webb D (2002) The alpha 21364 network architecture. *IEEE Micro* 22(1):26–35
16. Mukherjee SS, Silla F, Bannon P, Emer J, Lang S, Webb D (2002) A comparative study of arbitration algorithms for the Alpha 21364 pipelined router. In: Proceedings of the 10th international conference on architectural support for programming languages and operating systems, San Jose, 5–9 Oct 2002
17. Tullsen DM, Eggers SJ, Emer JS, Levy HM, Lo JL, Stamm RL (1996) Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In: Proceedings of the 23rd annual international symposium on computer architecture, Philadelphia, pp 191–202, 22–24 May 1996
18. Emer J (2001) EV8: the post-ultimate alpha. In: Keynote PACT 2001, Barcelona
19. Emer J (1999) Simultaneous multithreading: multiplying alpha performance. In: Proceedings of micropocessor forum, San Jose
20. Chrysos GZ, Emer JS (1998) Memory dependence prediction using store sets. In: Proceedings of the 25th annual international symposium on computer architecture, Barcelona, pp 142–153, 27 June–2 July 1998

21. Seznec A, Felix S, Krishnan V, Sazeides Y (2002) Design tradeoffs for the alpha EV8 conditional branch predictor. In: Proceedings of the 29th annual international symposium on computer architecture, Anchorage, 25–29 May 2002
22. Espasa R, Ardanaz F, Gago J, Gramunt R, Hernandez I, Juan T, Emer J, Felix S, Lowney G, Mattina M, Seznec A (2002) Tarantula: a vector extension to the alpha architecture. In: Proceedings of the 29th annual international symposium on computer architecture (ISCA '02), Anchorage
23. Barroso LA, Gharachorloo K, McNamara R, Nowatzky A, Qadeer S, Sano B, Smith S, Stets R, Verghese B (2000) Piranha: a scalable architecture based on single-chip multiprocessing. In: Proceedings of the 27th annual international symposium on computer architecture, Vancouver

Decentralization

► Peer-to-Peer

Decomposition

► METIS and ParMETIS

Deep Analytics

► Massive-Scale Analytics

Denelcor HEP

BURTON SMITH
Microsoft, Redmond, WA, USA

Synonyms

Heterogeneous element processor

Definition

The HEP was a general-purpose shared memory multiprocessor system manufactured by Denelcor, Inc. of Denver, Colorado. Design began in 1973 and a total of six systems were delivered between 1981 and 1985, the year Denelcor closed its doors.

Discussion

Initial Design

The HEP computer system was originally conceived as a digital replacement for the analog computers that were Denelcor's traditional products. The objective was to achieve comparable performance in doing what analog computers do, namely solving nonlinear ordinary differential equations (ODEs). Algorithms for ODEs exhibit only fine-grained parallelism and are poorly suited for vector computation. These characteristics account for the several similarities between the HEP and data flow architectures.

Initially, the HEP was a collection of heterogeneous pipelined functional units connected by a high-speed bus. It was implemented in Schottky TTL logic technology. The planned functional units included an algebraic module that implemented the basic 32-bit floating point operations, a 64-bit wide Runge-Kutta integrator unit, a first-in first-out buffer unit to help data scheduling among the other functional units, a small memory for holding constants, and analog and digital input and output units. A control processor called the *scheduler* explicitly moved data among the several functional units over the 32-bit high-speed bus. Synchronization between the scheduler and the functional units used *full-empty bits* provided with the bus-addressable input and output locations of each functional unit so that the scheduler could stall until both producer and consumer were ready. Later, it was observed that in-place atomic operations could be synchronized with full/empty bits by consuming the old value and then producing a new one.

A prototype of this system was built with funding from BRL, the US Army Ballistics Research Laboratory (now called the Army Research Laboratory) in Aberdeen, Maryland. A follow-on contract with them called for more robust technology, higher performance, and programmability in a high-level language (tacitly assumed to be Fortran). These requirements resulted in major architectural changes.

Process Execution Modules

To improve performance, multi-threaded algebraic modules (renamed Process Execution Modules or PEMs) were designed that could concurrently execute multiple instruction sequences independent of the

scheduler. They had 64-bit data paths throughout and were implemented entirely in 10K series ECL. A PEM had 64 hardware instruction streams (threads) to tolerate both functional unit and synchronization latency. An additional 64 threads were implemented for the operating system. User traps and system calls blocked the trapping thread and started a supervisor thread at the appropriate trap handler entry point. A PEM implemented eight protection domains, one of which was used by the operating system; the rest were made available to run independent user jobs concurrently.

2048 64-bit general-purpose registers, each equipped with a full/empty bit, and 4096 64-bit constant registers were provided in each PEM. Both general register and constant register addresses could be indexed using a per-thread register index or constant index, and were further translated by a pair of base addresses associated with the protection domain to which the thread belonged. There was a limit register as well for general register addresses. Instructions were 64 bits wide and were stored in a dedicated per-PEM program memory with base and limit registers for each protection domain. The integrator and first-in first-out units had been subsumed by the PEMs and disappeared, leaving only the scheduler and input-output functions to be dealt with.

To address the high-level language requirement, advice was sought from a compiler development company, Massachusetts Computer Associates. They recommended replacing the scheduler with a multi-ported shared data memory that all PEMs could access in parallel. Denelcor agreed with this assessment but insisted that all shared data memory locations would need to be equipped with full-empty bits. New PEM instructions would be required, as well as memory base and limit registers per protection domain. Given all the changes to the original HEP design, BRL decided to require a single PEM be built and demonstrated running a high-level language before the shared memory design could begin in earnest.

Register Fortran

The requirement to implement Fortran without data memory was a challenging one. An existing Fortran compiler was modified to use most of the 2048 general

registers of the PEM as shared variables collected into a special *register common* block. Full-empty synchronization was invoked by prefixing the variable name with a currency symbol (\$); this turned stores into produce operations and loads into consumes. Subscripted variable references used the register index and needed a four-instruction sequence for the address computation. Subroutines could be either called or created. The latter would create a hardware thread to run the subroutine, or trap if the number of hardware threads became too close to 64.

Excluding division, the maximum functional unit latency in a PEM was eight 100-ns cycles. Programs for this prototype one-PEM system that did very little synchronization or division exhibited no speed-up above eight hardware threads. Division was not pipelined, but a PEM could have from one to eight divide modules. Division took about 25 cycles. The floating point arithmetic format was compatible with the IBM 360. A few unusual features were provided such as unnormalized add and subtract operations and a loss-of-significance trap.

The PEM hardware was hosted by an Interdata 8/32 minicomputer, which had compatible arithmetic and was responsible for compilation as well as operating system functions. Several papers describe computational experiments with this system for ODEs [10], linear algebra[12], and graph shortest-path problems [5], and the system was demonstrated successfully to BRL in November 1979.

Shared Memory

The multi-threaded PEMs were already able to tolerate both functional unit and synchronization latency. Since functional unit latency was variable, unlike the peripheral processor “barrel” of the CDC 6600 and 7600 systems, the thread issue order was not round-robin but out-of-order, requiring only that the preceding operation of that hardware thread had been completed. Now that the time had come to design a multi-ported shared memory, the natural approach was to extend latency tolerance to the memory system by implementing a fully pipelined packet switched network connected to multiple pipelined memory banks rather than to use the circuit switches then popular in existing systems. Each three-port packet routing

node occupied two circuit boards and could handle a new memory request or response packet every 100 ns on each port. The clock period was 25 ns, and so a packet took four clocks to transit any point in the network.

To make the routers implementable, there was no buffering to accommodate conflicts for output ports; instead, each packet was immediately routed to some output port, which might be the wrong one if there was a routing conflict. This scheme, called “hot-potato” or “pure deflection” routing, was invented by Paul Baran in 1964 for the Internet. The HEP adapted the idea to computer interconnect. A misrouted packet had its priority increased to make it more likely to win future conflicts, and once a packet reached maximum priority, the routing tables were programmed to send these packets an Euler tour of the entire network to guarantee no further conflict losses. Gore-Tex twisted pair cables were used as in the Cray 1, permitting about 3 m of cable between routing nodes. The latency of a routing node and its associated cabling was 50 ns, which meant the graph of the network to be 2-colorable to guarantee correct timing.

The newly designed functional unit that connected the PEM to memory via the network was called the SFU, this abbreviation variously standing for Storage Function Unit or Scheduler Function Unit (since it indeed replaced the original scheduler). Informally, the SFU was fondly known as the Strange Function Unit, chiefly because it functioned in an unusual way. Instead of reporting synchronization failure, e.g., from an attempt to consume an empty memory location, back to the thread scheduling logic so the instruction could be reissued, it would repetitively retry the operation itself. Since a hardware thread would not be allowed to issue its next instruction until the SFU indicated the preceding one was complete, the semantic effect was the same. The major difference was that synchronization consumed a small amount of memory bandwidth and no instruction issue bandwidth at all. If there were enough hardware threads, typically 25–30, executing in parallel and tolerating all latencies, the HEP would steadily execute very close to one instruction per clock per PEM. This made each PEM the rough equivalent of a CDC 7600 or IBM 360/91 and about half the scalar speed of a Cray 1.

Software

The Fortran compiler was modified to address the new memory with little difficulty. The memory full/empty bits were invaluable, and had unexpected uses. Loops were executed in parallel with an arbitrary number of hardware threads by *self-scheduling* [14] the loop iterations: A thread seeking more iterations to execute would consume the iteration counter value, produce a new value incremented by a constant quantum, and then execute the iterations it had just acquired. The programmer wrote something like

```
I = $C  
$C = I + K
```

The quantum K would be chosen big enough to amortize scheduling overhead.

At the end of most loops, it was necessary to make sure all iterations were complete; this was accomplished by having each thread atomically decrement a counter and then wait at another location until the last thread, the one that decremented the counter to zero, freed all the rest. Harry F. Jordan named the underlying abstraction *barrier synchronization* [8] after the barrier used to start horse races because in both cases, all must enter before any may leave. Later, a pattern of pairwise produces and consumes resembling the wiring of an interconnection network was used to scale to larger numbers of threads. This was popularly called the *butterfly barrier*. Loops containing linear recurrences were parallelized in a similar way using parallel cyclic reduction [15] or one of the schemes due to Richard Ladner and Michael Fischer [11].

Harry Jordan developed a set of macros in M4 for automating much of the HEP Fortran programmer’s task including self-scheduling and barrier synchronization. This language extension was called The Force [9] and was the first Single Program, Multiple Data (SPMD) language. It was later implemented for other systems including Flex/32, Encore Multimax, Sequent Balance, Alliant FX series, Cray 2, Cray Y-MP, and Convex C220. IBM research adopted these ideas for the RP3 [4], and they achieved widespread adoption in OpenMP. Argonne National Laboratory wrote their own collection of M4 macros to implement a programming model based on monitors and message passing. This package was simply known as the “Argonne Macros” [2] and

evolved into the P4 and PARMACS languages for both C and Fortran.

The first implementation of the functional language Sisal was done on the HEP by Rodney R. Oldehoeft and his colleagues [1]. Sisal was also implemented on Cray vector processors and on the Manchester Data Flow Machine. The HEP let full/empty bits be used in a way in which loads did not consume but left the memory location full; this permitted a direct implementation of Sisal's single-assignment variables, known as *I-structures* [13] in the data flow community. That community in turn adopted the producer-consumer and atomic update semantics of full/empty bits in the concept of *M-structures* [13].

Parallel divide-and-conquer looked very simple in HEP Fortran but was often not simple at all; the system would run out of memory due to an excess of parallelism and its associated state. This problem was solved by judiciously limiting recursion depth. A much more elegant solution that needed no assist from the programmer was later devised and implemented by Robert H. Halstead in Multilisp [7].

Applications and Customers

There were six HEP machines delivered to customers. The first was a one-PEM system that went to Los Alamos National Laboratory, which was interested in using the system for their nonvector work. Monte Carlo particle simulation was one such application. The problem of parallel random number generation was first solved in using the HEP in addressing this kind of computation [6] and was quickly applied to vector processors as well. Argonne National Laboratory also acquired a one-PEM HEP system and used it for automated reasoning applications. Argonne also taught a summer school in parallel computing, and many students were exposed to HEP programming there.

A system was delivered to Messerschmitt-Bolkow-Blohm (MBB) outside Munich for simulating the flight dynamics of the Tornado fighter in real time. The original mission of the HEP, solving ordinary differential equations, was part of the job. The rest of it involved interfacing to physical hardware, in particular the avionics equipment for controlling the fighter's flight dynamics, the cockpit controls, and the vision system that provided the heads-up display and the view of the adversary aircraft, which in this case were Russian

MIGs. MBB started with one PEM but increased the size of the system to three PEMs during the mid-1980s.

BRL, the US Army Ballistics Research Laboratory that funded HEP development, acquired a four-PEM system around 1982. Their applications included not only physical simulations but also ray-tracing for rendering three-dimensional computer-aided design files. The application that resulted from this work was named BRL-CAD [3]. BRL was also a pioneer in using the Unix operating system for high-performance computing and strongly encouraged Denelcor to port Unix to the HEP. The resulting system, HEP/UPX, was delivered in early 1985. It was based on AT&T System 3 Unix, ran symmetrically on all PEMs, and was highly parallel internally. Another four-PEM system was delivered to the US Department of Defense, and a one-PEM system was delivered to Shoko Ltd. in Japan. Both of these systems were acquired for evaluation of the general potential of the HEP architecture for parallel computing.

Related Entries

- ▶ [CDC 6600](#)
- ▶ [Data Flow Computer Architecture](#)
- ▶ [Flynn's Taxonomy](#)
- ▶ [IBM System/360 Model 91](#)
- ▶ [MIMD \(Multiple Instruction, Multiple Data\) Machines](#)
- ▶ [Multi-Threaded Processors](#)
- ▶ [Networks, Direct](#)
- ▶ [Networks, Multistage](#)
- ▶ [Sisal](#)
- ▶ [SPMD Computational Model](#)
- ▶ [Synchronization](#)
- ▶ [Tera MTA](#)

Bibliography

1. Allan SJ, Oldehoeft RR (1985) HEP SISAL: parallel functional programming. In: Parallel MIMD computation: HEP supercomputer and its applications. Massachusetts Institute of Technology, Cambridge, MA, pp 123–150
2. Boyle J, Butler R, Disz T, Glickfeld B, Lusk E, Overbeek R, Patterson J, Stevens R (1987) Portable programs for parallel processors. Holt, Rinehart, and Winston, New York
3. BRL-CAD main website: [BRL-CAD | Open Source Solid Modeling](#)
4. Darema F, George DA, Norton VA, Pfister GF (1988) A single-program-multiple-data computational model for Epex/Fortran. *Parallel Comput* 5(7):11–24
5. Deo N, Pang CY, Lord RE (1980) Two parallel algorithms for shortest path problems. In: Proceedings of the 1980 international conference on parallel processing. New York

6. Frederickson P, Hiromoto R, Jordan TL, Smith B, Warnock T (1984) Pseudo-random trees in Monte Carlo. *Parallel Comput* 1(1):175–180
7. Halstead RH (Oct 1985) MULTILISP: a language for concurrent symbolic computation. *ACM Trans Program Lang Syst* 7(4): 501–538
8. Jordan HF (1983) Performance measurements on HEP: a pipelined MIMD computer. In: Proceedings of the 10th annual international symposium on computer architecture. Stockholm, Sweden, pp 207–212
9. Jordan HF (1986) Structuring parallel algorithms in an MIMD, shared memory environment. *Parallel Comput* 3(2): 93–110
10. Kumar SP, Lord RE (1985) Solving ordinary differential equations on the HEP computer. In: Kowalik JS (ed) Parallel MIMD computation: HEP supercomputer and its applications. Massachusetts Institute of Technology, Cambridge, MA, pp 231–273
11. Ladner RE, Fischer MJ (1980) Parallel prefix computation. *J ACM* 27(4):831–838
12. Lord RE, Kowalik JS, Kumar SP (1983) Solving linear algebraic equations on an MIMD computer. *J ACM* 30(1):103–117
13. Nikhil RS, Arvind (2001) Implicit parallel programming in pH. Morgan Kaufman, San Francisco
14. Smith BJ (1981) Architecture and applications of the HEP multiprocessor computer system. In: Proceedings of SPIE Real-Time Signal Processing IV. vol 298, New York, pp 241–248
15. Sweet RA (1988) A parallel and vector variant of the cyclic reduction algorithm. *SIAM J Sci Stat Comput* 9(4):761–765

Dense Linear System Solvers

BERNARD PHILIPPE¹, AHMED SAMEH²

¹Campus de Beaulieu, Rennes, France

²Purdue University, West Lafayette, IN, USA

Synonyms

Cholesky factorization; Direct schemes; Gaussian elimination; Linear equations solvers

Definition

A **dense system** is given by

$$Ax = f, \quad (1)$$

where $A \in \mathbb{R}^{n \times n}$ and $b \in \mathbb{R}^n$, with A having relatively very few zero elements.

Discussion

These are direct schemes based on block-Gaussian elimination with partial pivoting, for nonsymmetric

systems, and based on the block-Cholesky factorization for symmetric positive definite systems or symmetric indefinite systems. They take advantage of parallel architectures exclusively through efficient implementation of vector and matrix primitives on such computing platforms. In fact, most ScaLAPACK [1] procedures are generated by replacing the BLAS in the corresponding LAPACK procedures by their parallel counterparts PBLAS.

Nonsymmetric Systems

Consider first the nonsymmetric system $Ax = f$ where $A \in \mathbb{R}^{n \times n}$ is nonsingular. For the sake of illustration, let $n = 3k$ where k is the chosen block size. The first step in the procedure is to obtain the LU-factorization of the leading k columns with partial pivoting using the classical unblocked Gaussian elimination scheme, for example, see [6]. Thus if,

$$P_1 * A = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \quad (2)$$

in which P_1 is the permutation matrix affecting the partial pivoting in the LU-factorization of the first k columns of A , then

$$\begin{pmatrix} A_{11} \\ A_{21} \\ A_{31} \end{pmatrix} = \begin{pmatrix} L_{11} \\ L_{21} \\ L_{31} \end{pmatrix} * U_{11}, \quad (3)$$

where L_{11} and U_{11} are $k \times k$ unit lower triangular, and upper triangular matrices, respectively. The remainder of the first block row of the upper triangular factor of A is obtained by solving a triangular system with multiple right-hand sides,

$$L_{11}(U_{12}, U_{13}) = (A_{12}, A_{13}) \quad (4)$$

using the BLAS-3 primitive DTRSM. Setting

$$\begin{pmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{pmatrix} := \begin{pmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{pmatrix} - \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} (U_{12}, U_{13}), \quad (5)$$

which is obtained via the rank-k update BLAS-3 primitive DGEMM, the matrix $P_1 * A$ may be expressed as

$$2P_1 * A = \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & I & 0 \\ L_{31} & 0 & I \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & A_{22} & A_{23} \\ 0 & A_{32} & A_{33} \end{pmatrix}. \quad (6)$$

One is now ready to obtain the unblocked LU-factorization of the leading k columns of the lower principal submatrix of order k less, that is, of order $n - k$,

$$P_2 \begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} = \begin{pmatrix} L_{22} \\ L_{32} \end{pmatrix} * U_{22}. \quad (7)$$

Setting

$$\begin{pmatrix} A_{23} \\ A_{33} \end{pmatrix} := P_2 \begin{pmatrix} A_{23} \\ A_{33} \end{pmatrix} * U_{22} \text{ and} \quad (8)$$

$$\begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} := P_2 \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix}, \quad (9)$$

one can now obtain the remainder of the second block row of the upper triangular factor of A by solving the triangular systems

$$L_{22}U_{23} = A_{23}, \quad (10)$$

using the corresponding BLAS-3 primitive. Setting

$$A_{33} := A_{33} - L_{32}U_{23}, \quad (11)$$

which is also obtained via BLAS-3, one obtains the factorization

$$\begin{pmatrix} I_k & 0 \\ 0 & P_2 \end{pmatrix} * P_1 * A = \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & I \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & A_{33} \end{pmatrix}. \quad (12)$$

Now, for $n = 3k$, the last step consists of the unblocked LU-factorization

$$P_3A_{33} = L_{33}U_{33} \quad (13)$$

and setting

$$(L_{31}, L_{32}) := P_3(L_{31}, L_{32}) \quad (14)$$

resulting in the completed LU-factorization of A ,

$$\Pi * A = \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{pmatrix} \quad (15)$$

in which the permutation Π is given by

$$\Pi = \begin{pmatrix} I_{2k} & 0 \\ 0 & P_3 \end{pmatrix} \begin{pmatrix} I_k & 0 \\ 0 & P_2 \end{pmatrix} P_1. \quad (16)$$

This scheme is described in many references but the presentation above is closest to that in [4], where it is called the *right-looking algorithm*. It is the scheme implemented in the routines DGETRS of LAPACK and PDGETRS of ScaLAPACK [1].

The solution of $Ax = f$ is then obtained by block-forward and block-backward sweeps using the routines DGETRF of LAPACK and PDGETRF of ScaLAPACK.

Symmetric Positive-Definite Systems

For symmetric positive definite (s.p.d.) systems, one adopts the Cholesky factorization, $A = U^T U$, where U is upper triangular, through a block-oriented algorithm. For the sake of illustration, let A be partitioned as

$$A = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{12}^T & A_{22} & A_{23} \\ A_{13}^T & A_{23}^T & A_{33} \end{pmatrix}, \quad (17)$$

in which $A_{ij} \in \mathbb{R}^{k \times k}$ with k being the chosen block size.

Step 1:

Obtain the Cholesky factorization of the s.p.d. submatrix A_{11} using the unblocked Cholesky algorithm, using BLAS-1 and BLAS-2, for example, see [6],

$$A_{11} = U_{11}^T U_{11}, \quad (18)$$

where U_{11} is upper triangular of order k .

Step 2:

The remainder of the first block row of U is obtained by solving a triangular system with multiple right-hand

sides (Level-3 of BLAS),

$$U_{11}^T(U_{12}, U_{13}) = (A_{12}, A_{13}). \quad (19)$$

Step 3:

Setting

$$\begin{pmatrix} A_{22} & A_{23} \\ A_{23}^T & A_{33} \end{pmatrix} := \begin{pmatrix} A_{22} & A_{23} \\ A_{23}^T & A_{33} \end{pmatrix} - \begin{pmatrix} U_{12}^T \\ U_{13}^T \end{pmatrix} (U_{12}, U_{13}), \quad (20)$$

which is obtained using the BLAS-3 primitive DGEMM, one has the partial factorization

$$A = \begin{pmatrix} U_{11}^T & 0 & 0 \\ U_{12}^T & I & 0 \\ U_{13}^T & 0 & I \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & A_{22} & A_{23} \\ 0 & A_{23}^T & A_{33} \end{pmatrix}. \quad (21)$$

Similar to the LU-procedure for nonsymmetric systems, steps 1–3 are repeated twice to complete the block-Cholesky factorization $A = U^T U$ with U being of the form

$$U = \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{pmatrix}. \quad (22)$$

Again the solution of $Ax = f$ is obtained by block-forward and block-backward sweeps implemented in routines of LAPACK and ScaLAPACK based on Level-3 of BLAS.

Indefinite Symmetric Systems

Unlike the symmetric positive definite case (Cholesky factorization), one needs a pivoting strategy (Bunch–Parlett strategy [2]) for obtaining the factorization $U^T D U$, where U is unit upper triangular and D is block diagonal with $1 - by - 1$ and $2 - by - 2$ diagonal blocks. Aside of the pivoting scheme, and row and column pivoting, the main procedure steps are almost identical to the schemes discussed above.

Other Parallel Algorithms

Variants of the algorithm above include (a) the *left-looking* version of block Gaussian elimination in which

data is accessed only when absolutely needed, for example, for pivoting, and (b) the *Crout algorithm*. Characteristics of these versions are discussed by Dongarra, Duff, Sorensen, and van der Vorst in their survey [4].

The partial pivoting strategy guarantees a sound factorization but it limits the parallel efficiency due to loss of data locality. As a remedy, Gallivan, Plemmons, and Sameh discuss the *pairwise-pivoting* strategy in [5]. An error analysis of *pairwise pivoting* is given by Sorensen in [8]. While the upper bound on the error of the resulting factorization is worse than that of the classical LU-factorization with partial pivoting, extensive numerical experiments show that both factorization schemes yield almost identical solutions of the linear system.

The above block-oriented algorithms are quite robust and realize reasonably high performance on a variety of parallel architectures. In the early days of parallel computing, some work was done regarding the theoretical limit of parallelism inherent in Gaussian elimination. Csanky [3] made use of Leverrier's method [7] to show that a dense system of order n can be solved in $O(\log^2 n)$ parallel arithmetic operations employing $O(n^4)$ processors. Later, Preparata and Sarwate reduced the number of required processors to $O(n^{3.31}/\log^2 n)$ taking advantage of properties of *Newton identities* [7], and using *Strassen's* method for matrix multiplication [9].

Bibliography

- Blackford LS, Choi J, Cleary A, D'Azevedo E, Demmel J, Dhillon I, Don-garra J, Hammarling S, Henry G, Petitet A, Stanley K, Walker D, Whaley RC (1997) ScaLAPACK: A linear algebra library for message-passing computers. In: DOE scientific and technical information, #468499, available at <http://www.osti.gov/bridge/servlets/purl/468499-dBCNwX/webviewable/468499.pdf>, 1997
- Bunch JR, Parlett BN (1971) Direct methods for solving symmetric indefinite systems of linear equations. SIAM J Numer Anal 8:639–655
- Csanky L (1976) Fast parallel matrix inversion algorithms. SIAM J Comput 5:618–623
- Dongarra JJ, Duff IS, Sorensen DC, Van der Vorst H (1998) Numerical linear algebra for high performance computers. SIAM, Philadelphia, PA
- Gallivan KA, Plemmons RJ, Sameh AH (1990) Parallel algorithms for dense linear algebra computations. In: Gallivan KA, Heath MT, Ng E, Ortega JM, Peyton BW, Plemmons RJ, Romine CH, Sameh AH, Voigt RG (ed) Parallel algorithms computations. SIAM, Philadelphia, PA

6. Golub GH, Van Loan CF (1996) Matrix computations, 3rd edn. John Hopkins, Baltimore, MD
7. Householder A (1964) The theory of matrices in numerical analysis. New York, Dover
8. Sorensen DC (1985) Analysis of pairwise pivoting in Gaussian elimination. IEEE Trans Comput 34:274–278
9. Strassen V (1969) Gaussian elimination is not optimal. Numer Math 13:354–356

Dependence Abstractions

FRANÇOIS IRIGOIN
MINES ParisTech/CRI, Fontainebleau, France

Synonyms

Dependence accuracy; Dependence approximation; Dependence cone; Dependence direction vector; Dependence level; Dependence polyhedron; Exact dependence; Use-def chains

Definition

A data dependence between two computer instructions is due to some value passing or resource conflict. Dependencies usually cannot be known exactly at compile time. They must be modeled and over-approximated with different abstractions to be used statically by compilers and to decide if program transformations and optimizations, such as loop parallelization or locality improvement, can be applied without changing the program results. The characteristics of the dependence abstractions are linked to the program transformations whose legality must be proved. All dependence abstractions defined up to 2010 are families of convex affine sets, but different kinds of restrictions apply to the affine constraints. They are ordered from a less accurate to a most accurate abstraction, and they are all optimal with respect to at least one kind of transformation: the set of legal transformations cannot be increased by increasing the dependence abstraction accuracy.

Discussion

When two statements of a program use and/or define the same memory location, their execution order cannot be modified by the compiler to optimize the execution without modifying the program's result when the semantics of the operations is unknown. This sufficient constraint is known as Bernstein's condition [3] and is

used in all optimizing compilers as use-def chains, as defined by Aho et al. [1]. Let's consider the sequence:

```
S1: x = 2;
S2: y = 3*x+1;
```

The compiler cannot guarantee that the value of y is the same after executing $S1; S2$; or $S2; S1$. This is pretty obvious in a sequence, but $S1$ and $S2$ may be executed many times during the program execution, for instance because they are located in loops. So the compiler has to deal with all instances of Statements $S1$ and $S2$, in all possible execution traces.

Furthermore, the memory locations they access may be computed dynamically when pointers or arrays or function calls are used as in:

```
S1: x[i] = foo(x, y, 2);
S2: y[j] = 3*x[k]+1;
```

Without additional information, it is impossible to decide whether $S2$ could be executed before $S1$. It is unlikely, but the decision hinges on the ranges of Variables i , j , and k and on the behavior of Function foo (See ▶Banerjee's Dependence Test, ▶Dependences, ▶Dependence Analysis, ▶GCD Test, ▶Omega Test).

To sum up, an exact knowledge of statement dependencies depends on the program input and its resulting trace on one hand, and on our ability to specify statically the dynamic statement instances and memory locations accessed on the other. Since the exact dependence information is not computable in general, abstractions must be used to make the problem tractable in a compiler. These abstractions must be over-approximations to over-constrain the program optimizations and to guarantee their legality.

To start with, program inputs are ignored. Then a decision must be made about the representation of statement instances. Finally, a decision must be made about the representation of sets of arcs between statement instances. These decisions must be made according to the program transformations that the compiler implements.

Set Abstractions

Finite sets can be represented exactly in extension, but unbounded sets such as the number of iterations of a loop:

```
for(i=0; i<n; i++) { }
```

can only be represented implicitly via predicates. Different classes of predicates can be used, but Presburger arithmetic is the most powerful framework that is still computable. It subsumes less powerful frameworks such as convex polyhedra, i.e., affine constraints, affine subspaces, i.e., affine equations, and their sub-frameworks obtained by reducing the number of nonzero coefficients in the constraints, e.g., one or two, and/or the magnitude of the coefficients, e.g., $-1, 0$, or 1 . Bounded lists of abstract sets can be used to increase the accuracy of a given framework by providing a limited exact set union.

Sets of Statement Instances

Program parallelization has primarily been focused on loop parallelization because its optimization potential linked to the iteration counts is greater than task, instruction level, and superword level parallelizations.

The scope of interest is then limited to one loop or to a loop nest. Each instance of a statement in the loop body can be known exactly by its iteration vector, whose components are the values of the surrounding loop indices, from the outermost to the innermost one. Note that if some control structure is used within the loop body, some abstract instances may not exist in a real execution trace because some condition is true or false, but the set of abstract instances is greater than the set of effective instances, which is safe for most analyses and transformations. Note also that the nesting depths of two statements $S1$ and $S2$ may be different because all statements appearing within the loop nest do not always have the same nesting loops. The iteration (or instance) vectors i_1 for $S1$ and i_2 for $S2$ may not be comparable. A set abstraction must be chosen to define the sets of instances of each statement.

```

for(i=0; i<n; i++) {
S1:   a[i] = 0.
      for(j=0; j<m; j++) {
S2:     a[i] += b[i][j];
      }
}

```

In some cases, statement instances may be defined exactly over a small piece of source code because all iteration sets are defined exactly and because all conditions are predicates over the iteration vectors. The loop bounds and the test conditions must belong to the chosen set abstraction, e.g., affine constraints and convex sets. Such

pieces of code, which include sequences of loops, are called static control code (See ►Polyhedron Model):

```

for(i=0; i<n; i++) {
S1:   a[i] = 0.;
      for(j=0; j<m; j++) {
        if(i!=j)
          a[i] += b[i][j];
      }
}

```

D

Within a loop nest, two statements $S1$ and $S2$ may appear at different depths and/or may be nested within different loops. Their iteration vectors can be limited to their common nesting loops. Such an iteration vector does not always represent one instance of $S1$ or $S2$, but may represent a whole set of them:

```

for(i=0; i<n; i++) {
  for(j=0; j<m1; j++) {
    a[i][j] = 0.;
    for(j=0; j<m2; j++) {
      b[i][j];
    }
}

```

Here $S1$ and $S2$ have only one common loop, the i loop. Abstract instances of $S1$ or $S2$ represented by i contain each a set, possibly empty, of effective instances controlled by the j loops.

When all statements in a loop body appear at the same depth, the loop nest is said to be perfectly nested and statement instances are accurately represented by the loop-nest iteration vector, at least when the loop bounds are compatible with the set abstraction chosen (See ►Loop Nest Parallelization and ►Unimodular Transformations).

It is sometimes convenient to give a number to each statement and to integrate this number in its iteration vector. Thus, the set of all statement instances is abstracted by a set of vectors, which are easier to handle mathematically than several sets.

To sum up, some set abstraction must be chosen, for instance convex sets, as well as some coding rules to handle sequences, tests, and loop nesting. Control structures such as goto and exception preclude such instance abstractions and hence loop parallelization and optimization where they occur.

Sets of Dependence Arcs

Basically, Bernstein's conditions for Statements S_1 and S_2 must be checked for every instance pair of S_1 and S_2 . In the compiler, a collecting semantics is used to project the information about statement instances onto source statements. Information about arc instances is built on the abstract instance chosen, usually a vector, using a new arc predicate, i.e., yet another abstract set.

Each instance of S_1 are defined by an integer vector i_1 and a set predicate $P_1(i_1)$. Each instance of S_2 are defined by a vector i_2 and a set predicate $P_2(i_2)$. Dependence arcs between S_1 and S_2 are pairs (i_1, i_2) meeting Conditions $P_1(i_1)$ and $P_2(i_2)$ and some arc dependence condition $A(i_1, i_2)$, where A belongs to some set abstraction.

So the dependence abstraction is built with an instance abstraction and an arc abstraction. Predicates P_1 and P_2 are usually easy to derive from the program internal representation. The dependence predicate A requires some dependence test (See ▶Banerjee's Dependence Test, ▶GCD Test, ▶Omega Test).

Note that the arc set abstraction may be as simple as 0 and 1, empty set and non-empty set. The empty set is modeled by 0, i.e., no instance of S_2 ever depends on any instance of S_1 , and the non-empty sets by 1, i.e., there may exist at least one instance of S_2 that depends on at least one instance of S_1 . Again, a safe over-approximation is used. In case of doubt, a dependence arc is assumed. This safety property must be insured by the dependence test.

Note also that S_1 and S_2 may denote the same statement S , as some instances of S can depend on other instances of S .

Dependence Abstractions

Many dependence abstractions have been defined since the early seventies because there is a relationship between the program transformations applied by a compiler and the necessary accuracy of the dependence information.

In [16], Yang et al. survey the usual dependence abstractions:

- The dependence levels, DL
- The dependence direction vectors, DDV
- The dependence cone, DC

- The dependence polyhedron, DP
- The dependence set D , a.k.a. dependence distance when the dependences are constant
- The dependences between iterations, DI

They have been listed here from the less accurate to the most accurate one.

The dependence between iterations, DI , is the most accurate one and it stands out because the predicate associated is a predicate over two iteration vectors, i_1 and i_2 . It is the most flexible as it includes the modelization of the loops surrounding i_1 , of the loops surrounding i_2 , and of the relationship between i_1 and i_2 . It has been used by Feautrier [4] and Maydan et al. [10] (See ▶Polyhedron Model).

All other dependence abstractions assume that i_1 and i_2 are comparable vectors, i.e., they are related to the same surrounding loops. Since we need to know if i_1 occurred before or after i_2 , i_1 and i_2 can be replaced by the dependence vector d , with $d = i_2 - i_1$. Arcs (i_1, i_2) are replaced by dependence distances d .

D is the exact set of dependence distances. It can be used when its cardinal is bounded because the dependencies are constant. For instance, the loop nest:

```
for (i=0; i<n; i++)
S:   a[i] = a[i-1] + b[i];
```

contains dependences between instances of S and other instances of S . The i instance of S depends on the result of the $i - 1$ instance of S . This was used by Karp [8] for scheduling computations and introduced by Muraoka [11] for loop parallelization (See ▶Parallelization, Automatic). The systolic community also used this representation (See ▶Systolic Arrays).

The dependence polyhedron, DP , is derived from DI , when Predicates P_1 , P_2 , and A are affine, by adding the affine equation $d = i_2 - i_1$ and by projecting i_1 and i_2 .

The dependence cone DC , introduced by Irigoin and Triolet [7], is the transitive closure of DP . So it is a little bit less accurate, but easier to use because its dual representation is simplified. Vertices are transformed into rays, and only rays have to be dealt with.

The dependence direction vector abstraction, defined by Wolfe in [15], is also a convex set, but the acceptable constraints are signed constraints such as $c > 0$ (i.e., $c \geq 1$), $c \geq 0$, $c = 0$, $c < 0$ (i.e., $-c \geq 1$), and $c \leq 0$ (i.e., $-c \geq 0$), where c is any component of the dependence

vector d . As mentioned earlier, this is a special case of affine convex set: the constraints are limited to one variable, the coefficient is either 1 or -1 , and the constant term is either 0 or 1.

The dependence level abstraction used by Allen [2] is even simpler. The only constraints that can be used for the components of d are $c = 0$ and $c \geq 1$.

Other intermediate abstractions include the dependence vector abstraction of Sarkar et al. [13] which adds one-variable equations such as $c = n$, where n is a numerical constant to the dependence direction framework, and the abstraction by Wolf et al. [14] which also extends the dependence direction vector and Sarkar's abstraction by using constant terms different from 0 and 1. This leads to constraints bounding c such as $n_1 \leq c \leq 2$. With $n_1 = n_2$, Sarkar's constraints are obtained and hence the abstraction order.

Note that simpler less accurate abstractions can easily be derived from more complex more accurate ones with simple algebraic transformations. Note also that dependence tests must be developed for each abstraction and that their results depend on the subscript expressions used, on the pointer-based references, and on the procedure calls. Interprocedural dependence testing requires some more abstraction to modelize the effect of procedure calls on the local variables.

Note also that more complex set abstractions such as Presburger arithmetic, implemented by Pugh [12] in the Omega library, and Feautrier's QUAST [4], which uses finite lists to increase the accuracy of the underlying convex affine representation, are not covered here (see ►Polyhedron Model).

Relationships Between Dependence Abstractions and Program Transformations

Initial contributions, such as Karp's [8], Muraoka's [11], and Lamport's [9], were based on exact constant dependences, also called uniform dependences because they do not depend on the iterations. No abstraction of dependence was used, but the applicability is too limited because dependences are not always constant and dependence sets are not finite.

One may wonder why so many different dependence abstractions were introduced over the years. In fact, each of them is linked to a program transformation or to a class of program transformations. So, when a new

transformation is introduced, no dependence approximation might be available that is accurate enough for it and a new one has to be designed.

Yang et al. [16] shows that the dependence level, DL , is the best abstraction for loop reversal and loop parallelization, but dependence direction vectors are necessary to perform loop permutation (a.k.a. loop interchange) successfully. For unimodular loop transformations, for scalable loop tiling (see ►Tiling), and for multidimensional affine scheduling, the dependence cone is the best dependence abstraction. Some transformations require the dependence set D or the dependence polyhedron DP because they rely on the minimal size of a dependence arc. For instance, the odd and even iterations in the following loop can be executed concurrently:

```
for(i=0; i<n; i++) {
    a[i] = a[i-2]+b[i];
}
```

For transformations dealing with sequences of loops and global affine schedules as defined by Feautrier [5, 6], information about the dependence arc, DI , is necessary, although, in the important case of loop fusion, some sufficient dependence information D can be computed a posteriori since the statements $S1$ and $S2$ end up nested in the same loop by definition of the transformation.

Locality transformations, such as privatization or array scalarization, which may require information about the precision of a particular abstraction with respect to the exact information, are not covered here (see ►Parallelization, Automatic).

Future Directions

Dependence abstractions presented here are useful for loop nest transformations. Better information is needed to schedule larger piece of codes, such as sequences of loop nest: the information about the source and sink iterations must be preserved, when possible. This is part of the polyhedron model.

Conclusion

Several abstraction steps must be taken to make dependence analysis useful in an optimizing compiler. These abstraction steps are often overlooked when a particular program transformation and dependence abstraction are introduced, but a quick survey is useful to make

them understandable to newcomers to the automatic parallelization field.

However, when designing an optimizing compiler, it is important to understand the underlying affine base common to all abstractions and the impact of the choice of dependence abstractions on accuracy and computational requirements. The operator complexity is also often used for decision making, although the practical complexity is polynomial for most operations on unrestricted affine convex sets related to dependence analysis.

Related Entries

- [Banerjee's Dependence Test](#)
- [Dependences](#)
- [Loop Nest Parallelization](#)
- [Omega Test](#)
- [Parallelization, Automatic](#)
- [Polyhedron Model](#)
- [Systolic Arrays](#)
- [Tiling](#)
- [Unimodular Transformations](#)

Bibliographic Notes and Further Reading

The only paper dealing explicitly with the relationships between different dependence abstractions is Yang's [16]. Other papers deal only with one kind of dependence. The latest editions of the textbook by Aho et al. [1] presents several uses of dependences, e.g., scheduling, parallelization and memory allocation.

Bibliography

1. Aho V, Lam M, Sethi R, Ullman J (2007) Compilers. Principles, techniques & tools. Pearson Education, Addison-Wesley, Boston, MA
2. Allen R, Kennedy K (Oct 1987) Automatic translation of FORTRAN programs to vector form. ACM Trans Program Languages Syst (TOPLAS) 9(4):491–542
3. Bernstein AJ (Oct 1966) Analysis of programs for parallel processing. IEEE Trans Comput EC-15(5), 757–763
4. Feautrier P (1991) Dataflow analysis of scalar and array references. Int J Parallel Program 20(1):23–53
5. Feautrier P (Oct 1992) Some efficient solutions to the affine scheduling problem. Part I, one-dimensional time. Int J Parallel Program 21(5):313–347
6. Feautrier P (Oct 1992) Some efficient solutions to the affine scheduling problem. Part II, multi-dimensional time. Int J Parallel Program 21(6):389–420

7. Irigoin F, Triolet R (1987) Computing dependence direction vectors and dependence cones with linear systems, Technical Report CAI87E94, CRI, MINES ParisTech
8. Karp R, Miller R, Winograd S (1967) The organization of computations for uniform recurrence equations. J ACM 14(3):563–590
9. Lamport L (1974) The parallel execution of DO loops. Commun ACM 17(2):83–93
10. Maydan D, Amarasinghe S, Lam M (1993) Array dataflow analysis and its use in array privatization. POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on principles of programming languages. pp 2–15
11. Muraoka Y (1971) Parallelism exposure and exploitation in programs. PhD thesis, Report 424, Department of Computer Science, UIUC, Feb 1971
12. Pugh W (Aug 1992) A practical algorithm for exact array dependence analysis. Commun ACM 35(8):102–114
13. Sarkar V, Thekkath R (1992) A general framework for iteration-reordering loop transformations. ACM SIGPLAN 2010 conference on programming language design and implementation (PLDI), ACM SIGPLAN Notices, 27(7):175–187, San-Francisco
14. Wolf M, Lam M (1990) Maximizing parallelism via loop transformations. In 3rd Programming Languages and Compilers for Parallel Computing, Irvine, 1–3 Aug 1990
15. Wolfe M (1982) Optimizing supercompilers for supercomputers. PhD thesis, Department of Computer Science, UIUC
16. Yang Y-Q, Ancourt C, Irigoin F (1995) Minimal data dependence abstractions for loop transformations. Int J Parallel Program 23(4):359–388

Dependence Accuracy

- [Dependence Abstractions](#)

Dependence Analysis

- [Banerjee's Dependence Test](#)
- [Dependence Abstractions](#)
- [Dependences](#)
- [Omega Test](#)

Dependence Approximation

- [Dependence Abstractions](#)

Dependence Cone

- [Dependence Abstractions](#)

Dependence Direction Vector

► Dependence Abstractions

Dependence Level

► Dependence Abstractions

Dependence Polyhedron

► Dependence Abstractions

Dependences

PAUL FEAUTRIER

Ecole Normale Supérieure de Lyon, Lyon, France

Synonyms

Hazard (in hardware)

Definition

One needs a paradigm shift when reasoning about parallel programs. The usual approach, as for instance in denotational semantics, is to consider each statement as a function from memory state to memory state, and to consider two programs as equivalent if they implement the same function. For instance, $x = 0$; $x = x + 1$; and $x = 1$ are deemed equivalent. However, if these two programs are run in parallel with $x = 7$, they may give different results. Similarly, asking whether a statement can be run in parallel with itself does not make sense. However, the statement may be enclosed in a context – for instance, a loop – which causes it to be executed repeatedly. Each repetition is an instance or *operation*, and it makes sense to ask whether all or some of these operations can be executed in parallel.

These considerations motivate the following definitions. A program is a specification for a set of operations. Each operation is executed only once, and must have a unique name. The program must also specify the order in which operations are executed. It is easy to see that a sequential program has a total execution order,

and a parallel program has a partial execution order. In fact, an embarrassingly parallel program, in which everything can be done in parallel, has the empty execution order.

Methods for specifying operations and their execution order differ from one program style to another – loop programs, recursive programs, and functional programs all have different conventions.

Optimizing compilers, and in particular parallelizing compilers, try to transform the source program into an equivalent program that is better adapted to the target architecture, or runs faster, or has more parallelism. The problem is that in general, program equivalence is undecidable. A possible solution is to restrict the class of admissible transformations to a decidable subset. One of the classes of choice consists of *reordering* transformations. The operations of the transformed program are the same as in the source, but they may be executed in a different order (including parallel execution). The necessary information for validating such a transformation is the dependence relation:

Two operations u and v are independent if the programs $u; v$ and $v; u$ give the same results.

One can prove that a sufficient condition for the equivalence of two terminating programs which execute the same operations is that dependent operations are executed in the same order in both programs.

According to this definition, the dependence relation is symmetric. If one of the programs above is the reference program (e.g., if it is the sequential program to be parallelized), one usually orients the dependence relation according to the execution order of the reference program. With this convention, a transformed program is equivalent to the reference program if its execution order is an extension of the dependence relation.

Discussion

Bernstein's Conditions

Deciding if the execution order of two operations can be reversed can be arbitrarily difficult. Bernstein [3] has devised a simple test, which gives a sufficient condition for independence. This test depends only on the memory cells that are accessed by the operations to be tested.

Let $R(u)$ be the set of memory cells that are read by u . Similarly, let $W(u)$ be the set of memory cells which are written by u . Then u and v are independent if the three sets: $R(u) \cap W(v)$, $R(v) \cap W(u)$, and $W(u) \cap W(v)$ are all empty.

Note that in most languages, each operation writes at most one memory cell: $W(u)$ is a singleton. However, there are exceptions: multiple and parallel assignments, vector assignments among others.

Bernstein's conditions reflect a kind of worst-case reasoning. Consider:

$$y = e; \quad y = f;$$

The final value of y is given by the operation that executed last; in this case, this value is the value of f . If the operations are reversed, y will get the value of e , and *in the absence of any information* on the respective values of e and f , one must conclude that the two operations are dependent.

If the dependence relation is computed according to Bernstein's conditions, the above property can be extended in two directions. Firstly, it applies now also to nonterminating programs: one can show that the *history* of each variable (the succession of values it holds during execution) is the same for both programs. Secondly, it also holds if one of the programs has real parallelism. The reason is that two operations that write to the same location are always in dependence (this is the third Bernstein condition) and hence can never be executed in parallel. Hence, two writes to the same memory cell never occur at the same time.

Scalars, Arrays, and Beyond

Computing dependences for scalar variables is easy. The sets R and W above are finite, and computing intersections is trivial. The only real difficulty is caused by *aliasing*, when two distinct identifiers refer to the same memory cell. The detection of aliasing is a difficult problem. However, it is easy to detect cases in which there is no aliasing, for example, among the local variables of a procedure, and to handle all other cases conservatively. Orienting each dependence is easy, at least within linear code (*basic blocks*).

The case of arrays is more problematic. Arrays usually occur in loops, and their subscripts usually differ from one iteration to the next in complex ways. This has lead to the invention of the *polyhedral model* [6, 9], in

which one considers only regular DO loops, and subscripts which are affine functions of the surrounding loop counters and of constant parameters. A function is affine if it is the sum of a linear part and a constant. One may construct a dependence relation among operations (i.e., loop iterations) in the following way. To name an iteration, one may use its *iteration vector*, whose coordinates are the surrounding loop counters, ordered from outside inward. Iterations are ordered according to the lexicographic order of iteration vectors:

$$\vec{i} < \vec{j} \equiv i_1 < j_1 \vee (i_1 = j_1 \wedge i_2 < j_2) \vee \dots \quad (1)$$

where $<$ (read “before”) is the execution order.

Consider two iterations \vec{i} and \vec{j} of some loop nest, and two accesses to the same array, $A[f(\vec{i})]$ and $A[g(\vec{j})]$, one of them at least being a write. f and g are subscript functions, which relate subscripts to the surrounding iteration vectors. To be in dependence, the two iterations must satisfy the following conditions:

- They must access the same array cell: $f(\vec{i}) = g(\vec{j})$.
- They must be legal iterations, that is, each loop counter must be within the corresponding loop bounds, which define the *iteration domains* of the operations.
- The direction of the dependence is given by the execution order $\vec{i} < \vec{j}$.

For programs that fit in the polyhedral model, the subscript equations and the iteration domain constraints are conjunctions of affine equalities or inequalities. According to (1), the ordering constraint can be split in several conjunctions. Hence, the set of iterations in dependence is the union of several disjoint *dependence polyhedra*. Each polyhedron is characterized by the number of equations at the begining of the execution order, the so-called dependence depth, which goes from 0 to the number of loops that enclose both array references. For some authors, the dependence depth (or level) starts at 1 and goes up to infinity. The notation $\vec{i} \delta_p \vec{j}$ is commonly used to state that there is a dependence from iteration \vec{i} to \vec{j} at depth p , that is, the first p coordinates of \vec{i} and \vec{j} are equal.

For more general programs, one has to resort to approximations: a constraint is omitted or approximated if not affine. This has the effect of enlarging the dependence polyhedra, and hence reducing the amount

of parallelism. However, the correctness of the generated program is still guaranteed.

When a program uses pointers, the computation of dependences becomes much more difficult. The reason is that, depending on the source language, a pointer can point anywhere in memory, and that it is very difficult to decide whether two pointers point to the same memory cell. The best that can be done is to associate to each pointer (conservatively) a region in memory, and to decide a dependence when two regions intersect.

Classification

While in Bernstein's conditions the two operations play the same role, the symmetry is broken as soon as the direction of the dependence is taken into account. One usually distinguishes:

- Flow dependences (or true dependences, or Producer/Consumer dependences [PC], or Read after Write hazards [RAW]), in which the write to a memory cell occurs before the read.
- Output dependences (or PP dependences, or WAW hazards) in which both accesses are writes.
- Anti-dependences (or CP dependences, or WAR hazards) in which the read occurs before the write.
- In some contexts (e.g., when discussing program locality), one may consider Consumer/Consumer dependences, which do not constrain parallelization.

If the objective is just to decide which operations can be executed in parallel, all three kinds are equivalent. Differences appear as soon as one considers modifying the source code for increased parallelism. It is easy to see that in a flow dependence, a value which is computed by the first operation is reused later by the second operation. Hence, a flow dependence corresponds to the naming of an intermediate result, is intrinsic to the code algorithm, and cannot be removed except by modifying it. In contrast, an output dependence simply indicates that a memory cell is reused when the value it holds becomes useless. Such a dependence can be removed simply by using two distinct cells for the two values. Lastly, in a correct program, where all memory cells are defined before being used, removing output dependences has the side effect of removing anti-dependences.

It can even be shown that some of the flow dependences are spurious. Let us consider a memory cell and an operation v that reads it. There may be many writes to this cell. Intuitively, the only one on which v really depends is the last one u which is executed before v . The dependence from u to v is a *direct dependence*. If the source program is sufficiently regular, direct dependences can be computed using linear programming tools [5]. It is, however, difficult to find conservative approximations for more general programs.

Dependence Tests

Early automatic parallelizers were concerned only with the existence or nonexistence of dependences. For instance, to decide that a loop is parallel, one has only to show that there are no dependences between its iterations, that is, all related dependence polyhedra are empty.

Since dependences occur between a pair of statements, it follows that the number of dependences increases more or less as the square of the size of the program. Hence, the scientists of the 1970s initiated a quest for fast but approximate *dependence tests*. One may, for instance, observe that some of the dependence constraints are linear equations whose unknowns, the loop counters, are integers. Such an equation can have solutions only if the gcd of the unknown's coefficients divides the constant term. This observation is the basis of the very fast *gcd test*.

The Banerjee test [2] is based on the observation that in many cases, when solving an equation $f(x) = 0$, one knows a lower bound a and an upper bound b for x , which come, most of the time, from loop bounds. Now, the equation has solutions only if $f(a)$ and $f(b)$ are of opposite signs. The approximation comes from the observation that the eventual solution is not necessarily integral, hence the idea of coupling the gcd and Banerjee test, for increased precision. This idea can be extended in the following way: the inequalities of the problem dictate that x belongs to some polyhedron, and the condition for the existence of a solution is that the maximum of f over this polyhedron be positive, and the minimum be negative. Now the extrema of an affine function over a polyhedron are located at some of its vertices. Hence, one has only to test the sign of f at a finite set of points. This is especially efficient if the vertices are in evidence.

Many other tests were invented with the aim of handling systems of equations instead of a single equation. For instance, in the Lambda test [10] the Banerjee test is applied to well chosen linear combinations of equations. If the answer is negative, then the original system has no solution.

However, it was realized around 1990 that the question of the emptiness of the dependence polyhedron can be solved using classical linear programming algorithms. This approach was originally deemed too costly, but with improved algorithms and a 1,000-fold increase in processing power (Moore's law), the argument has lost its strength.

One possibility is to use the Fourier–Motzkin algorithm [12], in which the unknowns of the problem are eliminated by combining the inequalities that define the dependence polyhedron, until one obtains numerical inequalities, which can be decided by inspection. Programming the Fourier–Motzkin algorithm is quite simple, but its complexity is doubly exponential, which is not critical since dependence testing problems are usually small. The standard Fourier–Motzkin algorithm finds rational solutions or proves that none exists. An extension, the Omega test [11], considers only integral solutions. Other extensions are the I test [8] and the Power test [13].

Another possibility is to use the Simplex algorithm, which is more complex, but which runs most often in time proportional to the third power of the number of inequalities, and hence scales better for large problems. The algorithm can also be extended to the integer case using Gomory cuts, and can even solve parametric problems [4].

Dependence Approximations

Early parallelization algorithms (e.g., deciding if a loop has parallelism) did not need a precise knowledge of dependence polyhedra. Just testing them for emptiness was enough. As the sophistication of parallelization algorithms increased, more precise descriptions of dependences were needed. All such approximations can be described as supersets of dependence polyhedra. On the relations between dependence approximations and program transformations, see [14].

The simplest approximation consists in ignoring the fact that iteration vectors have integer components. When testing a transformed program for legality, this

may generate false negatives. However, this occurs sufficiently seldom that it is usually ignored.

Dependence Depth

The dependence depth abstraction is a natural byproduct of the decomposition of the lexical order into disjoint polyhedra as in (1). One simply has to record at which depth a dependence occurs, instead of taking the disjunction of the test results for several depths. Knowledge of the dependence depth allows one to decide, in a loop nest, which loop must be kept sequential and which one can be executed in parallel. This information is necessary for the Allen and Kennedy algorithm, which uses loop splitting to find more parallel loops.

Dependence Distances

The dependence distance is the difference between the iteration vectors of two dependent operations. One can define a *distance polyhedron* as:

$$D = \{\vec{d} \mid \exists \vec{i}, \vec{j} : \vec{i} \delta_p \vec{j}, \vec{d} = \vec{j} - \vec{i}\}.$$

A projection algorithm is needed to eliminate the existential quantifiers: one may use the Omega test or parametric integer programming. Observe also that the dependence distance is always lexicopositive.

Computing the distance polyhedron is especially interesting when it reduces to a single constant vector. One says in that case that the dependence is uniform. Many parallelization algorithms are especially simple when the source program has only uniform dependences.

Dependence Direction Vectors

In the presence of nonuniform dependences, one may further abstract the distance polyhedron by considering only the signs of the components of the distance vectors. The components of a Dependence Direction Vector (DDV) are either integers (meaning that the component is constant) or one of the symbols $<$, $>$, \geq , \leq (meaning that the component has the corresponding sign) or $*$, meaning that the component may have an arbitrary sign. DDVs are usually computed by adding the corresponding constraint to the definition of the dependence polyhedron and testing for feasibility. While simpler to compute than the distance polyhedron, DDVs give enough information to check the legality of some transformations, like loop permutation.

Dependence Cones

The dependence cone is simply the cone generated by the dependence distance vectors. The simplest way of computing the dependence cone is to compute first the vertices of the distance polyhedron, d_1, \dots, d_n . The dependence cone is then:

$$C = \left\{ \sum_{k=1}^n \lambda_k d_k \mid \lambda_k \geq 0 \right\}.$$

Knowledge of the dependence cone is especially useful when tiling a loop nest.

Related Entries

- [Banerjee's Dependence Test](#)
- [Bernstein's Conditions](#)
- [Dependence Abstractions](#)
- [Omega Test](#)
- [Polyhedra scanning](#)

Bibliographic Notes and Further Readings

There is a large literature on pointer analysis, which may be useful for program parallelization, albeit its orientation is more toward program verification and safety. A good starting point is [7] (110 references!).

For an attempt at parallelization of recursive programs, see [1].

Bibliography

1. Amiranoff P, Cohen A, Feautrier P (2006) Beyond iteration vectors: instancewise relational abstract domains. In: Static Analysis Symposium (SAS'06), Seoul, Corea
2. Banerjee U (1997) Loop transformations for restructuring compilers: Dependence analysis. Kluwer Academic Publishers
3. Bernstein AJ (1966) Analysis of programs for parallel processing. IEEE Transactions on Electronic Computers, EC-15:757–762
4. Feautrier P (1988) Parametric integer programming. RAIRO Recherche Opérationnelle 22:243–268
5. Feautrier P (1991) Dataflow analysis of scalar and array references. Int J Parallel Program 20(1):23–53
6. Feautrier P (1996) Automatic parallelization in the polytope model. In: Perrin G-R, Darte A (eds) The data-parallel programming model. LNCS vol 1132, pp 79–103, Springer, Heidelberg
7. Hind M (2001) Pointer analysis: haven't we solved this problem yet? In: PASTE'01. ACM, Snowbird
8. Kong X, Klappholz D, Psarris K (July 1991) An efficient data dependence analysis for parallelizing compilers. IEEE Trans Parallel Distrib Syst 1(3):342–359

9. Lengauer C (1993) Loop parallelization in the polytope model. In: Best E (ed) CONCUR'93, LNCS vol 715, pp 398–416, Springer, Berlin
10. Li Z, Yew P-C, Zhu C-Q (1990) An efficient data dependence analysis for parallelizing compilers. IEEE Trans Parallel Distrib Syst 1:26–34
11. Pugh W (1991) The omega test: a fast and practical integer programming algorithm for dependence analysis. In: Supercomputing, 1991
12. Triolet R, Irigoin F, Feautrier P (1986) Automatic parallelization of FORTRAN programs in the presence of procedure calls. In: Robinet B, Wilhelm R (eds) ESOP 1986, LNCS 213, Springer, Berlin
13. Wolfe M, Tseng C-W (1992) The power test for data dependence. IEEE Trans Parallel Distrib Syst 3(5):591–601
14. Yang Y-Q, Ancourt C, Irigoin F (1995) Minimal data dependence abstractions for loop transformations. In: LCPC'94: Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing, Springer, London, pp 201–216

Detection of DOALL Loops

- [Parallelism Detection in Nested Loops, Optimal](#)

Determinacy

CHRISTOPH VON PRAUN
Georg-Simon-Ohm University of Applied Sciences,
Nuremberg, Germany

Synonyms

[Determinism](#)

Definition

Determinacy refers to the property of repeatability. A parallel program is determinate, if all feasible executions of the program, when started on the same input, generate the same result.

Likewise, the result of a non-determinate parallel program can vary for different executions, depending on the timing of different threads. The reason for non-determinacy are race conditions.

Discussion

Introduction

This entry discusses the concept of determinacy in the context of parallel programming. The examples and classification of different kinds of determinacy are adopted from Emrath and Padua [6, Sect. 2]. The discussion focuses on explicitly parallel programs with threads that share memory, although the concepts of determinacy can be generalized to other parallel programming models.

Sequential programs are naturally determinate (Exceptions exist, namely languages that offer a choice operator, where the choice of program-flow is not directly specified by the programmer but chosen by the run-time system; such methods are, for example, embodied in backtracking systems.) If parallel programming is used to accelerate the performance of a sequential program without intention of changing the functional behavior, determinacy is a correctness requirement. In general, however, determinacy is neither a sufficient nor a necessary condition for the correctness of a parallel program.

The determinacy of a program can depend on the program input. An example is given in Fig. 1.

External Determinacy

Determinacy that refers to the repeatability of input-output behavior is called *external determinacy*.

```
int i = read()
if (i != 0)
    cobegin
        i = 1 ||
        i = 2
    coend
endif
print(i)
```

Determinacy. Fig. 1 Determinacy of this program is input-dependent: On input 0, the program will always output 0; for other inputs, the program is non-determinate and may output 1 or 2. The reason for the non-determinacy is the data race among the write accesses to variable *i* in the parallel block

Internal Determinacy

A parallel program is called *internally determinate* if the sequence of intermediate values taken by each shared location is the same in all executions. Figure 2 gives an example.

The notion of *internal determinacy* is stronger than external determinacy, since not only final values but also intermediary values of shared variables are considered. Naturally, programs that are internally determinate are also *externally determinate*.

Internal determinacy does not imply that the overall shared state of the program is transformed in a predetermined sequence. It may well be that some executions of the example program in Fig. 2 initialize variables at lower indices first, while other executions initialize variables in some other order.

Internal determinacy is however not required to achieve *external determinacy*. Figure 3 gives an example.

The internal non-determinacy in Fig. 3 is due to the race conditions among concurrent iterations of the second loop when accessing the critical section. The order in which different iterations enter the critical section is

```
int[] arr = new arr[N]
doall (i <- 0 to N-1)
    arr[i] = i + 1
enddoall
```

Determinacy. Fig. 2 Internal determinacy: The sequence of states for each shared variable *arr[i]* is the same in all executions, that is, irrespective of the timing of parallel iterations. Every execution transits the state of location *arr[i]* from its initial value to value *i+1*

```
int[] arr = new arr[N]
int sum = 0
doall (i <- 0 to N-1)
    arr[i] = i + 1
enddoall
doall (i <- 0 to N-1)
    critical
        sum += arr[i]
    endcritical
enddoall
```

Determinacy. Fig. 3 External determinacy but internal non-determinacy: The sequence of intermediate values of variable *sum* can differ in different executions. However, the final value of all shared variables is always the same

not predetermined. There is no data race however, since simultaneous updates of variable *sum* cannot occur.

External determinacy arises here, since the integer addition performed on the shared integer variable *sum* is commutative and associative.

Benefits: Internal non-determinacy can be exploited to accelerate parallel collective reduction and scan operations [10].

Debugging: Internal non-determinacy complicates debugging since a programmer may observe different sequences of operations and intermediate values of shared variables in different program executions.

Associative Non-Determinacy

A computer represents real numbers typically as floating point values with limited precision. Unlike their mathematical counterpart, operations such as addition may not be generally associative and distributive on the floating point representations.

The program in Fig. 4 seems similar to the programs in Fig. 3: It is internally non-determinate, since the *sum* is computed in some order that is not specified a priori by the program. This program is however not *externally determinate*, since the add operation is computed on the floating point, not the integer domain. The result of this computation may vary slightly across different runs. For some applications, this deviation and thus the external non-determinacy may be acceptable, hence the program could be considered correct.

External non-determinacy that is solely due to the lack of associativity and commutativity of floating point operations is called *associative non-determinacy*.

```
int N = 100000
double sum = 0
doall (i <- 0 to N)
    critical
        sum += 1.0/i
    endcritical
enddoall
```

Determinacy. Fig. 4 Associative non-determinacy: The final value of variable *sum* can vary slightly in different executions due to the non-associativity of floating point numbers

Complete Non-Determinacy

External non-determinacy may occur for reasons other than the special case of associative non-determinacy discussed in the previous paragraph. Programs with such form of external non-determinacy are called *completely non-determinate*.

When executed on a given input, the final state of a completely non-determinate program depends on the timing of threads and their operations on shared variables. An example for such a program is given in Fig. 5.

The non-determinacy is due to a race condition among the write operation to variable *j* in different loop iterations. In this particular case, the race condition is a *data race*, since there could be executions with simultaneous write accesses. But the kind of race is immaterial here. If even the execution of write operations were ordered due to a critical section, the precise order is unspecified in the program and thus the program is *completely non-determinate*.

Analysis of Non-Determinacy

Non-determinacy is caused by race conditions. Hence, tools for race detection are used to analyze cases of non-determinacy. When tool are specialized on the detection of *data races*, they will not be successful to identify causes of non-determinacy since general races that are not data races can also cause non-determinacy.

Determinacy in Other Parallel Programming Models

Non-determinacy may be introduced in any explicitly parallel programming model that is susceptible to race conditions. Besides multithreading with shared memory, these are, for example, task-based programming models such as Intel TBB [4] and also message-passing models such as MPI [7].

```
int N = 100000
int j = 0
doall (i <- 0 to N)
    j = i
enddoall
print(j)
```

Determinacy. Fig. 5 Program that is completely non-determinate

Determinate Parallel Programs

Parallel programs that are completely non-determinate are rarely useful in practice. Hence, programming model have been designed that prevent or at least reduce the risk of creating a parallel programs with undesired non-determinacy.

Autoparallelization: Starting point is a sequential program from which a parallel form is derived automatically. The resulting parallel programs are typically externally determinate.

Autoparallelization may involve the reordering of arithmetic operations. The associativity of arithmetic operations is defined in the language report; in particular, floating point operations may be non-associative. The user of a parallelization tool can typically configure if the parallelization should comply with the non-associativity for floating point operations or if such operations may be reordered. The latter may lead to programs with associative non-determinacy.

Speculative parallelization: Similar to autoparallelization, speculative parallelization avoids non-determinacy by design, however not at compile time, but at run time. A run-time system with support for speculative parallelization detects harmful race condition, and hence sources of non-determinacy [11]. When a possible violation of determinacy is detected, the results of the speculative parallel execution are discarded, followed by a re-execution. An example for a determinate programming model based on speculative parallelization is given in [14].

Programming languages for asynchronous parallel computations: Some parallel programming models are designed to limit the space of permissible programs to deterministic ones. Examples are Kahn Networks [8] and an experimental programming language described by Steele [13]. The key idea of the latter language is to limit the available forms of asynchrony and at the same time to restrict the side effects that each asynchronous computation can have on shared memory. Different models for the validation of side effects are feasible, ranging from purely dynamic to static analysis. Unintended non-determinacy due to race conditions is one of the most frequent errors in parallel programs. Thus, a language-enforced programming discipline to avoid non-determinacy seems to be useful. However, such determinate parallel programming models have not

yet been adopted widely. Possible reasons are that the models restrict the solution space too much, such that common algorithmic or parallel programming idioms cannot be expressed. Moreover, the performance impact due to run-time checking for forbidden side effects is deemed too high or not well understood.

Determinate Parallel Execution

Deterministic replay: *Deterministic replay* is a debugging aid and controlled run-time environment that enables repeating a particular parallel program execution of an otherwise non-determinate parallel program. A debugging methodology for data races using deterministic replay was first conceived and formalized by Choi and Min [3, Sect. 2.4]. They prove that an iterative debugging process of data race detection and replay will result eventually in a program execution that is data-race-free. The process requires a data race detection tool with the following property: If an execution has data races, the tool reports some but not necessarily all races.

Systems that support *deterministic replay*, operate in two phases: First, the *recording phase*, where program input and the synchronization order of a parallel program execution are memorized. Second, the *replay phase*, where recorded information is used by a controlled run-time environment to reproduce the original program execution.

The original and the replayed executions typically correspond to each other at least according to the requirements of internal determinacy, that is, the sequence of values taken by individual variables must be the same in both executions. Such guarantee is typically given by software-based implementations, for example [2]. Some systems are able to maintain a global order of all updates, which is an even stronger notion of correspondence among executions [9].

Computer architectures for determinate parallelism: Recently, computer architectures [5] have been proposed that constrain the possible execution schedules at the level of individual memory operations, or groups of memory operations called chunks. Constraints are such that inter-processor communication becomes determinate. At a programming level, this means, for example, that the order in which a lock is taken by different processors is the same in different executions. Since this architecture permits only one execution timing, it lets

any parallel program, determinate or not, execute in a determinate manner.

Checking Determinacy

Determinacy can be asserted at run time, by validating that an execution is free from race conditions. The property of race freedom is however too strong to be useful, since many correct parallel programs with internal non-determinacy would be flagged as erroneous. Hence, checking determinacy means checking external determinacy while allowing internal non-determinacy.

Burnim and Sen [1] developed a set of program annotations, called *bridge assertions*, to mark blocks of code that should behave externally determinate. These assertions validate external determinacy by relating the functional behavior of such block with observations from previous executions of the same block. The method is capable to describe associative non-determinacy.

An alternative technique is proposed by Sadowsky et al. [12]. Their run-time checker verifies the absence of interference while allowing race conditions due to certain synchronization idioms that are known to preserve determinacy. As in [1], code blocks for which (external) determinacy should be validated have to be specified explicitly.

Related Entries

- [Deterministic Parallel Java](#)
- [Parallelization, Automatic](#)
- [Race Conditions](#)
- [Race Detection Techniques](#)

Bibliography

1. Burnim J, Sen K (2009) Asserting and checking determinism for multithreaded programs. ESEC/FSE '09: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering on European Software Engineering Conference and Foundations of Software Engineering Symposium, ACM, New York, pp 3–12
2. Choi J-D, Alpern B, Ngo T, Sridharan M, Vlissides J (2001) A perturbation-free replay platform for cross-optimized multithreaded applications. IPDPS'01: Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01), IEEE Computer Society, Washington, DC
3. Choi J-D, Min SL (1991) Race frontier: reproducing data races in parallel-program debugging. PPoPP'91: Proceedings of the

- Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, New York, pp 145–154
4. Intel Corporation. Intel Threading Building Blocks. <http://www.threadingbuildingblocks.org/>
5. Devietti J, Lucia B, Ceze L, Oskin M (2009) DMP: deterministic shared memory multiprocessing. ASPLOS 09: Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, New York, pp 85–96
6. Emrath PA, Padua DA (1989) Automatic detection of nondeterminacy in parallel programs. Proceedings of the ACM Workshop on Parallel and Distributed Debugging, University of Wisconsin, Madison, Wisconsin, pp 89–99
7. Message Passing Interface Forum. MPI: A message passing interface standard. <http://www.mpi-forum.org/>, June 1995
8. Kahn G (1974) The semantics of a simple language for parallel programming. Information Processing 74: Proceedings of the IFIP Congress, Stockholm, Sweden, North-Holland, pp 471–475
9. Montesinos P, Ceze L, Torrellas J (2008) DeLorean: recording and deterministically replaying shared-memory multiprocessor execution efficiently. ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture, IEEE Computer Society, Washington, DC, pp 289–300
10. Owens J (2007) Data-parallel algorithms and data structures. SIGGRAPH '07: ACM SIGGRAPH 2007 courses, ACM, New York, p 3
11. Rauchwerger L, Padua DA (1999) The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. IEEE Trans Parallel Distrib Syst 10(2):160–180
12. Sadowski C, Freund SN, Flanagan C (2009) Singletrack: a dynamic determinism checker for multithreaded programs. ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems, Springer, Berlin/Heidelberg, pp 394–409
13. Steele GL Jr (1990) Making asynchronous parallelism safe for the world. POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, New York, pp 218–231
14. von Praun C, Ceze L, Cașcaval C (2007) Implicit parallelism with ordered transactions. PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, New York, pp 79–89

Determinacy Race

- [Race Conditions](#)

Determinism

- [Determinacy](#)

Deterministic Parallel Java

ROBERT L. BOCCINO, JR.

Carnegie Mellon University, Pittsburgh, PA, USA

Synonyms

DPJ

Definition

Deterministic Parallel Java (DPJ) is a parallel programming language based on Java and developed at the University of Illinois at Urbana-Champaign. DPJ uses a type and effect system to provide deterministic-by-default semantics. “Deterministic” means that for a particular input, a given program always produces the same output, regardless of the parallel schedule chosen. “By default” means that the program is deterministic unless the programmer explicitly requests nondeterministic behavior. DPJ guarantees this semantics at compile time for the programs it can express.

Discussion

Introduction

Many parallel programs are intended to have deterministic behavior. Typically, these programs are *transformative*: they accept some input, do a computation in memory, and produce some output, in contrast to programs such as servers that accept input throughout the computation. There are many examples of deterministic programs in domains such as scientific computing, graphics, multimedia, and artificial intelligence.

A programming model that can guarantee deterministic behavior for such programs has many advantages. Parallel programs become easier to reason about, because a key source of complexity (nondeterministic parallel schedules) has been removed, along with attendant difficulties such as unwanted data races and complex memory models. Testing and debugging are easier because, as in the sequential case, it suffices to test only one schedule per input.

Well-known programming models that guarantee determinism include dataflow, data parallel, and pure functional programming. While these models have the advantages described above, they are restrictive

in that they disallow sharing of mutable data through references. In contrast, widely used imperative languages such as Java that support references to mutable data typically provide no guarantee that a parallel program is deterministic or even race-free.

Deterministic Parallel Java (DPJ) is a Java-based programming language that provides deterministic-by-default semantics for parallel programs, while supporting references to mutable objects. The key feature of DPJ that allows this guarantee is a *type and effect system* that expresses what parts of memory are read and written by different parts of the program, so that parallel tasks can be checked for noninterference. DPJ has first-class support for a fork-join style of parallelism. Specifically, it includes a `foreach` statement for parallel loops and a `cobegin` block for parallel statements, most often used to express parallel recursion. Object-oriented frameworks, discussed below, provide a way to express other forms of parallel control.

The DPJ Effect System

The DPJ effect system [5] allows the programmer to partition the heap into *regions*, and to provide *method effect summaries* stating the effects of methods in terms of accesses (reads and writes) to those regions. The compiler uses simple, intraprocedural analysis to check two things:

1. *Correctness of method summaries.* Every method effect summary includes all the actual effects of the method it summarizes, as well as the effects of any methods overriding that method.
2. *Noninterference.* Any two memory accesses, one from each of two tasks that may run in parallel, do not conflict (i.e., they are both reads, or they operate on disjoint memory).

The programmer can omit the region and effect annotations for sequential code (an ordinary sequential Java program is a correct DPJ program) but must add the annotations to code that is executed in a parallel task.

Regions and Effects

[Figure 1](#) gives a simple example showing the use of regions and effects in DPJ. Line 2 declares region names `First` and `Second`. Lines 3 and 4 place instance fields `first` and `second` in regions `First` and `Second`. The

```

class Pair {
    region First, Second;
    int first in First;
    int second in Second;
    void setFirst(int first) writes First {
        this.first = first;
    }
    void setSecond(int second) writes Second {
        this.second = second;
    }
    void setBoth(int first, int second) {
        cobegin {
            setFirst(first); /* writes First */
            setSecond(second); /* writes Second */
        }
    }
}

```

Deterministic Parallel Java. Fig. 1 Regions and effects in DPJ (DPJ additions to Java are highlighted in bold)

region names have static scope, for example, all `first` fields in any instance of the `Pair` class reside in the same region `Pair.First`. (DPJ uses *region parameters*, explained below, to assign different regions to different object instances.) DPJ also provides *local regions* (declared within a method scope) for expressing effects on objects that do not escape the method.

Lines 5 and 8 illustrate the use of method effect summaries. The summary `writes First` in line 5 states that method `setFirst` writes region `First`, and similarly for `writes Second` in line 8. In general, a method effect summary has the form `reads region-list writes region-list`. If a method has no effect on the heap, it may be annotated `pure`. A method effect summary may be omitted entirely (as in line 11); in this case the compiler infers the most conservative effect (“writes the entire heap”) for the method. This default is generally used only for methods that are never called inside parallel tasks, so it is not important to know their precise effects.

A few simple rules govern the use of method effect summaries. First, all the actual effects of the method must be present in the summary. For example, the only heap effect in the body of `setFirst` is the write to field `first` in line 6, so the summary is correct. If the effect in line 5 were `pure`, the compiler would issue an error. (Line 6 also reads the method parameter `first`. However, method parameters and other local variables

in Java cannot have their addresses taken and never alias. Therefore, effects on such variables are handled automatically by the compiler and are ignored by the programmer-visible effect system.) Second, the summary may be conservative, that is, it may specify more effects than the method actually has. In particular, write effects subsume read effects, so it is permissible (but conservative) to say `writes R` in the summary when the method only reads region `R`. Finally, the effects of an overridden method must include the effects of any overriding method. This rule is similar to how plain Java handles `throws` clauses; it ensures sound reasoning about effects in the presence of polymorphic method dispatch.

Lines 12–15 illustrate the expression of deterministic parallelism. The `cobegin` block (line 12) says to execute each component statement in parallel. The compiler accumulates the effect of each component statement and checks that the effects are pairwise noninterfering. Here, the effect of invoking `setFirst` in line 13 is `writes First` (from the definition of method `setFirst` in line 5); and similarly the effect in line 14 is `writes Second`. Because `First` and `Second` are distinct names, the writes are to disjoint regions, so they are noninterfering. If the effects in lines 13 and 14 were interfering (e.g., they both wrote to the same region), then the compiler would issue a warning.

Region Parameters

DPJ allows classes and methods to be written with *region parameters* that become bound to actual regions when the class is instantiated into a type, or the method is invoked. Figure 2 illustrates the use of class region parameters. Line 1 declares a class `SimpleTree` with one region parameter `P`. Region parameter declarations coexist with Java generic type parameters and use a similar syntax; the keyword `region` distinguishes region parameters from type parameters. Line 3 places the instance field `data` in region `P`. When the class `SimpleTree` is instantiated with a type, and an object of that type is created with `new`, the type specifies the actual region of the storage for `data`, as illustrated in lines 4–5.

The compiler computes effects on fields by using the region specified in the class, after substituting actual regions for formal region parameters. This is illustrated

in lines 8–9. The effect of line 8 is `writes Left`, as shown, because the write is to field `left.data`. Line 3 says that `data` is in region `P`, and line 4 says that the type of `left` has `P = Left`. (There is also a read of field `left` in region `Left`, but this read is subsumed by the effect `writes Left`.) Similarly, the effect of line 9 is `writes Right`. Because `Left` and `Right` are distinct regions, the `cobegin` statement in lines 7–10 is legal. The DPJ type system ensures that this kind of reasoning is sound: for example, it is a type error to attempt to assign a reference of type `SimpleTree<Left>` to a variable of type `SimpleTree<Right>`.

Region parameters can have *disjointness constraints*. For example, `<region P1, P2 | P1 # P2>` declares two parameters `P1` and `P2` and constrains them to be disjoint. The compiler uses the constraint to check noninterference in parallel code that uses the parameters. If the disjointness constraint is not satisfied by the actual region arguments at the point where a class is instantiated or a method is invoked, the compiler issues a warning.

Region Path Lists (RPLs) and Nested Effects

In deterministic parallel computations, it is often essential to express a *nesting* relationship among regions. For example, to do a parallel update traversal on a binary tree, one must specify effects on “the left subtree” or “the right subtree.” A natural way to do this is to put a nesting structure on the regions that mirrors the nesting structure of the tree. DPJ represents this kind of nesting structure using *region path lists*, or RPLs.

An RPL is a colon-separated list of names beginning with `Root`, such as `Root:Left:Right`. RPLs naturally form a tree, where the RPL specifies the path in the tree from the root to the node that it represents. For example, `Root:Left:Right` is a child of `Root:Left`. In the execution semantics of DPJ, every region is represented as an RPL; a bare region name like `Left` is equivalent to `Root:Left`. RPLs may be *partially specified* by using `*` to stand in for any sequence of zero or more names, for example, `Root:*:Left` or `Root:Left:*`. This is useful in specifying *sets of regions* in types and effects.

[Figure 3](#) illustrates the use of RPLs to write a tree that can be traversed in parallel to update its elements. The key feature that makes this work is a *parameterized*

```
class Tree<region P> {
    region Left, Right;
    int data in P;
    Tree<P:Left> left in P:Left;
    Tree<P:Right> right in P:Right;
    int increment() writes P:*
    {
        ++data; /* writes P */
        cobegin {
            /* writes P:Left: */
            if (left != null) left.increment();
            /* writes P:Right: */
            if (right != null) right.increment();
        }
    }
}
```

Deterministic Parallel Java. [Fig. 3](#) A tree class

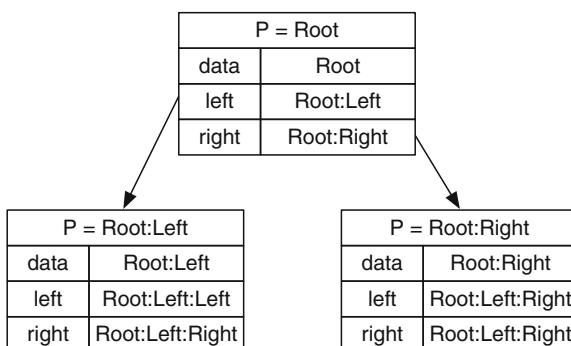
```
class SimpleTree<region P> {
    region Left, Right
    int data in P;
    SimpleTree<Left> left in Left = new SimpleTree<Left>();
    SimpleTree<Right> right in Right = new SimpleTree<Right>();
    void updateChildren(int leftData, int rightData) {
        cobegin {
            left.data = leftData; /* writes Left */
            right.data = rightData; /* writes Right */
        }
    }
}
```

Deterministic Parallel Java. [Fig. 2](#) Class region parameters

RPL: an RPL can begin with a parameter P , as shown in lines 4–5. In the execution semantics, the parameters are erased via left recursive substitution. [Figure 4](#) illustrates this procedure for the root node of a tree, and its two children. Each node of the tree has its `data` field in a distinct region, the RPL of which reflects the position of the node in the tree. Further, the DPJ type system enforces this structure: for example, it would be illegal to assign the right child to the `left` field of the root, because the types do not match.

Lines 6–14 show how to write an increment method that traverses the tree recursively in parallel and updates the data fields of the nodes. In line 6, the summary writes $P:\ast$ says that the method writes P and all regions under P . The effect of line 7 is writes P , because field `data` is declared in region P (line 3). Line 10 generates two effects: a read of $P:\text{Left}$ due to the read of field `left`, and a write to $P:\text{Left}:\ast$ obtained from the effect summary of the recursively invoked method `increment`, after substituting $P = P:\text{Left}$ from the type of `left`. Because writes subsume reads, both effects may be summarized as writes $P:\text{Left}:\ast$, as shown in line 9. Similarly, the effect of line 12 is writes $P:\text{Right}:\ast$.

As described above, the compiler has to check two things: first, that the method summary is correct; and second, that the statements inside the `cobegin` (line 8) are noninterfering. The first check passes, because all the statement effects are included in the effect writes $P:\ast$ stated in the summary in line 6. As to the second check, the effects of the two statements in the `cobegin` are writes $P:\text{Left}:\ast$ and writes $P:\text{Right}:\ast$. Because of the tree structure of



Deterministic Parallel Java. [Fig. 4](#) Runtime representation of the tree

RPLs, these two effects are on disjoint sets of regions for any common binding to P . Therefore the effects are noninterfering, so this check passes as well.

Arrays

The RPL mechanism is useful for trees, as shown above. It also supports two common patterns of parallel computation on arrays that are not well handled by previous type and effect systems: parallel updates of objects through arrays of references and divide-and-conquer updates to arrays.

Parallel updates of objects through arrays of references. DPJ supports this pattern with two features. First, an RPL may include an element $[e]$, where e is an integer expression, representing cell e of an array. This is called an *array RPL element*. Second, the region in the type of an array cell may be parameterized by the index of the cell. This is called an *index-parameterized array*. Together, these features allow the programmer to specify an array of references such that the object pointed to by each reference is instantiated with a distinct region.

[Figure 5](#) shows an example. The class `Body` has one region parameter P and a field `force` in region P . The instance method `computeForce` computes a force and writes it into `force`. The static method `computeForces` takes an array of bodies and iterates over it in parallel, calling `computeForce` on each body. The type of `bodies`, shown in line 7, is an index-parameterized array type. The `#i` declares a fresh variable i in scope over the whole array type. The element type `Body<[i]>` says that for any natural number n , the array element at index n is a reference of type `Body<Root: [n]>`.

[Figure 6](#) shows how the array might look at runtime. The assignment rules in the DPJ type system ensure that the types are correct: for instance `bodies[10]` must point to an object of type `Body<Root: [10]>`. Therefore, all the `force` fields are in distinct regions, and the parallel updates in line 10 of [Fig. 5](#) are noninterfering.

Divide-and-conquer updates to arrays. To support this pattern, DPJ allows *dynamic array partitioning*: an array may be divided into two (or more) disjoint parts that are updated in parallel. [Figure 8](#) illustrates how this works for a simple version of quicksort. The `quicksort` method (line 2) has a region parameter P and takes a `DPJArray<P>`, which points into a contiguous subset of a Java array. In lines 6–7, the array is

```

class Body<region P> {
    double force in P;
    void computeForce() reads Root writes P {
        ...
    }
    static void
    computeForces(Body<i>[]#i bodies) {
        foreach (int i in 0, bodies.length) {
            /* reads Root writes Root:[i] */
            bodies[i].computeForce();
        }
    }
}

```

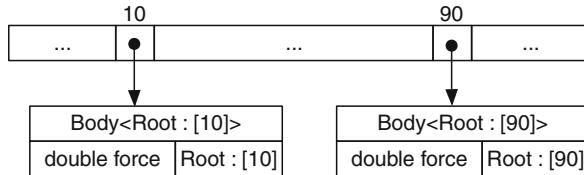
Deterministic Parallel Java. Fig. 5 Code using an index-parameterized array

```

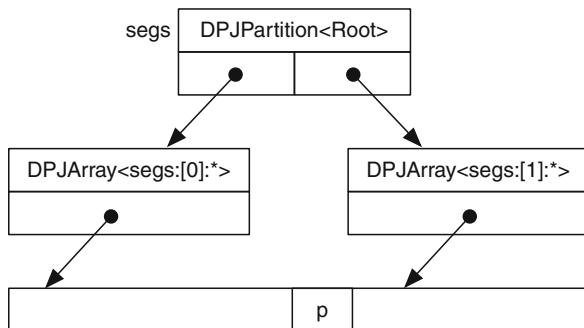
static <region P>
void quicksort(DPJArray<P> A) writes P::* {
    /* Ordinary quicksort partition */
    int p = quicksortPartition(A);
    /* Split array int two disjoint pieces */
    final DPJPartition<P> segs =
        new DPJPartition<P>(A, p);
    cobegin {
        /* writes segs:[0]:* */
        quicksort(segs.get(0));
        /* writes segs:[1]:* */
        quicksort(segs.get(1));
    }
}

```

Deterministic Parallel Java. Fig. 8 Parallel quicksort



Deterministic Parallel Java. Fig. 6 Runtime representation of the index-parameterized array



Deterministic Parallel Java. Fig. 7 Runtime representation of the dynamically partitioned array

split at index p , creating a $DPJPartition$ object holding references to two disjoint subarrays. The $cobegin$ in lines 8–13 calls $quicksort$ recursively on these two subarrays.

Figure 7 shows what this partitioning looks like at runtime, for the root of the tree and its children. A

$DPJPartition$ object referred to by the final local variable $segs$ points to two $DPJArray$ objects, each of which points to a segment of the original array. The $DPJArray$ objects are instantiated with *owner RPLs* $segs:[0]:*$ and $segs:[1]:*$ in their types. An owner RPL is like an RPL, except that it begins with a final local reference variable instead of $Root$. This allows different partitions of the same array to be represented. Again because of the tree structure of DPJ regions, $segs:[0]:*$ and $segs:[1]:*$ are disjoint region sets, and so the effects in lines 9 and 11 are noninterfering. Also, as in ownership type systems, region $segs$ is under the region P bound to the first parameter of its type, so the effect summary $writes P:*$ in line 2 covers the effects of the method body.

Commutativity Annotations

Sometimes, a parallel computation is deterministic, even if it has interfering reads and writes. For example, two inserts to a concurrent set can go in either order, and preserve determinism, even though there are interfering writes to the set object. DPJ supports this kind of computation with a *commutativity annotation* on methods (typically API methods for a concurrent data structure). For example:

```

interface Set<type T, region R> {
    commutative void add(T e) writes R;
}

```

This annotation is provided by a library or framework programmer and trusted by the compiler (it is not checked by the DPJ effect system). It means that `cobegin { add(e1); add(e2); }` is equivalent to doing the `add` operations in sequence (isolation) *and* that both sequence orders are equivalent (commutativity).

When the compiler encounters a method invocation, it generates an *invocation effect* that records both the method that was invoked, and the read-write effects of that method:

```
foreach (int i in 0, n) {
    /* invokes Set.add with writes R */
    set.add(A[i]);
}
```

The compiler uses the commutativity annotations to check interference of effect. For example, the effect `invokes Set.add with writes R` is noninterfering with itself (because `add` is declared commutative). However, the same effect interferes with `invokes Set.size with reads R`.

Expressivity and Limitations

DPJ can express a wide variety of algorithms that use a fork-join style of parallelism and access data through a combination of (1) read-only sharing; (2) disjoint updates to arrays and trees; (3) commutative operations on concurrent data structures like sets and maps; and (4) accesses to task-private heap objects. DPJ has been used to write the following parallel algorithms, among others: IDEA encryption, sorting, k-means clustering, an n-body force computation, collision detection algorithm from a game engine, and a Monte Carlo financial simulation. The main advantages of the DPJ approach are (1) a compile-time guarantee of determinism in all executions; and (2) no overhead from checking determinism at runtime, as the region information is erased before parallel code generation. Further, the effect annotations provide checkable documentation and enable modular analysis, and the defaults allow incremental transformation of sequential to parallel programs.

DPJ does have some limitations. One limitation, mentioned above, is that currently only fork-join parallelism can be expressed. A second limitation is that

to ensure soundness, the language disallows some patterns that are in fact deterministic, but cannot be proved deterministic by the type system. One example is that the type system disallows reshuffling of the elements in an index-parameterized array. This limitation poses a problem for the Barnes-Hut n-body force computation, as rearranging the array of bodies is required for good locality. A workaround for this limitation is to recopy the bodies on reinsertion into the array, but this is somewhat clumsy and could degrade performance. A third limitation is that some programmer effort is required to write the DPJ annotations, although this effort is repaid by the benefits discussed above.

D

Ongoing Work

The foregoing describes the state of the public release of DPJ as of October 2010. The following subsections describe ongoing work that is expected to be part of future releases.

Controlled Nondeterminism

DPJ is being extended to support controlled nondeterminism [6], for algorithms such as branch and bound search, where several correct answers are equally acceptable, and it would unduly constrain the schedule to require the same answer on every execution. The key new language features for controlled nondeterminism are (1) `foreach_nd` and `cobegin_nd` statements that operate like `foreach` and `cobegin`, except they allow interference among their parallel tasks; (2) `atomic` statements supported by software transactional memory (STM); and (3) *atomic regions* and *atomic effects* for reasoning about which memory regions may have interfering effects, and where effects occur inside `atomic` statements.

Together, these features provide several strong guarantees for nondeterministic programs. First, the extended language is *deterministic by default*: the program is guaranteed to be deterministic *unless* the programmer explicitly introduces nondeterminism with `foreach_nd` or `cobegin_nd`. Second, the extended language provides both strong isolation (i.e., the program behaves as if it is a sequential interleaving of `atomic` statements and unguarded memory accesses) and data race freedom. This is true even if

the underlying STM provides only weak isolation (i.e., it allows interleavings between unguarded accesses and `atomic` statements). Third, `foreach` and `cobegin` are guaranteed to behave as isolated statements (as if they are enclosed in an `atomic` statement), even inside a `foreach_nd` or `cobegin_nd`. Finally, the extended type and effect system allows the compiler to boost the STM performance by removing unnecessary synchronization for memory accesses that can never cause interference.

Object-Oriented Parallel Frameworks

DPJ is being extended to support object-oriented parallel frameworks. Current support focuses on two kinds of frameworks: (1) collection frameworks like Java’s `ParallelArray` [1] that provide parallel operations such as `map`, `reduce`, `filter`, and `scan` on their elements via user-defined methods; and (2) a framework for pipeline parallelism, similar to the pipeline framework in Intel’s Threading Building Blocks [11]. The key challenge is to prevent the user’s methods from causing interference when invoked by the parallel framework. For example, a user-defined `map` function must not do an unsynchronized write to a global variable. The OO framework support incorporates several extensions to the DPJ type and effect system for expressing generic types and effects in framework APIs, with appropriate constraints on the effects of user-defined methods.

Frameworks represent one possible solution to the problem of array reshuffling and other operations that the type system cannot express. Such operations can be encapsulated in a framework. The type and effect system can then check the *uses* of the framework, while the framework *internals* are checked by more flexible (but more complex and/or weaker) methods such as program logic or testing. This approach separates concerns between framework designer and user, and it fosters modular checking. It also enables different forms of verification with different trade-offs in complexity and power to work together. In addition, frameworks provide a natural way to extend the fork-join model of DPJ by adding other parallel control abstractions, including finer-grain synchronization (e.g., pipelined loops) or domain-specific abstractions (e.g., kd-tree querying and sparse matrix computations).

Inferring DPJ Annotations

Research is underway to develop an interactive tool that helps the user write DPJ programs by partially inferring annotations. A tool called DPJizer has been developed [14] that uses interprocedural constraint solving to infer method effect summaries for DPJ programs annotated with the other type, region, and effect information. DPJizer simplifies DPJ development by automating one of the more tedious, yet straightforward, aspects of writing DPJ annotations. The tool also reports the effect of each program statement, helping the programmer to isolate and eliminate effects that are causing unwanted interference.

The next step is to infer region information, given a program annotated with parallel constructs. This is a more challenging goal, because there are many more degrees of freedom. The current strategy is to have the programmer provide partial information (e.g., partition the heap into regions and put in the parallel constructs), and then have the tool infer types and effects that guarantee noninterference.

Runtime Checks

Research is underway to add runtime checking as a complementary mode for guaranteeing determinism, in addition to the compile-time type and effect checking. The advantage of runtime checking is that it is more precise. The disadvantages are (1) it adds overhead, so it is probably useful only for testing and debugging; and (2) it provides only a fail-stop check, instead of a compile-time guarantee of correctness. One place where runtime checking could be particularly useful is in relaxing the prohibition on inconsistent type assignments (e.g., in reshuffling an index-parameterized array). In particular, if the runtime can guarantee that a reference is unique, then a safe cast to a different type can be performed at runtime. Using both static and runtime checks, and providing good integration between the two, would offer the user a powerful set of options for managing the trade-offs among the expressivity, performance, and correctness guarantees of the different approaches.

Related Entries

- ▶ [Determinacy](#)
- ▶ [Formal Methods-Based Tools for Race, Deadlock, and Other Errors](#)
- ▶ [Transactional Memories](#)

Bibliographic Notes and Further Reading

For further information on the DPJ project and its goals, see the DPJ web site [2] and the position paper presented at HotPar 2009 [4]. The primary reference for the DPJ type and effect system is the paper presented at OOPSLA 2009 [5]. A paper on the nondeterminism work will appear at POPL 2011 [6]. The Ph.D. thesis of Robert Bocchino [3] presents the full language, together with a formal core language and effect system, soundness results for the core system, and proofs. A paper presented at ASE 2009 [14] describes the work on inferring method effect summaries.

The main influence on DPJ has been the prior work on type and effect systems, including FX [10] and the various effect systems based on ownership types [7, 8]. Other influences include work on design-by-contract for framework APIs [13], work on type and effect inference [12], and the vast body of literature on transactional memory [9].

Bibliography

1. <http://gee.cs.oswego.edu/dl/jsr166/dist/extral66ydocs/index.html?extral66y/package-tree.html>
2. <http://dpj.cs.uiuc.edu>
3. Bocchino RL Jr (2010) An effect system and language for deterministic-by-default parallel programming. PhD thesis, University of Illinois, Urbana-Champaign
4. Bocchino RL Jr, Adve VS, Adve SV, Snir M (2009) Parallel programming must be deterministic by default. In: HOTPAR ’09: USENIX workshop on hot topics in parallelism, Berkeley, March 2009
5. Bocchino RL Jr, Adve VS, Dig D, Adve SV, Heumann S, Komuravelli R, Overbey J, Simmons P, Sung H, Vakilian M (2009) A type and effect system for deterministic parallel Java. In: OOPSLA ’09: Proceedings of the 24th ACM SIGPLAN conference on object-oriented programming systems, languages, and applications, New York, pp 97–116
6. Bocchino RL Jr, Heumann S, Honarmand N, Adve S, Adve V, Welc A, Shpeisman T, Ni Y (2011) Safe nondeterminism in a deterministic-by-default parallel language. In: POPL ’11: Proceedings of the 38th ACM SIGACT-SIGPLAN symposium on principles of programming languages, New York
7. Cameron NR, Drossopoulou S, Noble J, Smith MJ (2007) Multiple ownership. In: OOPSLA ’07: Proceedings of the 22nd ACM SIGPLAN conference on object-oriented programming systems and applications, New York, pp 441–460
8. Clarke D, Drossopoulou S (2002) Ownership, encapsulation and the disjointness of type and effect. In: OOPSLA ’02: Proceedings of the 17th ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications, New York, pp 292–310
9. Harris T, Larus J, Rajwar R (2010) Transactional memory (Synthesis lectures on computer architecture), 2nd edn. Morgan & Claypool Publishers, San Rafael
10. Lucassen JM, Gifford DK (1988) Polymorphic effect systems. In: POPL ’88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on principles of programming languages, New York, pp 47–57
11. Reinders J (2007) Intel threading building blocks: outfitting C++ for multi-core processor parallelism. O’Reilly Media Sebastopol
12. Talpin J-P, Jouvelot P (1992) Polymorphic type, region and effect inference. J Funct Program 2:245–271
13. Thomas P, Weedon R (1998) Object-oriented programming in Eiffel, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston
14. Vakilian M, Dig D, Bocchino R, Overbey J, Adve V, Johnson R (2009) Inferring method effect summaries for nested heap regions. In: ASE ’09: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, pp 421–432

Direct Schemes

► Dense Linear System Solvers

Distributed Computer

- Clusters
- Distributed-Memory Multiprocessor
- Hypercubes and Meshes

Distributed Hash Table (DHT)

► Peer-to-Peer

Distributed Logic Languages

► Logic Languages

Distributed Memory Computers

- Clusters
- Disturbed-Memory Multiprocess
- Hypercubes and Meshes

Distributed Process Management

► [Single System Image](#)

Distributed Switched Networks

► [Networks, Direct](#)

► [Networks, Multistage](#)

Distributed-Memory Multiprocessor

MARC SNIR

University of Illinois at Urbana-Champaign, Urbana, IL, USA

Synonyms

[Distributed computer](#); [Distributed Memory Computers](#); [Massively parallel processor \(MPP\)](#); [Multicomputers](#); [Multiprocessors](#); [Server farm](#)

Definition

A computer system consisting of a multiplicity of processors, each with its own local memory, connected via a network. Load or store instructions issued by a processor can only address the local memory, and different mechanisms are provided for global communication.

Discussion

Terminology

A Distributed-Memory Multiprocessor (DMM) is built by connecting *nodes*, which consist of uniprocessors or of shared memory multiprocessors (SMPs), via a *network*, also called *Interconnection Network* (IN) or *Switch*. While the terminology is fuzzy, *Cluster* generally refers to a DMM mostly built of commodity components, while *Massively Parallel Processor* (MPP) generally refers to a DMM built of more specialized components that can scale to a larger number of nodes and is used for large, compute-intensive tasks – in particular, in scientific computing. A *server farm* is a cluster

used as a server, in order to provide more performance or better cost-performance than a shared memory multiprocessor server. A *high-availability* (HA) *cluster* is a cluster with software and possibly hardware that supports fast failover so as to tolerate component failures. Finally, a *cloud* is a (generally very large) cluster that supports dynamic provisioning of computing resources, usually virtualized, to remote users, via the Internet.

Historical Background

High-performance computing was focused until the late 1970s on SMPs and vector computers built of bipolar gates. The fast evolution of VLSI technology indicated the possibility of achieving much better cost-performance by assembling a larger number of less performing, but much cheaper, microprocessors. The 64 node Cosmic Cube built at Caltech in the early 1980s was an early demonstration of this technology [9]. Several companies started manufacturing such systems in the mid-1980s; a 1,024 node nCUBE system was installed at Sandia National Lab in the late 1980s and a 400 Transputer nodes system was running at Edinburgh in 1989. The success of these systems, which was documented in the influential talk “The Attack of the Killer Micros” by Eugene D. Brooks III at the Supercomputing 1990 Conference, led to a flurry of commercial and governmental activities. Companies such as Thinking Machines (CM-5), Intel (iPSC, Paragon), and Meiko (CS-2) started offering Distributed-Memory Multiprocessor (DMM) products in the mid-1980s, followed by Fujitsu (AP100), IBM (SP1, SP2), and Cray (T3D, T3E) in the early 1990s. DMSs have dominated the TOP500 list of fastest supercomputers since the early 1990s.

The use of clusters evolved in parallel, and largely independently, in the commercial world, driven by several trends: Clusters were used in the early 1990s for *server consolidation*, replacing multiple, distinct servers with one cluster, thus reducing operating costs. The explosive growth of the Internet in the late 1990s led to a need for large clusters with a small footprint (to fit in downtown buildings) that can be quickly expanded, to serve Internet companies. Web requests can easily be served on a DMM, as each TCP/IP request is a small, independent action and request failures can be tolerated.

DMMs were also used to extend SMP architectures beyond the point where hardware could conveniently support coherent shared memory. Thus, IBM introduced in 1994 the *parallel sysplex* architecture that allowed coupling of multiple S/390 mainframes into a larger cluster. The use of such clusters became more widespread as software vendors developed distributed computing firmware and adapted applications to run on clusters.

Clusters have become increasingly used since the late 1990s for providing fault tolerance. Rather than using very specialized hardware, such as in the early Tandem computers, *High Availability (HA) Clusters* provide fault tolerance through hardware replication that eliminates single points of failure, and suitable software. Many vendors (IBM, Microsoft, HP, Sun, etc.) offer HA cluster software solutions.

Utility computing, namely the provisioning of computing resources (CPU time and storage) using a pay-per-use model, has been pursued by multiple companies over decades. Changes in technology (low-cost, high-bandwidth networking, decline in hardware cost and relative increase in operation cost, increased use of off-the-shelf software and improved support for hardware virtualization) and in market needs (very rapid and unpredictable growth of the compute needs of Internet companies) have led to the recent success of this model, under the name of *cloud computing*. Cloud computing platforms are large clusters with software that supports the dynamic provisioning of virtualized compute and storage resources (hardware as a service – HaaS) and, possibly, firmware and application software (software as a service – SaaS)

Architecture

At its simplest, a DMM can be assembled by connecting personal computers or workstations with a commercial network such as Ethernet, as shown in Fig. 1. The Beowulf system built in 1994 is an early example [2]. A more compact and more maintainable cluster can be assembled from rack-mounted servers, as shown in Fig. 2. In addition to the network that is used for communication between nodes, such a cluster will have a control network that is used for controlling and monitoring each node from a centralized console.

The network of a DMM can be (a) a commodity Local Area Network (LAN), such as Ethernet;



Distributed-Memory Multiprocessor. Fig. 1 Beowulf cluster

(b) a higher performance, third party switched network, such as Infiniband or Myrinet; or (c) a vendor-specific network, such as available on IBM and Cray systems. Communication over Ethernet will typically use the TCP/IP protocols; communication over non-Ethernet fabrics can support lower latency protocols. Adapters for networks of type (a) and (b) will connect to a standard I/O bus, such as PCIe; vendor-specific networks can connect to the (nonstandard) memory bus, thus achieving higher bandwidth and lower latency.

The basic communication protocol supported by any of these networks is *message passing*: Two nodes communicate when one node sends a message to the second node and the second node receives the message. Networks of type (b) and (c) often support *remote direct memory access* (rDMA): A node can directly access data in the memory of another node, using a *get* operation, or update the remote memory, using a *put* operation.



Distributed-Memory Multiprocessor. Fig. 2 Rack-mounted cluster

Support for such operations in user space requires an adapter that can translate virtual addresses to physical addresses. Oftentimes, such an adapter will also support read-modify-write operations on operands in remote memory, such as *increment*; this avoids the need for a round-trip for such operations and facilitates their atomic execution.

Networks of type (c) may have hardware support for collective operations, such as *barrier*, *broadcast*, and *reduce*.

The *topology* of the interconnection network impacts its performance: Networks with a low-dimension *mesh* topology (2D or 3D) can be packaged so that only short wires are needed; this enables the use of cheaper copper cables while maintaining high bandwidth. *Radix-network* topologies, such as *butterfly* or *fat-tree*, reduce the average number of hops between nodes, but require longer wires [11]. Optical cables are needed for high-speed transmission over long distances.

Commercial clusters (server farms and clouds) typically use Ethernet networking technology and communicate using TCP/IP protocols. The nodes often share storage, using Storage Area Networks (SAN) or

Network Attached Storage (NAS) technologies. Some commercial clusters, such as the IBM Parallel Sysplex, provide clustering hardware that accelerates data sharing and synchronization across nodes.

Software

DMMs reuse, on each node, software designed for node-sized uniprocessors or SMPs. This includes the operating system, programming languages, compilers, libraries, and tools. An additional layer of parallel software is built atop the node software, in order to integrate the multiple nodes into one system.

This includes, for compute clusters:

- Software for system initialization, monitoring and control. This enables the control of a large number of nodes from one console; concurrent booting of a large number of nodes; network firmware initialization; error detection; etc.
- Scheduler and resource manager for allocating resources to parallel jobs, loading them and starting them. Parallel jobs often run on dedicated partitions; the scheduler has to allocate nodes in large chunks, while ensuring high node utilization and reducing waiting time for batch jobs. The resource manager has to reduce job-loading time (e.g., by using a parallel tree broadcast algorithm) and ensure that all nodes of a job start correctly.
- Parallel file system. Such a system integrates a large number of disks into one system and provides scalable file coherence protocols. Files are striped across many nodes so as to support parallel access to different parts of the same file and ensure load balancing. Software RAID schemes may be used to enhance the reliability of such a large file system.
- Checkpoint/restart facilities. On large DMMS, the mean time between failures is often shorter than the running time of large jobs. To ensure job completion, jobs periodically checkpoint their state; if a failure occurs, then job execution is restarted from the last checkpoint. A checkpoint facility supports coordinated checkpoint or restart by many nodes.
- Message passing libraries for supporting communication across nodes. MPI is the de facto standard for message passing on large DMMs.
- Tools for debugging and performance monitoring of parallel jobs. These are often built atop node

level debugging and performance monitoring; they enable users to control many nodes in debug mode, and to aggregate information from many nodes in debug or monitoring nodes.

The use of conventional software on the nodes of DMMs is limiting; in particular, the use of a conventional operating systems on each node results in *jitter*: Background OS activities may slow down nodes at random times; applications where barriers are used frequently can slow down significantly as all threads wait at each barrier to the slowest thread. Some systems, e.g., the IBM Blue Gene system, are using on their compute node-specialized *light-weight kernels* (LWK) that provide only critical services and offload to remote server nodes heavier or more asynchronous system services.

Commercial clusters use TCP/IP for inter-node communication, and distributed file systems for file sharing. They leverage firmware and application software designed for distributed systems, such as support for remote procedure calls (or remote method invocations), distributed transaction processing monitors, messaging frameworks, and “shared-nothing” databases.

A critical component of server farms that handle Web requests is a *load balancer* that distributes TCP/IP requests coming on one port to distinct nodes.

HA clusters have software that monitor the health of the system components and its applications (heartbeat) and report failures; software to achieve consensus after failures and to initiate recovery actions; and application-specific recovery software to restart each application in a consistent state. Disk state is preserved, either through mirroring, or through the sharing of highly reliable RAID storage via multiple connections.

Clouds often use virtual machine software to virtualize processors and storage, thus facilitating their dynamic allocation. They need a resource management infrastructure that can allocate virtual machines on demand.

Research

Early DMMs had directly connected networks; the network topology was part of the programming model. Theoretical research focused on the mapping of parallel algorithms to specific topologies; in particular, hypercubes and related networks (the primary conference

on DMMs in the late 1980s was the conference on Hypercube Concurrent Computers and Applications). The abstract model used was of unit time communication on each edge, with communications either being limited to one incident edge per node at a time, or occurring concurrently on all edges incident to a node. For example, the problem of sorting on a hypercube is treated by several tens of publications.

As indirect networks became more prevalent, significant amount of research focused on the topology of routing networks and cost-performance tradeoffs, and routing protocols for different topologies that are efficient and deadlock free, and that can handle failures.

Modern DMMs largely hide the network topology from the user and may not provide mechanisms for controlling the mapping of processes and communications onto the network nodes and edges. Therefore, the topology is not part of the programming model. The abstract programming models used to represent such systems assume that communication time between two nodes is a function of the size of the message communicated. Two of the most frequently used models are the postal model [7] and the logP model [5].

System research on DMMs has focused on communication protocols, operating systems, file systems, programming environments, and applications. Some of the larger efforts in this area are the Network of Workstation (NOW) effort at Berkeley [1], and the SHRIMP project at Princeton [3].

Related Entries

- [Checkpointing](#)
- [Clusters](#)
- [Connection Machine](#)
- [Cray T3E](#)
- [Distributed-Memory Multiprocessor](#)
- [Ethernet](#)
- [Hypercubes and Meshes](#)
- [IBM RS/6000 SP](#)
- [InfiniBand](#)
- [Meiko](#)
- [Moore's Law](#)
- [MPI \(Message Passing Interface\)](#)
- [Myrinet](#)
- [Network Interfaces](#)
- [Network of Workstations](#)
- [Networks, Direct](#)

- ▶ [Networks, Multistage](#)
- ▶ [Parallel Computing](#)
- ▶ [PCI Express](#)
- ▶ [Routing \(Including Deadlock Avoidance\)](#)

Bibliographic Notes and Further Reading

The book of Fox et al. [9] covers the early history of MPPs, their hardware and software, and the applications enabled on them. The book of Culler and Singh [6] is a good introduction to MPP architectures and their networks, with a detailed description of the Cray T3D and IBM SP networks. The book of Pfister [11] is a general introduction to clusters, their design, and their use, while the two books of Buyya [4, 5] provide in-depth coverage of cluster hardware, software, and applications. The book of Dally and Towles [8] provides a very good coverage of IN theory and practice. Finally, the book of Leighton [10] is an excellent introduction to the theory results that are relevant to DMMs and INs.

Bibliography

1. Anderson TE, Culler DE, Patterson DA, the NOW team (1995) A case for NOW (Networks of Workstations). *IEEE Micro* 15(1):54–64
2. Bar-Noy A, Kipnis S (1992) Designing broadcasting algorithms in the postal model for message-passing systems. In: Proceedings of the fourth annual ACM symposium on parallel algorithms and architectures SPAA '92. San Diego, California, United States, June 29–July 01, 1992. ACM, New York, pp 13–22
3. Blumrich MA, Li K, Alpert R, Dubnicki C, Felten EW, Sandberg J (1998) Virtual memory mapped network interface for the SHRIMP multicomputer. In: Sohi GS (ed) 25 years of the international symposia on computer architecture (selected papers) (ISCA '98). ACM Press, New York, pp 473–484
4. Buyya R (1999) High performance cluster computing: architectures and systems, vol 1. Prentice-Hall, New Jersey
5. Buyya R (1999) High performance cluster computing: programming and applications, vol 2. Prentice-Hall, New Jersey
6. Culler DF, Singh JP (1999) Parallel computer architecture: a hardware/software approach. Morgan Kaufman, San Francisco
7. Culler D, Karp R, Patterson D, Sahay A, Schausler EK, Santos E, Subramonian R, von Eicken T (1993) LogP: towards a realistic model of parallel computation. In: Proceedings of the fourth ACM SIGPLAN symposium on principles and practice of parallel programming, San Diego, California, United States, 19–22 May 1993, pp 1–12
8. Dally WJ, Towles BP (2004) Principles and practices of interconnection networks. Morgan Kaufmann, San Francisco
9. Fox GC, Williams RD, Messina PC (1994) Parallel computing works, Morgan Kaufman, San Francisco

10. Leighton FT (1991) Introduction to parallel algorithms and architectures: arrays, trees, hypercubes. Morgan Kaufman, San Francisco
11. Pfister GF (1997) In search of clusters, 2nd edn, Prentice Hall, New Jersey
12. Sterling T, Becker DJ, Savarese D, Dorband JE, Ranawake UA, Packer CV (1995) BEOWULF: a parallel workstation for scientific computation. In: Proceedings of the 24th international conference on parallel processing. Oconomowoc, Wisconsin, pp II–14

Ditonic Sorting

- ▶ [Bitonic Sort](#)

DLPAR

- ▶ [Dynamic Logical Partitioning for POWER Systems](#)

Doall Loops

- ▶ [Loops, Parallel](#)

Domain Decomposition

THOMAS GEORGE¹, VIVEK SARIN²

¹IBM Research, Delhi, India

²Texas A&M University, College Station, TX, USA

Synonyms

[Functional decomposition](#); [Grid partitioning](#)

Definition

Domain decomposition, in the context of parallel computing, refers to partitioning of computational work among multiple processors by distributing the computational domain of a problem, in other words, data associated with the problem. In the scientific computing literature, domain decomposition mainly refers to techniques for solving partial differential equations (PDE) by iteratively solving subproblems corresponding to smaller subdomains. Although the evolution of these

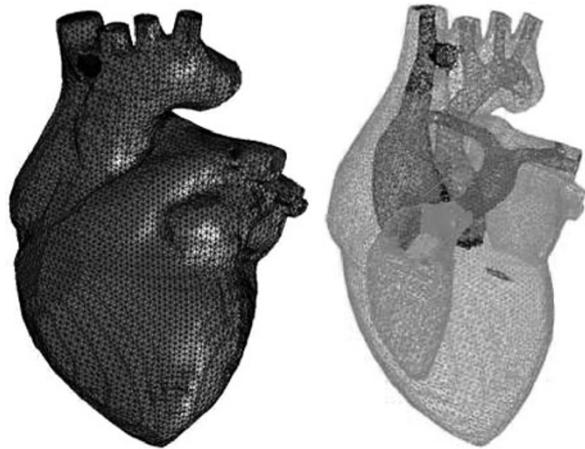
techniques is motivated by PDE-based computational simulations, the general methodology is applicable in a number of scientific domains not dominated by PDEs.

Introduction

One of the key steps in parallel computing is partitioning the problem to be solved into multiple smaller components that can be addressed simultaneously by separate processors with minimal communication. There are two main approaches for dividing the computational work. The first one is domain decomposition, which involves partitioning the data associated with the problem into smaller chunks and each parallel processor works on a portion of the data. The second approach, functional decomposition, on the other hand, focuses on the computational steps rather than the data and involves processors executing different independent portions of the same program concurrently. Most parallel program designs involve a combination of these two complementary approaches.

In this entry, we focus on domain decomposition techniques, which are based on data parallelism. [Figure 1](#) shows a 3-D finite element mesh model of the human heart [29] used for simulation of the human cardiovascular system and in particular, the flow of blood through the various chambers and valves (http://www.andrew.cmu.edu/user/jessicaZ/medical_data/Heart_Valve_new.htm). Computation is performed repeatedly for each point on the mesh using Darcy's law [19], which describes the flow of a fluid through a porous medium and is widely used in blood flow simulations. [Figure 1](#) also illustrates decomposition of this mesh model into 22 subdomains (indicated by different colors) that are relatively easier to model due to their simpler geometry.

Domain decomposition is vital for high-performance scientific computing. In addition to providing enhanced parallelism, it also offers a practical framework for solving computational problems over complex, heterogeneous and irregular domains by decomposing them into homogeneous subdomains with regular geometries. The subdomains are each modeled separately using faster, simpler computational algorithms, and the solutions are then carefully merged to ensure consistency at the boundary cases. Such decomposition is also useful when the modeling equations are different across subdomains as in the case of singularities and



Domain Decomposition. [Fig. 1](#) Tetrahedral meshing of the heart. The figure on the *right* shows the mesh partitioned into subdomains indicated with different colors

anomalous regions. Lastly, it is also useful for developing out-of-core (or external memory) solution techniques for large computational problems with billions of unknowns by facilitating the partitioning of the problem domain into smaller chunks, each of which can fit into main memory.

Domain Decomposition for PDEs

Scientific computing applications such as weather modeling, heat transfer, and fluid mechanics involve modeling complex physical systems via partial differential equations. Though there exist many different classes of PDEs, the most widely used ones are second-order elliptic equations, for example, the Poisson's equation, which manifests in many different forms in a number of applications. For example, the Fourier law for thermal conduction, Darcy's law for flows in porous media and Ohm's law for electrical networks, all arise as special cases of the Poisson's equation.

Example 1. For the sake of illustration, we consider a problem that requires solving the Poisson equation over a complex domain Ω with Γ denoting the boundary

$$\nabla^2 u = f \text{ in } \Omega,$$

$$u = u_\Gamma \text{ on } \Gamma.$$

In practice, the domain is discretized using either finite difference or finite element methods, which yields a large sparse symmetric positive definite linear system, $Au = f$ where A is determined by the discretization and u is a vector of unknowns. Domain decomposition for this problem involves partitioning the domain Ω into subdomains $\{\Omega_i\}_{i=1}^s$, which may be overlapping or nonoverlapping depending on the nature of the problem. The partitioning into subdomains is sometimes referred to as the coarse grid with the original discretization being described as the fine grid.

Given a discretized computational problem as in the example, there are four main steps that need to be addressed in order to obtain an efficient parallel solution.

- The first step is to identify the preferred characteristics of subdomains for the given problem. This includes choosing the number of subdomains, the type of partitioning (element/edge/vertex-based), and the level of overlap between the subdomains (with disjoint subdomains being the extreme case). Each of these choices depend on a variety of factors such as size, type, and geometry of the problem domain, the number of parallel processors, and the PDE/linear system being solved.
- The next step is to partition the problem domain into the desired subdomains. This can be achieved by using a variety of geometric and graph partitioning algorithms. For certain applications involving adaptive mesh refinement, one may chose to iteratively refine the partitioning based on the load imbalance.
- Given the subdomains, the next step involves solving the PDE for the entire problem domain. This consists of :
 - Solving the PDE over each subdomain, that is, $\nabla^2 u = f$ in Ω_i , for $1 \leq i \leq s$, or, equivalently the linear systems $A_i u_i = f_i$, for $1 \leq i \leq s$, where A_i , u_i , and f_i are restrictions of A , u , and f to Ω_i , respectively
 - Ensuring consistency at the interfaces (or the overlap in case of overlapping regions) of the subdomains, which is referred to as the coarse solve
- Depending on the overlap between the subdomains, there are two classes of solution approaches.

Specifically, Schur's complement techniques are suitable for subdomains with no overlap and Schwarz alternating techniques are suited for overlapping subdomains. Since these solution techniques iteratively solve the constituent subproblems, these subproblems need not be solved accurately in each step and different choices of approximation over the subproblems lead to variants of the overall algorithms.

- Given a parallel architecture, the final step is to map the subdomains and the interfaces to the individual processors and translate the domain-decomposition-based solution approach to a parallel program that can be implemented efficiently on this architecture. This mapping depends on the relative computational and communication requirements for solving the interfaces.

Each of the above steps will be discussed in the following sections.

Nature of Subdomains

The effectiveness of any domain decomposition approach critically depends on the relative suitability of the sub-domains both with respect to the computational problem (PDE/linear system) being solved and the parallel architecture being employed. Below, we discuss certain key characteristics of subdomains that influence the overall performance as well as some general guidelines proposed in literature for choosing them for a given problem.

Number of Subdomains

The choice of the number of subdomains has a significant effect on both the computational time and the quality of the final solution. In general, a large number of subdomains, or alternately a small size for the coarse grid, results in a better, but more expensive solution since a large number of subdomain problems need to be solved and there is much more communication. On the other hand, choosing a small number of subdomains has the opposite effect.

Given a fine grid, that is, a discretized problem, it has been observed empirically [12] that there exists an optimal choice for the number of subdomains that minimizes the overall computational time. However, theoretical results [3] exist only for fairly simple scenarios

where the same solver is used for all the subdomain problems as well as the original problem, and the convergence rate is independent of the coarse grid size. For example, for a uni-processor setting on a structured n^d grid with fine grid scale h and a solver with complexity $O(n^\alpha)$, the optimal choice for the coarse grid size is given by

$$H_{opt} = \left(\frac{\alpha}{\alpha - d} \right)^{\frac{1}{\alpha-d}} h^{\frac{\alpha}{2\alpha-d}}.$$

Practical scientific computing problems, however, often involve fairly complex domains as in Fig. 1 and this requires taking into account a number of other considerations. Firstly, it is preferable to partition the domain into subdomains with regular geometry on which faster solvers can be deployed. Secondly, in a parallel setting, an ideal choice is to assign each subdomain to an individual processor to minimize communication costs. Such a choice might, however, lead to load imbalance in case the number of subdomains is less than that of the available processors or there is vast difference in the relative sizes of the subdomains. For a parallel scenario, there exist limited theoretical results. For example, for a structured n^d grid where all the subdomain solves are performed in parallel followed by an integration phase performed by one of the solvers, it can be shown that the optimal number of processors is $n^{d/2}$, but this analysis only considers load distribution and ignores the communication costs. For a real-world application, one needs to empirically determine an appropriate partitioning size that optimizes both the load balance as well as the communication costs.

Type of Partitioning

There are three main types of partitioning depending on the constraints imposed on subdomain interfaces. The first type is element-based partitioning, common in case of finite-element discretizations, where each element is solely assigned to a single subdomain. The second type is edge-based partitioning, more suitable for finite-volume methods, where each edge is solely assigned to a single subdomain so as to simplify the computation of fluxes across edges. The last type is vertex-based partitioning, which is the least restrictive and only requires each vertex to be assigned to a unique subdomain. The choice of partitioning type determines how the subdomain interface values are computed and can have a significant effect on the complexity and

convergence of the algorithm, for example, Schur complement method can be implemented much more efficiently for edge-based partitionings than the other two types [22]. For most problems, the appropriate type of partitioning is determined by the nature of the computational problem (PDE) and the discretization itself, but where there is flexibility, it is important to make a judicious choice in order to optimize the computation time for the coarse solve.

Level of Overlap

Overlap between the subdomains is another important characteristic that determines the solution approach. In particular, Schur complement-based approaches are applicable only in case of nonoverlapping subdomains whereas Schwarz alternating procedures (SAP) are applicable in case of overlapping subdomains. The appropriate level of overlap is often dependent on the geometry of the problem domain and the PDE to be solved. A decomposition into subdomains with regular geometry (e.g., circles, rectangles) is often preferable irrespective of the level of overlap. In general, SAP-based techniques are easier to implement in addition to providing close to optimal convergence rate and being more robust. However, extra computational effort is required for the regions of overlap and they are not suitable for handling discontinuities.

Partitioning Algorithms

For scientific computing tasks, the problem domain is invariably discretized making it natural to use a graph (or in certain cases hypergraph) representation. Decomposing the problem domain into subdomains, therefore, becomes a graph partitioning problem. For parallel computing with homogeneous processors, it is desirable to have a partitioning such that the resulting subdomains require comparable amounts of computational effort, that is, balanced load, and there are minimal communication costs for handling the subdomain interfaces. In other words, the subdomains need to be of similar size with few edges between them. In case of structured and quasi-uniformly refined grids, such a partitioning can be obtained with relative ease. However, for unstructured grids resulting from most real-world applications, it is an NP-complete problem [11]. A number of algorithms have been proposed in literature to address this problem and [11, 14, 15] provides

a detailed survey of these algorithms. Some of the key approaches are discussed below.

Geometric Approaches

Geometric partitioning approaches rely mainly on the physical mesh coordinates to partition the problem domain into subdomains corresponding to regions of space. Such algorithms are preferable for applications such as crash and particle simulations, where geometric proximity between grid points is much more critical than the graph connectivity.

Recursive bisection methods form an important class of geometric approaches that involve computing a separator that partitions a region into two subregions in a desired ratio (usually equal). This bisection process is performed recursively on the subregions until the desired number of partitions are obtained. The simplest of these techniques is the Recursive Coordinate Bisection (RCB) [1], which uses separators orthogonal to coordinate axes. A more effective technique is Recursive Inertial Bisection (RIB) [7] that employs separators orthogonal to principal inertial axes of the geometry. Over the last 2 decades, a number of extensions [4] as well as hybrid techniques [18] that combine RIB with local search methods such as the Kernighan-Lin's method, in order to simultaneously minimize the edge cut between the subdomains have been proposed.

Space-filling curve (SFC)-based partitioning approaches [23] are another class of geometric techniques. In these methods, a space-filling curve maps an n-dimensional space to a single dimension, which provides a linear ordering of all the mesh points. The mesh is then partitioned by dividing the ordered points into contiguous sets of desired sizes, which correspond to the subdomains.

The explicit use of geometric information and the incremental nature of the above techniques makes them highly beneficial for applications where geometric locality is critical and there are frequent changes in the geometry requiring dynamic load balancing. However, these techniques have limited applicability for computational problems where one needs to consider graph connectivity and communication costs.

Coordinate-Free Approaches

Coordinate-free partitioning approaches include spectral and graph-theory-based techniques that rely only on the connectivity structure of the problem domain.

These techniques are widely used in a number of scientific computing applications since they can provide better control of communication costs and do not require an embedding of the problem in a geometric space.

There currently exist a large number of such algorithms [11], but most of them have been derived from combinations of three basic techniques: (a) Recursive Graph Bisection, (b) Recursive Spectral Bisection, and (c) Greedy approach. Of these, the recursive graph bisection method [26] begins by finding a pseudo-peripheral vertex, (i.e., one of a pair of vertices on the longest graph geodesic), computing the graph distance to all the other vertices from this vertex. This distance is then used to sort the vertices, which are then split into two groups in the desired ratio (usually equal) such that one group is closer to the peripheral vertex and the other one is away, and this process is repeated recursively till the desired number of partitions are obtained. The second technique, that is, recursive spectral bisection method [21] makes use of the Fielder vector, which is defined as the eigenvector corresponding to the second lowest eigenvalue of the Laplacian matrix of the graph. The size of this vector is the same as the number of vertices in the graph and the difference between the Fielder coordinates is closely related to the graph distance between the corresponding vertices. The Fielder vector, thus, provides a linear ordering of the vertices, which can be used to divide the vertices into two groups and recursively deployed as in the case of SFC-based partitioning algorithms. The greedy approach typically uses a breadth first search and uses graph growing heuristics to create partitions [8].

There are also a number of fast multilevel techniques that are especially suitable for large problem domains. These techniques typically consist of three phases: coarsening, partitioning, and refining. In the first phase, a sequence of increasingly coarse graphs is constructed by collapsing vertices that are tightly connected to each other. The coarsened graph is then partitioned into the desired number of subdomains in the second phase. The graph is then progressively uncoarsened and refined using local search methods such as the Kernighan-Lin method [17], which improves the edge cut via a series of vertex-pair exchanges, in the third phase.

Dynamic Approaches

When the computational structure is relatively invariant through out a simulation, it is sufficient to consider a

static partitioning of the problem domain. However, for applications where the structure changes dynamically (e.g., in crash simulations with geometric deformations or in case of adaptive mesh refinement methods), it is critical to dynamically repartition the problem domain to ensure load balance. Such a repartitioning needs to account for not only the load distribution and communication costs, but also the costs of data redistribution across the partitionings [15]. Furthermore, in case of large problems, since it is expensive to load the entire mesh onto a single processor, these partitionings themselves need to be computed in parallel. Currently, there exist a number of dynamic partitioning algorithms such as [20, 24, 27] which can viewed as extensions of existing static partitioning algorithms that incorporate an additional objective term based on the data redistribution costs. These techniques often involve a clever remapping of subdomain labels to reduce the redistribution costs. In case of multilevel methods such as Locally-Matched Scratch-Remap (LMSR) [24], the coarsening phase is similar to that in the static case, the partitioning (i.e., repartitioning) phase leads to tentative new subdomains, and the final uncoarsening/refinement phase is adapted so as to additionally optimize the data redistribution costs.

PDE Solution Techniques

There are two main classes of approaches for solving PDEs over a partitioned problem domain [22] depending on the overlap, namely, (a) Schur complement-based approaches and (b) Schwarz alternating procedures, which we discuss below.

Schur Complement-Based Approaches

This class of approaches is based on the notion of a special matrix called the Schur complement, which enables one to solve for the subdomain interface values independent of the interior unknowns. The computation of the unknowns in the interior of each of the subdomains is then performed in parallel independently.

Consider a computational problem corresponding to a second-order elliptic PDE as in Example 1. As mentioned earlier, a discretization leads to the linear system $Au = f$, $u \in \Omega$. In case of nonoverlapping subdomains, that is, $\Omega_i \cap \Omega_j = \emptyset$, and edge-based partitioning on a finite difference mesh, this linear system can be

expressed as

$$\begin{bmatrix} B_1 & & D_1 \\ & B_2 & D_2 \\ & \ddots & \vdots \\ & & B_s & D_s \\ E_1 & E_2 & \cdots & E_s & C \end{bmatrix} \begin{bmatrix} u_1^{int} \\ u_2^{int} \\ \vdots \\ u_s^{int} \\ u^{bnd} \end{bmatrix} = \begin{bmatrix} f_1^{int} \\ f_2^{int} \\ \vdots \\ f_s^{int} \\ f^{bnd} \end{bmatrix}, \quad (1)$$

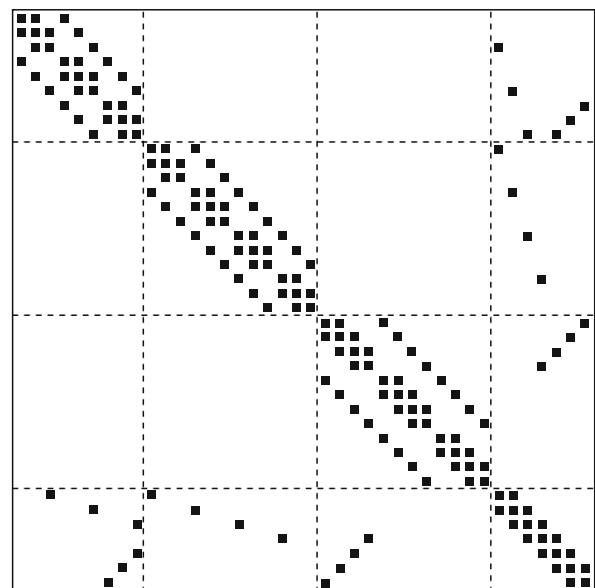
where each f_i^{int} denotes the vector of unknowns interior to subdomain Ω_i and u^{bnd} denotes the vector of interface unknowns.

Figure 2 shows the nonzero structure for such a linear system. This can be equivalently expressed in a block form as

$$\begin{pmatrix} B & D \\ E & C \end{pmatrix} \begin{pmatrix} u^{int} \\ u^{bnd} \end{pmatrix} = \begin{pmatrix} f^{int} \\ f^{bnd} \end{pmatrix}, \quad (2)$$

where B, D, E, f^{int} , and u^{int} are aggregations over the corresponding subdomain specific variables. From **Eq. 2**, by substituting for u^{int} , we obtain a reduced system

$$(C - EB^{-1}D)u^{bnd} = f^{bnd} - EB^{-1}f^{int}.$$



Domain Decomposition. Fig. 2 Nonzero structure of the resulting linear system

Algorithm 1 Schur complement method

1. Form the reduced system based on Schur complement matrix
2. Solve the reduced system for the interface unknowns
3. Solve the decoupled systems for interior unknowns by back-substituting the interface unknowns

The matrix $S = (C - EB^{-1}D)$ is called the Schur complement matrix. This matrix can be computed directly from the original coefficient matrix A given the partitioning and can be used to compute the interface values u^{bnd} . Back-substitution of u^{bnd} is then used to solve smaller decoupled linear systems $B_i u_i^{int} = f_i^{int} - D_i u^{bnd}$.

In the case of vertex-based and element-based partitionings, the coefficient matrix A exhibits a more complex block structure unlike in Eq. 1. However, even for these cases, one can construct a reduced system based on a Schur complement matrix assembled from local (subdomain specific) Schur complements and the interface coupling matrices, which can be used to solve for the interface unknowns.

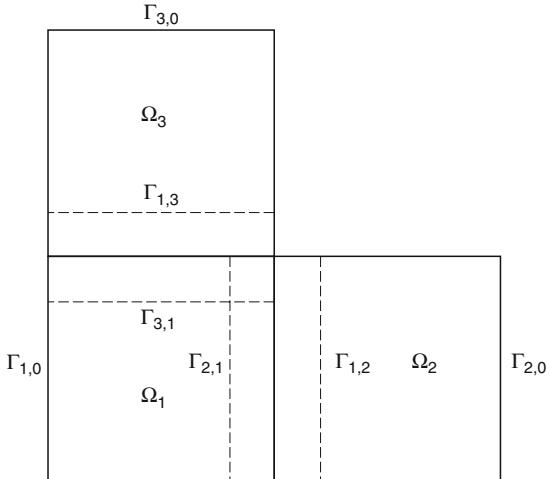
[Algorithm 1](#) describes the key steps for this class of techniques irrespective of the partitioning type. Depending on the choice of the methods (e.g., direct vs. iterative methods, exact vs. approximate solve) used for forming the Schur complement and solving the reduced systems, there are multiple possible variants.

Schwarz Alternating Procedures

Schwarz alternating procedures refer to techniques that involve alternating between overlapping subdomains and in each case, performing the corresponding subdomain solve assuming boundary conditions based on the most recent solution.

Consider the discretized elliptic PDE problem in Example 1 for the case where each subdomain Ω_i overlaps with its neighboring subdomains (as in Fig. 3). Let Γ_{ij} denote the boundary of Ω_i that is included in Ω_j . The original solution method proposed by Schwarz [25] in 1870 was a sequential method where each step consists of solving the restriction of $Au = f$ over the subdomain Ω_i assuming the boundary conditions over Γ_{ij} , $\forall j$.

Since the approach is iterative and sweeps through each subdomain multiple times, it suffices to perform



Domain Decomposition. Fig. 3 An L-shaped domain showing overlapping subdomains

an approximate subdomain solve or in the extreme case, one iteration of the subdomain solve. Thus, each step can be viewed as an additive update of u . This update step can be concretely specified using the notion of a *restriction operator*. Let $R_i \in \{0,1\}^{|\Omega_i| \times |\Omega|}$ be a binary matrix that denotes the restriction of Ω to Ω_i , that is, the j^{th} component of R_i is 1 iff the j^{th} unknown belongs to Ω_i . Given $\{R_i\}_{i=1}^s$, one can define the restriction of A to Ω_i as the matrix $A_i = R_i A R_i^T$ and the update is then given by

$$u = u + R_i^T A_i^{-1} R_i (f - Au).$$

Since the updates over each subdomain are applied sequentially, it is equivalent to a multiplicative composition and this procedure is referred to as Multiplicative Schwarz Alternation.

To make this approach amenable to parallelization, an alternate procedure Additive Schwarz Alternation [6] was proposed, which involves using the same residual ($f - Au$) to compute the incremental updates over each subdomain in a single sweep and the incremental updates are then aggregated to obtain the new solution. [Algorithm 2](#) shows the main steps for the basic additive Schwarz alternation and each of the δ_i computations in a single iteration are independent of each other and can be performed in parallel. [Algorithm 2](#) corresponds to the most basic version of additive Schwarz. Real applications, however

Algorithm 2 Additive Schwarz method

```

Choose an initial guess for  $u$ 
repeat
  for  $i = 1 \dots s$  do
    Compute  $\delta_i = R_i^T A_i^{-1} R_i (f - Au)$ 
     $u_{new} = u + \sum_{i=1}^s \delta_i$ 
  until convergence

```

employ variants of this approach with similar independent additive updates, but are based on more complex preconditioners [28].

Mapping to Parallel Processors

A natural way to map a domain decomposition algorithm to a parallel architecture is to assign each subdomain to a separate processor. In this case, the subdomain solves can be performed independently in parallel, but the coarse solve (handling of interface values) is nontrivial since the relevant interface data is scattered among all the processors. For example, in Schur complement approaches, solving the reduced system based on the Schur complement, in the general case, requires access to data from all the subdomains.

There are three common ways to address the above problem. The first is to keep the data in place and perform the solve in parallel with the necessary communication. Such an approach would require an intelligent grouping of the interface unknowns with minimal dependencies across groups, which can then be assigned to different subdomains so as to minimize the communication cost. The second option is to collect all the data on a single processor, perform the coarse solve, and broadcast the output while the third approach is to replicate the relevant data on all the processors and perform the coarse solve in parallel. There have been empirical studies [13] that evaluated the relative benefits of these approaches, but the appropriate choice typically depends heavily on the characteristics of the specific problem and the parallel architecture.

Advanced Methods

More recent work on domain decomposition has focused on efficient techniques for specific topics. These include advanced methods for graph partitioning and for solving PDEs.

In graph partitioning, researchers have developed hyper-graph algorithms [5, 16] that can be used for graphs with edges that involve more than two nodes. Such graphs arise naturally in many applications such as higher-order finite elements, and can also be constructed from regular graphs. Empirical evidence points to improvement in the efficiency of partitioning algorithms as well as in the quality of partitions. It has also been recognized for a while that large graph problems that require domain decomposition techniques often arise in a parallel setting, and as a result, work has focused on developing parallel partitioning methods. These methods are, in general, straightforward parallelization schemes, with additional care taken to ensure quality of the partitions. Another important area of work involves adaptive partitioning and refining [2, 24] for applications where the graph changes frequently during the simulation. A key component of these algorithms is the ability to repartition quickly and efficiently starting from an initial partition. The new partitioning must ensure good load balance without increasing communication requirement or introducing large overhead.

The domain decomposition step has implications on the numerical procedure used to solve the PDE. In particular, the rate of convergence of iterative solvers can be improved significantly by considering domain decomposition as a “preconditioning” step that reduces the number of iterations. In this regard, there has been some work on improving the Schur-complement approach by using Schwarz-type methods for the reduced linear system. This can be useful when the Schur complement is complex and large enough to require a parallel solver. Another approach that is motivated by solving PDEs in parallel is the Finite Element Tearing and Integration (FETI) [9, 10] method in which splitting of the domain into subdomains is accompanied by modification of the linear operator to achieve faster convergence of the iterative solver. After splitting the domain, one needs to enforce continuity across subdomain boundaries using Lagrange multipliers. The solution process determines these Lagrange multipliers via an iterative method that requires repeated subdomain solves that can be done in parallel.

Application Areas

Domain decomposition has been used in a wide variety of disciplines. These include aerospace, nuclear,

mechanics, electrical, materials, petroleum, and others. The benefits of the method are most apparent when used for applications that require large systems to be solved. In such cases, the solver phase can consume up to 90% or more of the total running time of the application. The domain decomposition concept can be applied in a fairly straightforward manner once the subdomains are identified. However, it may be important to fine-tune the parameters for each application in order to get the best performance.

Future Directions

The advent of multicore processors and the availability of large multiprocessor machines pose significant challenges to researchers and developers in the area of domain decomposition. Challenges that arise from large-scale compute capability involve the ability to use a large number of cores concurrently and efficiently. It will be much harder to achieve good partitioning and load balance on such platforms. The ability to perform asynchronous computation and to adapt to changing computational structure is equally important. In addition, it is imperative to build fault tolerance into the algorithmic structure to protect against inevitable architectural instabilities. One must also consider relaxing the numerical requirements on the computations in favor of obtaining an approximate but acceptable solution quickly. Another area of interest is the automatic tuning of parameters required by the algorithms to ensure good performance. It is also important to understand that domain decomposition is often used as a low-level computational tool in applications where the main goal is to optimize within a design space or to conduct sensitivity studies. A tighter integration of the domain decomposition step with other computational modules is necessary to realize the full benefit of this technique for an application.

Related Entries

► [Graph Partitioning](#)

Bibliography

1. Berger MJ, Bokhari SH (1987) A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans Comput* 36(5):570–580
2. Çatalyürek UV, Boman EG, Devine KD, Bozdag D, Heaphy RT, Riesen LA (2009) A repartitioning hypergraph model for dynamic load balancing. *J Parallel Distrib Comput* 69(8):711–724
3. Chan TF, Shao JP (1995) Parallel complexity of domain decomposition methods and optimal coarse grid size. *Parallel Comput* 21(7):1033–1049
4. Devine KD, Boman EG, Heaphy RT, Hendrickson BA, Teresco JD, Faik J, Flaherty JE, Gervasio LG (2005) New challenges in dynamic load balancing. *Appl Numer Math* 52(2–3):133–152
5. Devine KD, Boman EG, Heaphy RT, Bisseling RH, Çatalyürek UV (2006) Parallel hypergraph partitioning for scientific computing. In: *Proceedings of the 20th parallel and distributed processing symposium 2006*, Rhodes Island, Greece, 10 pp
6. Dryja M, Widlund OB (1994) Domain decomposition algorithms with small overlap. *SIAM J Sci Comput* 15(3):604–620
7. Farhat C, Lesoinne M (1993) Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. *Int J Numer Methods Eng* 36(5):745–764
8. Farhat C (1988) A simple and efficient automatic fem domain decomposer. *Comput Struct* 28(5):579–602
9. Farhat C, Pierson K, Lesoinne M (2000) The second generation feti methods and their application to the parallel solution of large-scale linear and geometrically non-linear structural analysis problems. *Comput Methods Appl Mech Eng* 184(2–4): 333–374
10. Farhat C, Roux F-X (1991) A method of finite element tearing and interconnecting and its parallel solution algorithm. *Int J Numer Meth Eng* 32:1205–1227
11. Fjällström P-O (1998) Algorithms for graph partitioning: a survey, vol 3, part 10. Linköping University Electronic Press, Sweden
12. Gropp WD, Keyes DE (1989) Domain decomposition on parallel computers. *IMPACT Comput Sci Eng* 1(4):420–439
13. Gropp WD, Smith BF (1993) Experiences with domain decomposition in three dimensions: overlapping schwarz methods. In: *Proceedings of the 6th international symposium on domain decomposition*, AMS, Providence, 1993, pp 323–333
14. Hendrickson B, Kolda TG (2000) Graph partitioning models for parallel computing. *Parallel Comput* 26(12):1519–1534
15. Karypis G, Kumar V (1996) Parallel multilevel k-way partitioning scheme for irregular graphs. In: *Proceedings of the 8th ACM/IEEE conference on supercomputing (CDROM) (Super computing '96)*, Article 35, Pittsburgh, PA, USA
16. Karypis G, Kumar V (1999) Multilevel -way hypergraph partitioning. In: *Proceedings of the 36th design automation conference*, New Orleans, LA, USA, pp 343–348
17. Kernighan BW, Lin S (1970) An efficient heuristic procedure for partitioning graphs. *Bell Syst Tech J* 49(1):291–307
18. Leland R, Hendrickson B (1994) An empirical study of static load balancing. In: *Proceedings of scalable high performance computing conference*, pp 682–685
19. Neuman SP (1977) Theoretical derivation of Darcy's law. *Acta Mech* 25(3):153–170
20. Oliker L, Biswas R (1998) Plum: parallel load balancing for adaptive unstructured meshes. *J Parallel Distrib Comput* 52(2):150–177
21. Pothen A, Simon HD, Liou K (1990) Partitioning sparse matrices with eigenvectors of graphs. *SIAM J Matrix Anal Appl* 11(3): 430–452
22. Saad Y (2003) *Iterative methods for sparse linear systems*, 2nd edn. SIAM, Philadelphia, PA

23. Schamberger S, Wierum J (2005) Partitioning finite element meshes using space-filling curves. Future Gen Comp Syst 21(5):759–766
24. Schloegl K, Karypis G, Kumar V (2001) Wavefront diffusion and LMSR: Algorithms for dynamic repartitioning of adaptive meshes. IEEE Transactions on Parallel and Distributed Systems 12(5):451–466
25. Schwarz HA (1870) Über einen Grenzübergang durch alternierendes Verfahren. Vierteljahrsschrift der Naturforschenden Gesellschaft in Zürich, 15:272–286
26. Simon HD (1991) Partitioning of unstructured problems for parallel processing. Comput Syst Eng 2(2–3):135–148
27. Sohn A, Simon H (1994) Jove: a dynamic load balancing framework for adaptive computations on an sp-2 distributed multiprocessor. Technical report, CIS, New Jersey Institute of Technology, NJ
28. Sun K, Zhou Q, Mohanram K, Sorensen DC (2007) Parallel domain decomposition for simulation of large-scale power grids. In: ICCAD, pp 54–59
29. Zhang Y, Bajaj C (2004) Finite element meshing for cardiac analysis. Technical Report 04–26, ICES, University of Texas at Austin, TX

DPJ

► [Deterministic Parallel Java](#)

DR

► [Dynamic Logical Partitioning for POWER Systems](#)

Dynamic Logical Partitioning for POWER Systems

JOEFON JANN

T. J. Watson Research Center, IBM Corp., Yorktown Heights, NY, USA

Synonyms

[DLPAR](#); [DR](#); [Dynamic LPAR](#); [Dynamic reconfiguration](#)

Definition

Dynamic Logical Partitioning (DLPAR) is the nondisruptive reconfiguration of the real or virtual resources made available to the operating system (OS) running in a logical partition (LPAR). Nondisruptive means that no OS reboot is needed.

Discussion

Introduction

A Logical PARtition (LPAR) is a subset of hardware resources of a computer upon which an operating system instance can run. One can define one or more LPARs in a physical server, and an OS instance running in an LPAR has no access and visibility to the other OS instances running in other LPARs of the same server. In October 2002, IBM introduced the DLPAR technology on POWER4-based servers with the general availability of AIX 5.2. This DLPAR technology enables resources such as memory, processors, disks, network adaptors, and CD/DVD/tape drives to be moved nondisruptively (i.e., without requiring a reboot of the source and target operating systems) between LPARs in the same physical server. In simple terms, one can perform the following basic nondisruptive dynamic operations with DLPAR:

- Remove a unit of resource from an LPAR.
- Add a unit of resource to an LPAR.
- Move a unit of resource from one LPAR to another in the same physical server.

For POWER4 systems, the granularity/unit of resource for a DLPAR operation was 1 CPU for processors, 1 Logical Memory Block (LMB) of size 256MB for memory, and 1 IO-slot for I/O (disks, network, and media) adapters. Those units of processor, memory, and I/O-slots that are not assigned to any LPAR reside in a “free pool” of the physical server, and can be allocated to a new or existing LPARs via DLPAR operations later on. Similarly, when a unit of resource is DLPAR-removed from an LPAR, it goes into the free pool.

The fundamentals of DLPAR are described in [1].

AIX has supported DLPAR removal of memory since 10/2002; however, DLPAR removal of memory was not available for Linux on POWER until 2009, with the availability of SUSE SLES 11. On POWER5 and follow-on POWER systems, the minimum size of an LMB (unit of DLPAR removal or add) can be as small as 16MB. This minimum size is a function of the total amount of physical memory in the physical server.

Since the availability of POWER5, all POWER servers are partitioned into one or more LPARs. These LPARs are defined via the POWER HYPervisor firmware (PHYP). An LPAR can be a dedicated-processor LPAR or a Shared-Processor LPAR (SPLPAR). A dedicated-processor LPAR is an LPAR which contains

an integral number of CPUs. SPLPAR was introduced with AIX 5.3 on POWER5 systems in 6/2004. One can define a pool of CPUs to be shared by a set of LPARs, which will be called shared-processor LPARs (SPLPARs). The PHYP controls the time-slicing and allocation of CPU resources across the SPLPARs sharing the pool. An SPLPAR has the following *attributes*:

- (a) Minimum, assigned, and maximum number of *Virtual Processors (VPs)*.

The number of VPs is the degree of processor-concurrency made available to the OS instance.

- (b) Minimum, assigned, and maximum number of *processor-units of Capacity*.

Capacity is the amount of cumulative real-CPU-equivalent resources allocated to the OS instance in a reasonable interval of time (e.g., 10 ms).

- (c) *Capped or uncapped mode*. A capped SPLPAR cannot use excess CPU resources in the pool beyond its maximum capacity.
- (d) If uncapped, an SPLPAR has an *uncapped-weight value* (currently, an integer between 0 and 255), which is used for prioritizing the SPLPARs in a pool.

The DLPAR technology was then enhanced to be able to dynamically move units of a SPLPAR's CPU-resources nondisruptively from one SPLPAR to another in the same SPLPAR pool. More precisely, DLPAR technology can nondisruptively and dynamically

- Remove or add an integral number of Virtual Processors from or to an SPLPAR, or move such from one SPLPAR to another
- Remove or add Capacity (in processor-units) from or to an SPLPAR, or move capacity from one SPLPAR to another
- Change the processor Mode of an SPLPAR to Capped or Uncapped
- Change the Weight of an Uncapped SPLPAR (between a value of 0 and 255)

DLPAR Technology

DLPAR Removal of a CPU

DLPAR enables the dynamic removal of a CPU from a running OS instance in one LPAR, without requiring a reboot of the OS instance.

At a very high level, the following sequence of operations initiated from the OS is used to accomplish a DLPAR removal of a CPU:

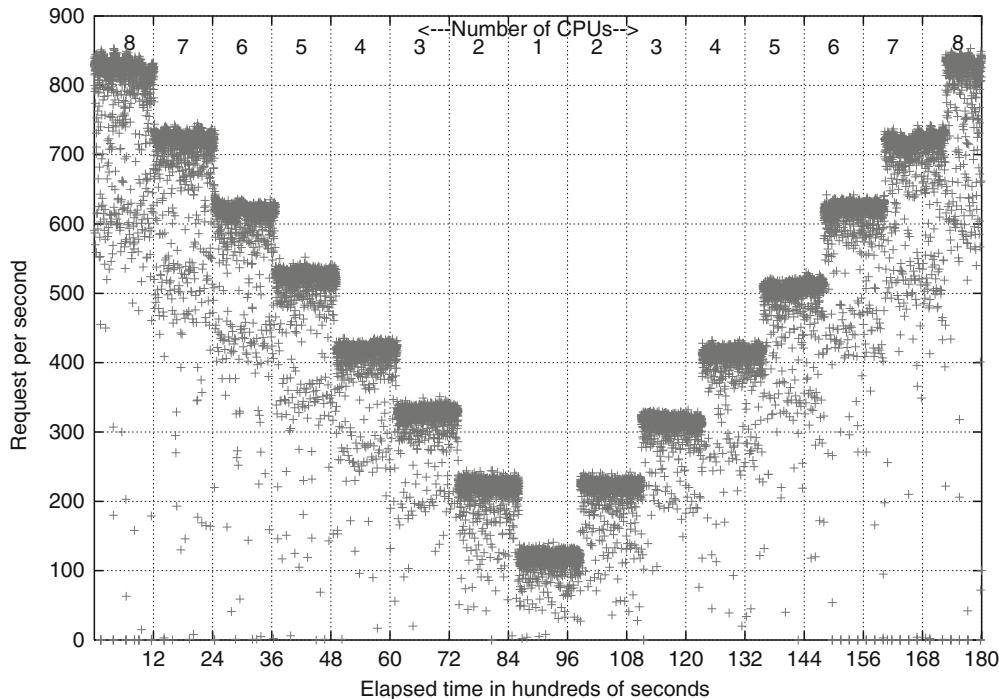
1. Notify all applications and kernel extensions that have registered to be notified of CPU DLPAR events, so that they may remove dependencies on the CPU to be removed. This typically involves unbinding their threads that are bound to the CPU being removed.
2. Migrate threads that are bound to the CPU being removed to another CPU in the same LPAR.
3. Reroute existing and future hardware-interrupts that are directed to the CPU being removed. This involves changing the interrupt controller data structures.
4. Migrate the timers and threads from the CPU being removed to another CPU in the same LPAR.
5. Notify the hypervisor/firmware to complete the removal task.

DLPAR Addition of a CPU

Dynamic addition of a CPU involves the following tasks initiated from the OS:

1. Create a process (waitproc) to run an idle loop on the incoming CPU, before it starts to do real work.
2. Set up various hardware registers (e.g., GPR1 for the kernel stack, GPR2 for the kernel Table Of Contents) of the incoming CPU.
3. Allocate and/or initialize the processor-specific kernel data structures (e.g., runqueue, per-processor data area, interrupt stack) for the incoming CPU.
4. Add support for the incoming CPU to the interrupt subsystem.
5. Notify the DLPAR-registered applications and kernel extensions that a new CPU has been added.

Dynamic addition and removal of processors to and from an LPAR allow it to adapt to varying workloads. Figure 1 shows that the throughput (in requests-per-second) of the WebSphere Trade3 benchmark scales almost linearly with the number of CPUs in a dedicated-processor-LPAR, as CPUs were removed and added to the LPAR with DLPAR operations [2].



Dynamic Logical Partitioning for POWER Systems. Fig. 1 WebSphere Trade3 benchmark: throughput (in requests-per-second) as a function of the number of CPUs (the *upper x-axis labels*), and as a function of elapsed time (the *lower x-axis labels*)

DLPAR Addition of a Memory Block (LMB)

When a DLPAR memory-add request arrives at the OS, the latter has to perform two tasks:

1. Allocate and initialize software page descriptors that will hold metadata for the incoming memory.
2. Distribute the incoming memory among the framesets of a mempool (mempools are described two paragraphs below). The challenges encountered in implementing these two tasks in AIX and how they were resolved are described as follows.

Prior to implementing DLPAR, software page descriptors could be accessed in translation-off mode (where virtual address is used as real address) while trying to reload a page mapping into the hardware page table. If page descriptors are allowed to be accessed in translation-off mode, then the memory allocated for the incoming new descriptors has to be physically contiguous to the memory for the existing descriptors; which implies that memory has to be reserved at boot-time for enough descriptors for the maximal amount

of memory that the OS instance can grow into. This reservation can incur much memory wastage and performance degradation, particularly if not utilized. We avoided this wastage by changing the AIX kernel so that software page descriptor data structures are always accessed in translation-on mode.

The challenge of distributing the incoming memory across different page replacement daemons, so that each daemon handles a roughly equal load was resolved as follows:

In AIX, memory is hierarchically represented by the data structures vmpool, mempool, and frameset. A vmpool represents an affinity domain of memory. A vmpool is divided into multiple mempools, and each mempool is managed by a single page replacement LRU (least recently used) daemon. Each mempool is further subdivided into one or more framesets that contain the free-frame lists, so as to improve the scalability of free-frame allocators. When new memory is added to AIX, the vmpool that it should belong to is defined by the physical location of the memory chip. Within that vmpool, we wanted to distribute the

memory across all the available mempools to balance the load on page replacement daemons. However, the kernel assumed that a mempool consisted of physically contiguous memory. Thus, to be able to break up the new memory (LMB) into several parts and distribute them across different mempools, the kernel was modified to allow mempools to be made up of discontiguous sections of memory.

DLPAR Removal of a Memory Block (LMB)

This is by far the hardest-to-implement of all the DLPAR operations, particularly when there are ongoing DMA operations in the OS instance. For the purpose of DLPAR memory removal, we classify the memory page-frames in AIX into five categories: unused, page-able, pinned, DMA-mapped, and translation-off memory. The approach taken to remove a page-frame (4,096 bytes) in each of these categories is as follows:

1. A page-frame containing a free/unused page is simply removed from its free-list.
2. A page-frame containing a page-able page can be made to either page out its contents to disk or to migrate its contents to a different free page-frame.
3. A page-frame containing a pinned page will have its contents migrated to a different page-frame; also the page-fault reload handler had to be made to spin during the migration.
4. A page-frame containing a DMA-mapped page cannot be removed nor have its contents migrated until all the accesses to the page are blocked. Here, the term “DMA-mapped page-frame” is used generically to mean a page-frame whose physical address is subject to read or write by an external (to kernel) entity such as a DMA engine. The contents of a DMA-mapped page-frame are migrated to a different page-frame with a new hypervisor call (*h_migrate_dma*) that will selectively suspend the bus traffic while it is modifying the TCEs (translation control entries) in system memory used by the bus unit controller for DMA accesses.

Other page-frames whose physical addresses are exposed to external entities are handled by invoking preregistered DLPAR callback routines and then waiting for completion of the removal of their dependencies on the page.

5. A page-frame containing translation-off pages will not be removed by DLPAR in AIX 5.2. Fortunately,

there is just a small amount of these page-frames, and they are usually collocated in low memory.

The design and implementation of DLPAR memory removal has manifested itself as three modular functions, such that one can mix and match these functions in several possible ways, adapting to the state of the system at the time of memory removal, thus achieving the desired end result with the most optimal path. These three modular functions perform, respectively, the following three tasks on the memory (LMB) being removed:

- (a) Remove its free and clean pages from the regular use of VMM (Virtual Memory Manager)
- (b) Page out its page-able dirty pages
- (c) Migrate the contents of each remaining page-frame in the LMB to a free page-frame outside the LMB

Memory removal can be implemented with any one of the sequences: abc, ac, bc, or just c. For example, the decision to either invoke page-out (task b) or to migrate (task c) all the pages depends on the load in the LPAR at that particular time. If the LMB being removed contains a lot of dirty pages that belong to highly active threads, then it does not make sense to invoke task b, because these pages will be paged back in almost immediately, hence negatively impacting the efficiency of the system.

As a second example, if there are not enough free frames in other LMBs to migrate the pages to, then the memory removal procedure can invoke task b before invoking task c, so that there will be far fewer pages left that need to be migrated in task c.

DLPAR of I/O Slots

The methods to dynamically configure or unconfigure a device have been introduced as early as AIX version 3. The changes required in the kernel design for the DLPAR of I/O slots were not in the same scale as those for the DLPAR of processors, and particularly as those for the DLPAR of memory. The reason is that the onus of configuring or unconfiguring a device lies with the device driver software, which operates in the kernel extension environment. The kernel just acts as a provider of serialization mechanisms for devices accessing common resources, and as an intermediary between the applications and the device drivers.

Values and Uses of DLPAR

DLPAR is the foundation of many advanced technologies for ensuring scalability, reliability, and full utilization of the resources in POWER Systems servers. It opens up a whole set of possibilities for great flexibility in dealing with dynamically changing workload demands, server consolidations of workloads from different time zones, and high availability for applications.

Some Obvious Values Enabled by DLPAR

- Ability to dynamically move processors and memory from a *test LPAR* to a *production LPAR* in periods of peak production workload demand, then move them back as demand decreases.
- Ability to programmatically and dynamically move processors and memory from less loaded LPARs to busy LPARs, whenever the need arises [14]. This is particularly useful for economically providing resources to LPARs that service *workloads in different time zones* or even different continents.
- Ability to dynamically *move an infrequently used I/O device* between LPARs, such as a CD/DVD/tape drives for installations, applying updates, and for backups. Occasionally, one may also want to move Ethernet adaptors and disk drives dynamically from 1 LPAR to another.
- Ability to dynamically release a set of processor, memory, and I/O resources into the “free pool,” so that a *new LPAR can be created* using the resources in the free pool.
- Providing high availability for the applications in an LPAR. For example, you can configure a set of minimal LPARs on a single system to act as “failover” backup LPARs to a set of primary LPARs, and also keep some set of resources free. If one of the associated primaries fails, then its backup LPAR can pick up the workload, and resources can be dynamically added to the backup LPAR as needed.

Some Non-Obvious Values Enabled by DLPAR

- *Dynamic CPU Guard*, which is the automatic and graceful de-configuration of a processor which has intermittent errors.
- *Dynamic CPU Sparing*, which is the Dynamic CPU Guard feature enhanced with automatic enablement of a spare CPU to replace a defective CPU.

- *CUoD* (Capacity Upgrade on Demand), which allows a customer to activate preinstalled but inactive and unpaid for processors as resource needs arise, as soon as IBM is notified and enables the corresponding license keys. Many enterprises with seasonally varying business activities extensively utilize this feature in their IT infrastructure.
- *Hot repair of PCI devices*, and possibly future hot repair of MCM (Multi-Chip Modules).
- *The failover of an LPAR to another LPAR in another physical server, used with the IBM HACMP LPP* (Licensed Program Product). This is detailed in the 60-page IBM Redpaper titled “HACMP v5.3, Dynamic LPAR, and Virtualization” [8]. HACMP is acronym for High Availability Cluster Multiprocessing. It is IBM’s solution for high-availability clusters on the AIX and Linux POWER platforms.

DLPAR-Safe, DLPAR-Aware, and DLPAR-Friendly Programs/Applications

A *DLPAR-safe program* is one that does not fail as a result of DLPAR operations. Its performance may suffer when resources are taken away, and performance may not scale up when new resources are added. However, by default, most applications are DLPAR-safe, and only a few applications are expected to be impacted by DLPAR.

A *DLPAR-aware program* is one that has code that is designed to adjust its use of system resources commensurate with the actual resources of the LPAR, which is expected to vary over time. This may be accomplished in two ways.

The first way is by regularly polling the system to discover changes in its LPAR resources.

The second way is by registering a set of DLPAR code that is designed to be executed in the context of a DLPAR operation. At a minimum, DLPAR-aware programs are designed to avoid introducing conditions that may cause DLPAR operations to fail. They are not necessarily concerned with the impact that DLPAR may have on their performance.

A *DLPAR-friendly program* is a DLPAR-aware application or middleware that automatically tunes itself in response to the changing LPAR resources. AIX provides a set of DLPAR scripts and APIs for applications to dynamically resize. Notable vendor applications

that are DLPAR-friendly include IBM DB2 [4], Lotus Domino [5], Oracle [6], etc.

More information about these scripts and APIs can be found in chapter 18 (Dynamic Logical Partitioning) of the online IBM manual “*AIX Version 6.1 General Programming Concepts*” [7].

To maintain expected levels of application performance when memory is removed, buffers may need to be drained and resized. Similarly when memory is added, buffers may need to be dynamically increased to gain performance.

Similar treatment needs to be applied to threads, whose numbers, at least in theory, need to be dynamically adjusted to the changes in the number of online CPUs. However, thread-based adjustments are not necessarily limited to CPU-based decisions; for example, the best way to reduce memory consumption in Java programs may be to reduce the number of threads, since this should reduce the number of active objects that need to be preserved by the Java Virtual Machine’s garbage collector.

Future Directions

Future uses of DLPAR are already being explored; among these are:

- Power/energy conservation for the server by dynamically DLPAR-remove of unused CPUs and memory
- Dynamic hot repair of server components, even MCM (Multi-Chip Modules)

Related Entries

- ▶ [IBM Power Architecture](#)
- ▶ [Multi-Threaded Processors](#)

Bibliography

1. Jann J, Browning L, Burugula RS (2003) Dynamic reconfiguration: basic building blocks for autonomic computing in IBM pSeries servers. *IBM Syst J (Spec Issue Auton Comput)* 42(1)
2. Jann J, Dubey N, Pattnaik P, Burugula RS (2004) Dynamic reconfiguration of CPU and WebSphere on IBM pSeries servers. *Softw Pract Exp J (SPE Issue)* 34(13):1225–1255
3. Lynch J Dynamic LPAR – the way to the future. <http://www.ibmsystemsmag.com/aix/junejuly07/administrator/l5824p1.aspx>

4. Shah P (2005) DB2 and dynamic logical partitioning. http://www.ibm.com/developerworks/eserver/articles/db2_dlpars.html
5. Bassemir R, Faurot G (2003) Lotus Domino and AIX DLPAR. <http://www.ibm.com/developerworks/aix/library/au-DominoandDLPAver1.html>
6. Shanmugam R (2006) Oracle database and Oracle RAC 10gR2 on IBM AIX. <http://www.ibm.com/servers/enable/site/peducation/wp/a696/a696.pdf>
7. IBM manual “*AIX Version 7.1: General Programming Concepts: Writing and Debugging Programs*” SC23-6718-00, 09/2010 pp 465–480 <http://publib.boulder.ibm.com/infocenter/aix/v7r1/topic/com.ibm.aix.genprogc/doc/genprogc/genprogc.pdf>
8. Quintero D, Bodily S, Pothier P, Lasca O (2006) HACMP v5.3, dynamic LPAR, and virtualization. IBM Redpaper, <http://www.redbooks.ibm.com/redpapers/pdfs/redp4027.pdf>
9. Bodily S, Killeen R, Rosca L (2009) PowerHA for AIX cookbook. IBM Redbook, <http://www.redbooks.ibm.com/redbooks/pdfs/sg247739.pdf>
10. Matsubara K, Guérin N, Reimbold S, Niijima T (2003) The complete partitioning guide for IBM eServer pSeries servers. IBM Redbook, SG24-7039-01, <http://portal.acm.org/citation.cfm?id=995991> ISBN:0738499447
11. Irving N, Jenner M, Kortesniemi A (2005) Partitioning implementation for IBM eServer p5 servers. IBM Redbook, SG24-7039-02, <http://www.redbooks.ibm.com/redbooks/pdfs/sg247039.pdf>
12. Hales C, Milsted C, Stadler O, Vågmo M (2008) PowerVM virtualization on IBM system p: introduction and configuration, 4th edn. IBM Redbook, <http://www.redbooks.ibm.com/redbooks/pdfs/sg247940.pdf>
13. Dimmer I, Haug V, Huché T, Singh AK, Vågmo M, Venkataraman AK (2009) Ch 6: DLPAR. In: PowerVM virtualization managing & monitoring. IBM RedBook, <http://www.redbooks.ibm.com/redbooks/pdfs/sg247590.pdf>
14. Jann J, Burugula RS, Dubey N IBM DLPAR tool set for pSeries for P4 & P5 systems. <http://www.alphaworks.ibm.com/tech/dlpar> (Contact joefon@us.ibm.com)
15. Wikipedia entry on DLPAR: <http://en.wikipedia.org/wiki/DLPAR>

Dynamic LPAR

- ▶ [Dynamic Logical Partitioning for POWER Systems](#)

Dynamic Reconfiguration

- ▶ [Dynamic Logical Partitioning for POWER Systems](#)

E

Earth Simulator

KEN'ICHI ITAKURA

Japan Agency for Marine-Earth Science and Technology
(JAMSTEC), Yokohama, Japan

Synonyms

ES

Definition

The Earth Simulator is a highly parallel vector supercomputer system consisting of 640 processor nodes and an interconnection network.

Discussion

The Earth Simulator, which was developed, as a national project, by three governmental agencies, the National Space Development Agency of Japan (NASDA), the Japan Atomic Energy Research Institute (JAERI), and Japan Marine Science and Technology Center (JAMSTEC), was led by Dr. Hajime Miyoshi. The ES is housed in the Earth Simulator Building (approximately 50 m × 65 m × 17 m). The fabrication and installation of the ES at the Earth Simulator Center of JAMSTEC by NEC was completed at the end of February in 2002. It was first on the Top500 list for two and a half years starting in June 2002 ([Fig. 1](#)).

Hardware

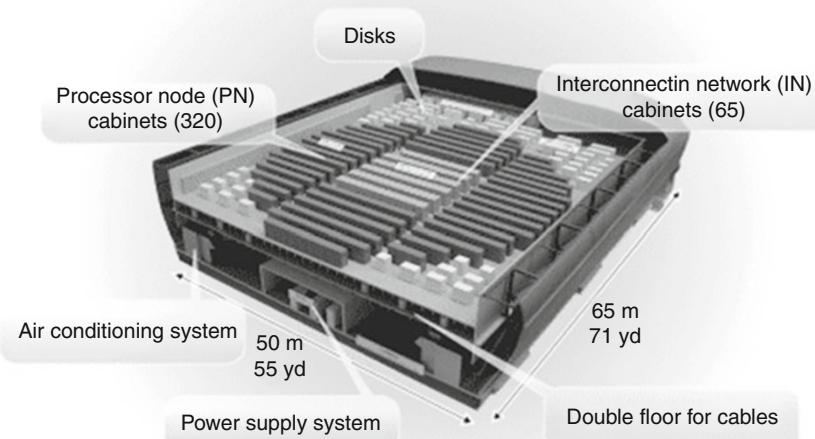
The ES is a highly parallel vector supercomputer system of the distributed-memory type, and consisted of 640 processor nodes (PNs) connected by 640×640 single-stage crossbar switches. Each PN is a system with a shared memory, consisting of eight vector-type arithmetic processors (APs), a 16-GB main memory system (MS), a remote access control unit (RCU), and an I/O processor. The peak performance of each AP is 8 Gflops.

The ES as a whole thus consists of 5,120 APs with 10 TB of main memory and the theoretical performance of 40 Tflops.

Each AP consists of a four-way super-scalar unit (SU), a vector unit (VU), and main memory access control unit on a single LSI chip. The AP operates at a clock frequency of 500 MHz with some circuits operating at 1 GHz. Each SU is a super-scalar processor with 64 KB instruction caches, 64 KB data caches, and 128 general-purpose scalar registers. Branch prediction, data prefetching, and out-of-order instruction execution are all employed. Each VU has 72 vector registers, each of which has 256 vector elements, along with eight sets of six different types of vector pipelines: addition/shifting, multiplication, division, logical operations, masking, and load/store. The same type of vector pipelines works together by a single vector instruction and pipelines of different types can operate concurrently. The VU and SU support the IEEE 754 floating-point data format. This LSI chip technology is 150 nm CMOS and eight layers copper interconnection. The die size is 20.79 mm × 20.79 mm. It has 60 million transistors and 5,185 pins. The maximum power consumption is 140 W.

The overall MS is divided into 2,048 banks and the sequence of bank numbers corresponds to increasing addresses of locations in memory. Therefore, the peak throughput is obtained by accessing contiguous data which are assigned to locations in increasing order of memory address.

The RCU is directly connected to the crossbar switches and controls internode data communications at 12.3 GB/s bidirectional transfer rate for both sending and receiving data. Thus the total bandwidth of internode network is about 8 TB/s. Several data-transfer modes, including access to three-dimensional (3D) sub-arrays and indirect access modes, are realized in hardware. In an operation that involves access to the data of a sub-array, the data is moved from one PN to



Earth Simulator. Fig. 1 Bird's-eye view of the earth simulator system

another in a single hardware operation, and relatively little time is consumed for this processing.

The Interconnection Network (IN) cabinet has two switches of 640×640 . The signal from PN changes from Serial to Parallel and is inputted into a switching circuit. The signal which came out from the switching circuit is performed Parallel/Serial conversion. Then, it is sent to PN cabinet. The number of Cables which connects PN cabinet and IN cabinet is $640 \times 130 = 83,200$, and the total extension is 2,400 km.

MDPS

In October 2003, Mass Data Processing System (MDPS) was installed as a massively data storage system which renews the tape library system. It consists of four file service processors (FSPs), 250 TB hard disk drives, and a currently used 1.5 PB cartridge tapes library (CTL).

MDPS was adopted aiming to improve data transfer throughput and accessibility.

1. The transfer speed of saving and extracting the data between the ES and the storage became two to five-times faster. This improvement can be realized by expansion of transfer cable ability and replacement of a tape archive with disks.

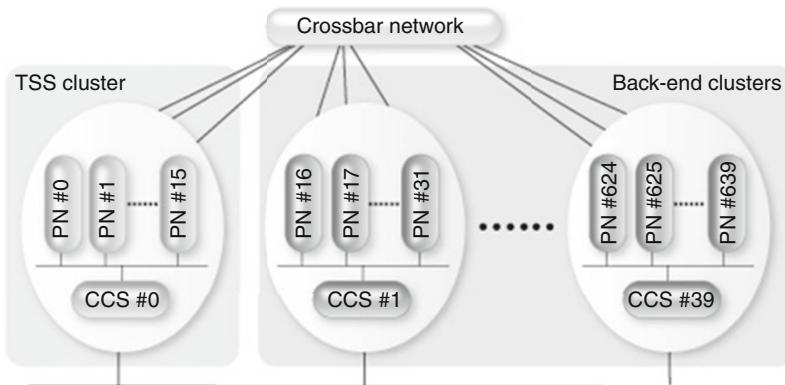
2. User view is a very large disk and usual UNIX commands and tools can access the MDPS data on the login server. Data I/O procedure becomes easy.
3. MDPS enables users to access the results computed by ES remotely, because it can transfer the data to a dedicated server out of ES LAN (ES-Network FTP Server).

Operating System

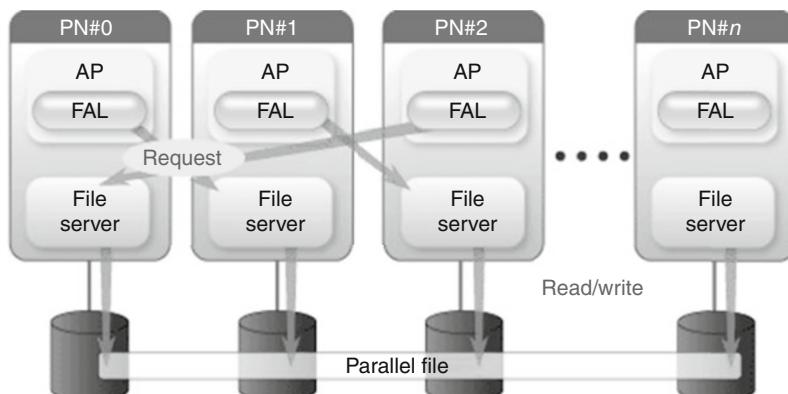
The operating system running on ES is an enhanced version of NEC's UNIX-based OS called "SUPER-UX" that is developed for NEC's SX Series supercomputers. To support ultra-scale scientific computations, SUPER-UX was enhanced mainly in the following two points:

1. Extending scalability
2. Providing special features for ES

Extending scalability up to the whole system (640 PNs) is the major requirement for the OS of ES. All functions of the OS, such as process management, memory management, file management, etc., are fully optimized to fulfill the requirement. For example, any of the OS tasks costing order n , such as scattering data in sequence over all PNs, is replaced with the equivalent one costing order $\log n$, such as binary-tree copy, if possible, where n is the number of PNs.



Earth Simulator. Fig. 2 Super cluster system of ES: a hierarchical management system is introduced to control the ES. Every 16 nodes are collected as a cluster system and therefore, there are 40 sets of cluster in total. A set of cluster is called an “S-cluster” which is dedicated for interactive processing and small-scale batch jobs. A job within one node can be processed on the S-cluster. The other sets of cluster is called “L-cluster” which are for medium-scale and large-scale batch jobs. Parallel processing jobs on several nodes are executed on some sets of cluster. Each cluster has a cluster control station (CCS) which monitors the state of the nodes and controls electricity of the nodes belonged to the cluster. A super cluster control station (SCCS) plays an important role of integration and coordination of all the CCS operations.



Earth Simulator. Fig. 3 Parallel file system (PFS): A parallel file, i.e., file on PFS is striped and stored cyclically in the specified blocking size into the disk of each PN. When a program accesses to the file, the File Access Library (FAL) sends a request for I/O via IN to the File Server on the node that owns the data to be accessed

On the other hand, the OS provides some special features which aim for efficient use or administration of such a large system. The features include internode high-speed communication via IN, global address space among PNs, super cluster system (Fig. 2), batch job environment, etc.

Parallel File System

If a large parallel job running on 640 PNs reads from/writes to one disk installed in a PN, each PN accesses to the disk in sequence, and performance degrades terribly. Although local I/O in which each PN

reads from or writes to its own disk solves the problem, it is a very hard work to manage such a large number of partial files. Therefore, parallel I/O is greatly demanded in ES from the point of view of both performance and usability. The parallel file system (PFS) provides the parallel I/O features to ES (Fig. 3). It enables handling multiple files, which are located on separate disks of multiple PNs, as logically one large file. Each process of a parallel program can read/write distributed data from/to the parallel file concurrently with one I/O statement to achieve high performance and usability of I/O.

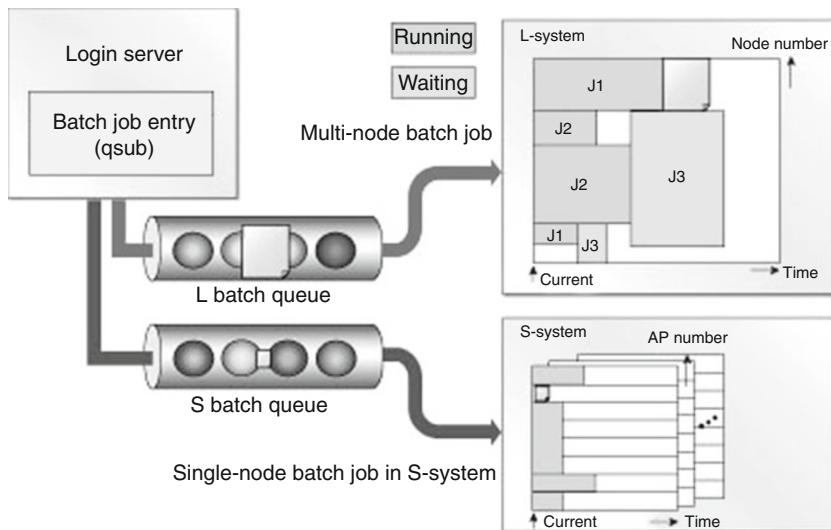
Job Scheduling

ES is basically a batch-job system. Network Queuing System II (NQSII) is introduced to manage the batch job (Fig. 4).

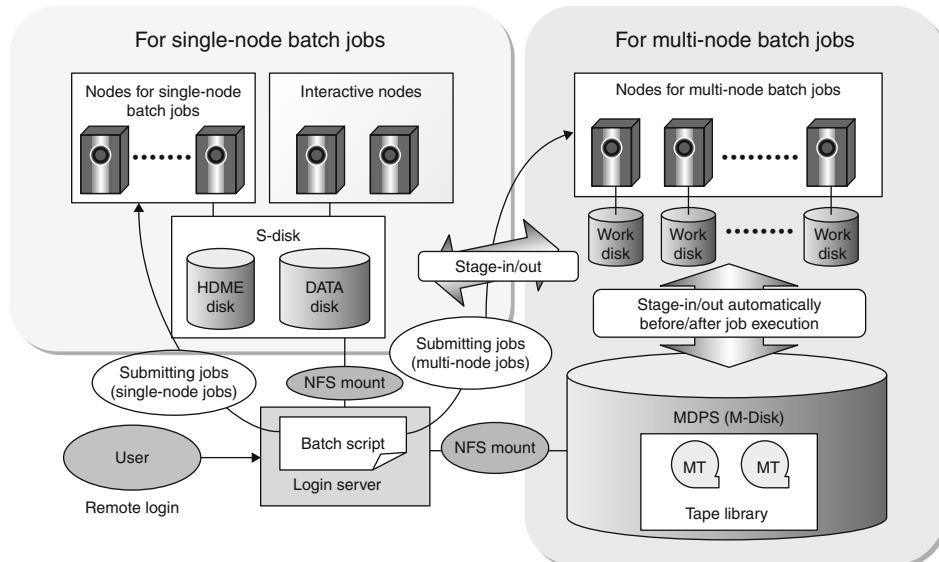
There are two type queues. One is L batch queue and the other is S batch queue. S batch queue is aimed at being used for a pre-run or a post-run for large-scale

batch jobs (making initial data, processing results of a simulation, and other processes), and L batch queue is for a production run. Users choose an appropriate queue for their jobs.

S batch queue is designed for single-node batch jobs. In S batch queue, Enhanced Resource Scheduler (ERSII) is used as a scheduler and does a job scheduling based



Earth Simulator. Fig. 4 Shows the queue configuration of ES



Earth Simulator. Fig. 5 Job execution. The user writes a batch script and submits the batch script to ES. The node scheduling, the file staging, and other processing are automatically processed by the system

on CPU time. On the other hand, L batch queue is for multi-node batch jobs. In this queue, the customized scheduler for ES is used as the scheduler. We have been developing this scheduler with following strategies:

1. The nodes allocated to a batch job are used exclusively for that batch job.
2. The batch job is scheduled based on elapsed time instead of CPU time.

Strategy (1) enables to estimate the job termination time and to make it easy to allocate nodes for the next batch jobs in advance. Strategy (2) contributes to an efficiency

job execution. The job can use the nodes exclusively and the processes in each node can be executed simultaneously. As a result, the large-scale parallel program is able to be executed efficiently.

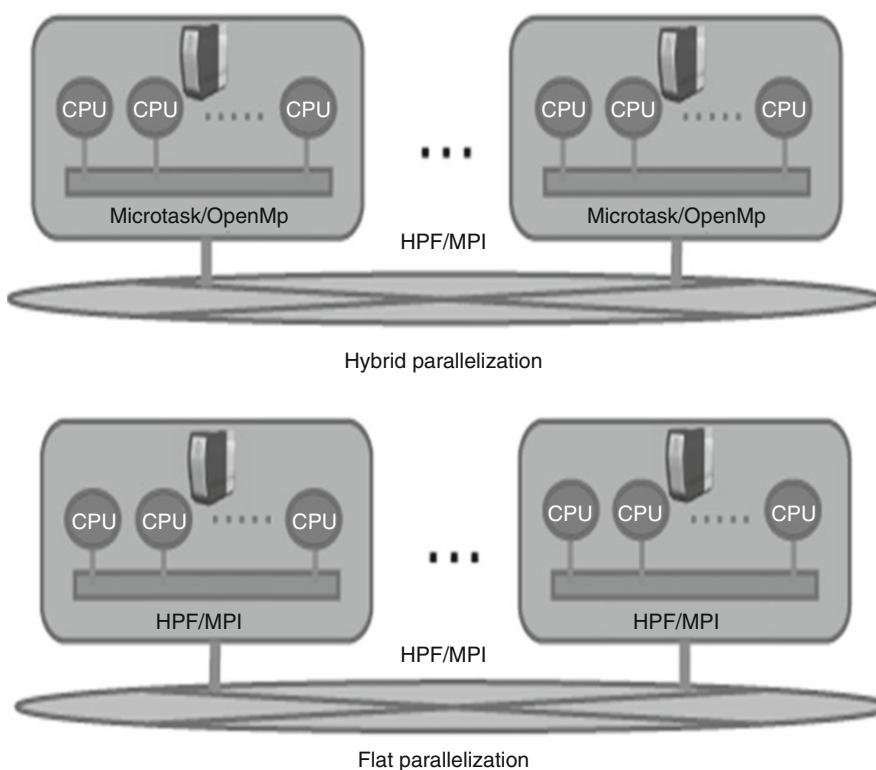
PNs of L-system are prohibited from access to the user disk to ensure enough disk I/O performance. Therefore, the files used by the batch job are copied from the user disk to the work disk before the job execution. This process is called “stage-in.” It is important to hide this staging time for the job scheduling.

Main steps of the job scheduling are summarized as follows:

1. Node allocation
2. Stage-in (copies files from the user disk to the work disk automatically)
3. Job escalation (rescheduling for the earlier estimated start time if possible)
4. Job execution
5. Stage-out (copies files from the work disk to the user disk automatically)

Earth Simulator. Table 1 Programming model on ES

	“Hybrid”	“Flat”
Inter-PN	HPF/MPI	HPF/MPI
Intra-PN	Microtasking/openMP	
AP	Automatic vectorization	



Earth Simulator. Fig. 6 Two types of parallelization of ES

When a new batch job is submitted, the scheduler searches available nodes (Step 1). After the nodes and the estimated start time are allocated to the batch job, stage-in process starts (Step 2). The job waits until the estimated start time after stage-in process is finished. If the scheduler finds the earlier start time than the estimated start time, it allocates the new start time to the batch job. This process is called “Job Escalation” (Step 3). When the estimated start time has arrived, the scheduler executes the batch job (Step 4). The scheduler terminates the batch job and starts stage-out process after the job execution is finished or the declared elapsed time is over (Step 5) ([Fig. 5](#)).

Programming Model in ES

The ES hardware has a three-level hierarchy of parallelism: vector processing in an AP, parallel processing with shared memory in a PN, and parallel processing among PNs via IN. To bring out high performance of ES fully, you must develop parallel programs that make the most use of such parallelism.

As shown in [Table 1](#), the three-level hierarchy of parallelism of ES can be used in two manners, which are called hybrid and flat parallelization, respectively ([Fig. 6](#)). In the hybrid parallelization, the internode parallelism is expressed by HPF or MPI, and the intranode by microtasking or OpenMP, and you must, therefore, consider the hierarchical parallelism in writing your programs. In the flat parallelization, the both inter- and intranode parallelism can be expressed by HPF or MPI, and it is not necessary for you to consider such complicated parallelism. Generally speaking, the hybrid parallelization is superior to the flat in performance and vice versa in ease of programming. Note that the MPI libraries and the HPF runtimes are optimized to perform as well as possible both in the hybrid and flat parallelization.

Bibliography

1. Sato T, Kitawaki S, Yokokawa M (2002) Earth simulator running In: 17th international supercomputer conference ISC2002, Hedielberg
2. Kitawaki S (2002) The development of the earth simulator. In: LACSI symposium 2002 (Keynote speech), Santa Fe
3. Habata S, Kitawaki S, Yokokawa M (2003) The earth simulator system. NEC Res Dev 44:(1):21–26
4. Inasaka J, Ikeda R, Umezawa K, Ko Y, Yamada S, Kitawaki S (2003) Hardware technology of the earth simulator. NEC Res Dev 44(1):27–36

Eden

RITA LOOGEN

Philipps-Universität Marburg, Marburg, Germany

Definition

Eden is a parallel functional programming language that extends the non-strict functional language Haskell with constructs for the definition and instantiation of parallel processes. The programmer is freed from managing synchronization and data exchange between processes while keeping full control over process granularity, data distribution, and communication topology. Eden is geared toward distributed settings, that is, processes do not share any data. Common and sophisticated parallel communication patterns and topologies, that is, algorithmic skeletons, are provided as higher-order functions in a skeleton library written in Eden.

Eden is implemented on the basis of the Glasgow Haskell Compiler GHC [[31](#)], a mature and efficient Haskell implementation. While the compiler frontend is almost unchanged, the backend is extended with a *parallel* runtime system (PRTS) [[7](#)]. This PRTS uses suitable middleware (currently PVM or MPI) to manage parallel execution.

Discussion

Introduction

Functional languages are promising candidates for parallel programming, because of their high level of abstraction and, in particular, because of their referential transparency. In principle, any sub-expression could be evaluated in parallel. As this implicit parallelism would lead to too much overhead, modern parallel functional languages allow the programmers to specify parallelism explicitly. The underlying idea of Eden is to enable programmers to specify process networks in a declarative way. Processes map input to output values. Inputs and outputs are transferred via unidirectional one-to-one channels. A comprehensive definition of Eden including a discussion of its formal semantics and its implementation can be found in the journal paper [[27](#)]. We describe only the essentials here.

Basic Eden Constructs

The basic Eden coordination constructs are *process abstraction* and *instantiation*:

```
process :: (Trans a, Trans b) =>
  (a -> b) -> Process a b
(#)    :: (Trans a, Trans b) =>
  Process a b -> a -> b
```

The function `process` embeds functions of type `a -> b` into *process abstractions* of type `Process a b`, while the instantiation operator (`#`) takes such a process abstraction and input of type `a` to create a new process which consumes the input and produces output of type `b`. Thus, the instantiation operator leads to a function application, with the side effect that the function is evaluated by a remote child process. Its argument is concurrently, that is, by a new thread, evaluated in the parent process and sent to the child process which, in turn, fully evaluates and sends back the result of the function application. Both processes are using *implicit 1:1 communication channels* established between child and parent process on process instantiation.

The type context `(Trans a, Trans b)` states that both `a` and `b` must be types belonging to the type class `Trans` of transmissible values. In general, Haskell type classes provide a structured way to define overloaded functions. `Trans` defines *implicitly* used communication functions which by default transmit normal form values in a single piece. The overloading is used twofold. Lists are communicated element-wise as streams and tuple components are evaluated concurrently. An independent thread will be created for each component of an output tuple and the result of its evaluation will be sent on a separate channel. The connection points of channels to processes are called *imports* on the receiver side and *outports* on the sender side. There is a one-to-one correspondence between the threads and the outports of a process while data that is received via the imports is shared by all threads of a process. Analogously, several threads will be created in a parent process for tuple inputs of a child process. During its lifetime, an Eden process can thus contain a variable number of threads.

The demand-driven (lazy) evaluation of Haskell is an obstacle for parallelism. A completely demand-

driven evaluation creates a parallel process only when its result is already needed to continue the main evaluation. This suppresses any real parallel evaluation. Although Eden overrules the lazy evaluation, for example, by evaluating inputs and outputs of processes eagerly and by sending those values immediately to the corresponding recipient processes, it is often necessary to use explicit demand control in order to start processes speculatively before their result values are needed by the main computation. Evaluation strategies [32] are used for that purpose. We will not go into further details. In the following, we will use the (pre-defined) Eden function `spawn` to eagerly and immediately instantiate a finite list of process abstractions with their corresponding inputs. Appropriate demand control is incorporated in this function. Neglecting demand control, `spawn` would be defined as follows:

```
spawn :: (Trans a, Trans b) =>
  [Process a b] -> [a] -> [b]
-- definition without demand control
spawn = zipWith (#)
```

The variant `spawnAt` additionally locates the created processes on given processor elements (identified by their number).

```
spawnAt :: (Trans a, Trans b) =>
  [Int] -> [Process a b] -> [a] -> [b]
```

Example: A parallel variant of the function `map :: (a -> b) -> [a] -> [b]` which creates a process for each application of the parameter function to an element of the input list can be defined as follows:

```
parMap :: (Trans a, Trans b) =>
  (a -> b) -> [a] -> [b]
parMap f
= spawn (repeat (process f))
```

The Haskell prelude function `repeat :: a -> [a]` yields a list infinitely repeating the parameter element. □

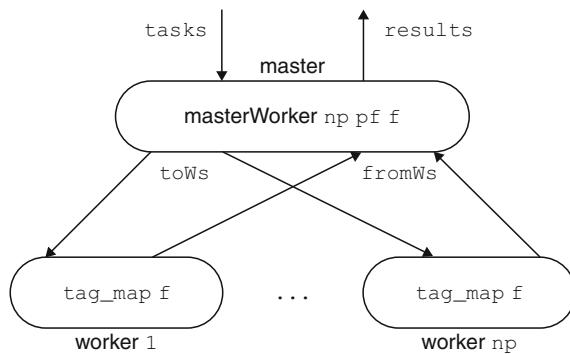
To model many-to-one communication, Eden provides a nondeterministic function `merge` that merges a list of streams into a single stream.

```
merge :: Trans a => [[a]] -> [a]
```

This function can, for example, be used by the master in a master-worker system to receive the results of the workers as a single stream in the time-dependent order in which they are provided.

Although `merge` is of great worth, because it is the key to specify many reactive systems, one has to be aware that functional purity and its benefits are lost when `merge` is being used in a program. Fortunately, functional purity can be preserved in most portions of an Eden program. In particular, it is, for example, possible to use sorting in order to force a particular order of the results returned by a `merge` application and thus to encapsulate `merge` in a deterministic context.

Example: A simple master-worker system with the following functionality can easily be defined in Eden as shown in Fig. 1. The master process distributes tasks to worker processes, which solve the tasks and return the results to the master:



The Eden function `masterWorker` (evaluated by the “master” process) takes four parameters: `np` specifies the number of worker processes that will be spawned, `prefetch` determines how many tasks will initially be sent by the master to each worker process, the function `f` describes how tasks have to be solved by the workers, and the final parameter `tasks` is the list of tasks that have to be solved by the whole system. The auxiliary pure Haskell function `distribute :: Int -> [a] -> [Int] -> [[a]]` (code not shown) is used to distribute the tasks to the workers. Its first parameter determines the number of output lists, which become the input streams for the worker processes. The third parameter is the request list `reqs` which guides the task distribution. The same list is used to sort

the results according to the original task order (function `orderBy :: [[b]] -> [Int] -> [b]`, code not shown).

Initially, the master sends as many tasks as specified by the parameter `prefetch` in a round-robin manner to the workers (see definition of `initReqs`). Further tasks are sent to workers which have delivered a result. The list `newReqs` is extracted from the worker results tagged with the corresponding worker id which are merged according to the arrival of the worker results. This simple master-worker definition has the advantage that the tasks need not be numbered to reestablish the original task order on the results. Moreover, worker processes need not send explicit requests for new work together with the result values.

Defining Non-hierarchical Process Networks in Eden

With the Eden constructs introduced up to now, communication channels are only established between parent and child processes during process creation. This results in purely hierarchical process topologies. Eden further provides functions to create and use explicit channel connections between arbitrary processes. These functions are rather low level and it is not easy to use them appropriately. Therefore, we will not show them here, but explain two more abstract approaches to define non-hierarchical process networks in Eden: remote data and graph specifications.

Remote data of type `a` is represented by a handle of type `RD a` with the following interface functions [13]:

```
release :: a -> RD a
fetch   :: RD a -> a.
```

The function `release` yields a remote data handle that can be passed to other processes, which will in turn use the function `fetch` to access the remote data. The data transmission occurs automatically from the process that releases the data to the process which uses the handle to fetch the remote data.

Example: We show a small example where the remote data concept is used to establish a direct channel connection between sibling processes. Given functions `f` and `g`, one can calculate $(g \circ f) a$ in parallel creating a process for each function.

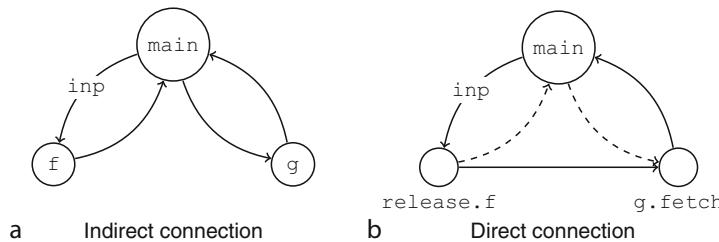
```

masterWorker :: (Trans a, Trans b) =>
               Int → Int → (a → b) → [a] → [b]
masterWorker np prefetch f tasks
= orderBy fromWs reqs
where
  fromWs = spawn workers toWs
  workers = process (tag_map f) | n ← [1..np]
  toWs = distribute np tasks reqs
  newReqs = merge [[i | r ← rs] | (i, rs) ← zip [1..np] fromWs]
  reqs = initReqs ++ newReqs
  initReqs = concat (replicate prefetch [1..np])

```

E

Eden. Fig. 1 Eden definition of a simple master-worker system



Eden. Fig. 2 A simple process graph

Simply replacing the function calls by process instantiations

```
(process g # (process f # inp))
```

leads to the process network in Fig. 2a where the process evaluating the above expression is called `main`. Process `main` instantiates a first child process calculating `g`, thereby creating a concurrent thread to evaluate the input for the new child process. This thread instantiates the second child process for calculating `f`. It receives the remotely calculated result of the `f` process and passes it to the `g` process. The drawback of this approach is that the result of the `f` process will not be sent directly to the `g` process. This causes unnecessary communication costs.

In the second implementation, we use remote data to establish a direct channel connection between the child processes:

```
process (g o fetch) #
  (process (release of) # inp)
```

It uses function `release` to produce a handle of type `RD a` for data of type `a`. Calling `fetch` with remote data returns the value released before. The use of remote

data leads to a direct communication of the actual data between the processes evaluating `f` and `g` (see Fig. 2b). The remote data handle is treated like the original data in the first version, that is, it is passed via the main process from the process computing `f` to the one computing `g`. \triangleleft

The remote data concept enables the programmer to easily build complex topologies by combining simpler ones [13].

Grace

(Graph-based Communication in Eden) is a library that allows a programmer to specify a network of processes as a graph, where the graph nodes represent processes and the edges of communication channels [22]. The graph is described as a Haskell data structure `ProcessNetwork a`, where `a` is the type of the result computed by the process network. A function `build` is used to define a process network, while a function `start` instantiates such a network and automatically sets up the corresponding process topology, that is, the processes are created and the necessary communication channels are installed.

```

build :: forall f a g r p n e.
    -- type context
    (Placeable f a g r p,
     Ord e, Eq n) =>
    (n, f)      -> -- main node
    [Node n]    -> -- node list
    [Edge n e]  -> -- edge list
    ProcessNetwork r
start :: (Trans a) =>
    ProcessNetwork a -> a
  
```

The function `build` transforms a graph specification into a value of type `ProcessNetwork r`. The graph is defined by a list of nodes of type `[Node n]` and a list of edges of type `[Edge n e]`. Type variables `n` and `e` represent the types of node and edge labels, respectively. To allow functions with different types to be associated with graph nodes, the type `f` of node functions is existentially quantified and only explicitly given for the main node, because the function placed on the main node determines the result type `r` of the process network. This is also the reason why the main node is not a member of the node list but provided as a separate parameter. The function `build` uses multiparameter classes with functional dependencies and explicit quantification to achieve flexibility in the specification of graphs, for example, by placing arbitrary functions on nodes and nevertheless using standard Haskell lists to represent node lists. The type context `Ord e` and `Eq n` ensures that edges can be ordered by their label and that nodes can be identified by their label. `Placeable` is a multiparameter type class with dependent types, which is used by the Grace implementation to partition user supplied function types into their parts, for example,

```

example = start network
where
  -- fct. spec. of process behaviour
  f, g, root :: (Trans a) -> [a] -> [a]
  ...
  -- process graph specification
  network = build ("root", root) nodes edges
  nodes   = [N "nd_f" f, N "nd_g" g]
  edges   = [E "nd_f" "nd_g" 0 nothing,
            E "nd_g" "root" 0 nothing,
            E "root" "nd_f" 0 nothing]
  
```

parameter and result types. This is needed to create individual channels for these parts. Suitable instances will be derived automatically for every possible function. In the type context `Placeable f a g r p` the function's type `f` determines the other types: `a` is the type of the function's first argument, `g` is the remaining part of the function's type without the first argument. The final result type of the function after applying all parameters is `r`. Finally, `p` is a type level list of all the parameters.

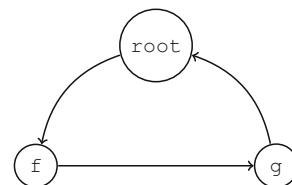
The main benefit of Grace is a clean separation between coordination and computation. The network specification encapsulates the coordination aspects. The graph nodes are annotated with functions describing the computations of the corresponding processes.

Example: We consider again a simple process network which is similar to the one in Fig. 2b and show its specification in Grace. The behavior of the three processes is specified by functions `f`, `g`, and `root`. The communication topology is defined by a graph consisting of three nodes (a root node and two additional nodes) and three edges. The specification is shown in Fig. 3. \triangleleft

Algorithmic Skeletons

Algorithmic skeletons [11] capture common patterns of parallel evaluations like task farms, pipelines, divide-and-conquer schemes, etc. The application programmer only needs to instantiate a skeleton appropriately, thereby concentrating on the problem-specific matters and trusting on the skeleton with respect to all parallel details.

The small introductory example functions `parMap` and `masterWorker` shown above are examples for skeleton definitions in Eden. In Eden and other parallel functional languages, skeletons are no more than polymorphic higher-order functions which can be applied



Eden. Fig. 3 Grace specification of a simple process graph

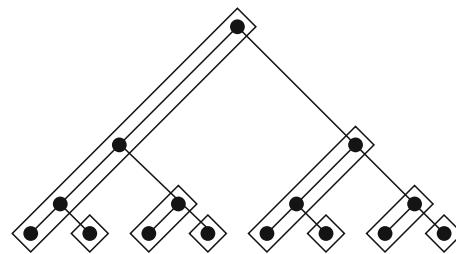
with different types and parameters. Thus, programming with skeletons follows the same principle as programming with higher-order functions. An important issue is that skeletons can be both *used* and *implemented* in Eden. In other skeletal approaches, the creation of new skeletons is considered as a system programming task, or even as a compiler construction task. Skeletons are implemented by using imperative languages and/or parallel libraries. Therefore, these systems offer a closed collection of skeletons which the application programmer can use, but without the possibility of creating new ones, so that adding a new skeleton usually implies a considerable effort.

The Eden skeleton library contains a big collection of different skeletons. Various kinds of skeleton definitions in Eden together with cost models to predict the execution time of skeleton instantiations have been presented in a book chapter [26]. Parallel map implementations have been analyzed in [23]. An Eden implementation of the large-scale *map-and-reduce* programming model proposed by Google [12] has been investigated in [3]. Hierarchical master-worker schemes with several layers of masters and submasters can elegantly be created by nesting a simple single-layer master-worker scheme. More elaborate hierarchical master-worker systems have been discussed in [4].

Example: (A regular fixed-branching divide and conquer skeleton) As a nontrivial example of a skeleton definition we present a dynamically unfolding divide-and-conquer skeleton [2]. The essence of a divide-and-conquer algorithm is to decide whether the input is trivial and, in this case, to solve it, or else to decompose nontrivial input into a number of subproblems, which are solved recursively, and to combine the output. A general skeleton takes parameter functions for this functionality, as shown here:

```
type DivideConquer a b
= (a → Bool) → -- trivial?
  (a → b) → -- solve
  (a → [a]) → -- split
  ([b] → b) → -- combine
  a → b → -- input / result
```

The resulting structure is a tree of task nodes where successor nodes are the sub-problems, the leaves representing trivial tasks. A fundamental Eden skeleton which specifies a general divide and conquer algorithm structure can be found in [26]. Here, we show a version where every nontrivial task is split into a *fixed* number of sub-tasks. Moreover, the process tree is created in a *distributed* fashion: One of the tree branches is processed locally, the others are instantiated as new processes, as long as PEs are available. These branches will recursively produce new parallel subtasks, resulting in a *distributed expansion* of the computation. In the following binary tree of task nodes, the boxes indicate which task nodes will be evaluated by the eight processes:



In this setting, explicit placement of processes is essential to ensure that processes are not placed on the same processor element while leaving others unused. In [2], this kind of skeleton is compared with a *flat expansion* version, where the main process unfolds the tree up to a given depth, usually with more branches than available processor elements (PEs). The resulting subtrees can then be evaluated by a farm of parallel worker processes, the main process combines the results of the subprocesses. A master-worker system (as shown above) can be used to implement this version.

Figure 4 shows the distributed expansion divide-and-conquer skeleton for k -ary task trees. Besides the standard parameter functions, the skeleton takes the branching degree, and a ticket list with PE numbers to place newly created processes. The left-most branch of the task tree is solved locally, other branches are instantiated using the Eden function `spawnAt`, which instantiates a collection of processes (given as a list) with respective input, on explicitly specified PEs. Results are combined by the `combine` function.

Explicit Placement via Tickets: The ticket list is used to control the placement of newly created processes. First, the PE numbers for placing the immediate child

```

dcN :: ('Trans a, Trans b) =>
    Int → [Int] → -- branch degree/tickets
    DivideConquer a b
dcN k tickets trivial solve split combine x
| null tickets = seqDC x
| trivial x     = solve x
| otherwise      = ... -- code for demand control omitted
                    combine (myRes:childRes ++ localRes)
where
    -- sequential computation
    seqDC x = if trivial x then solve x
               else combine (map seqDC (split x))
    -- child process generation
    childRes = spawnAt childTickets childProcs procIns
    childProcs = map (process o rec_dcN) theirTs
    rec_dcN ts = dcN k ts trivial solve split combine
    -- ticket distribution
    (childTickets, restTickets) = splitAt (k-1) tickets
    (myTs: theirTs)             = unshuffle k restTickets
    -- input splitting
    (myIn:theirIn)              = split x
    (procIns, localIns)          = splitAt (length childTickets) theirIn
    -- local computations
    myRes = ticketF myTs myIn
    localRes = map seqDC localIns

```

Eden. Fig. 4 Distributed expansion divide and conquer skeleton for k -ary task trees

processes are taken from the ticket list. Then, the remaining tickets are distributed to the children in a round-robin manner using the function `unshuffle :: Int -> [a] -> [[a]]` which unshuffles a given list into as many lists as the first parameter tells. Child computations will be performed locally when no more tickets are available. The explicit process placement via ticket lists is a simple and flexible way to control the distribution of processes as well as the recursive unfolding of the task tree. If too few tickets are available, computations are performed locally. Duplicate tickets can be used to allocate several child processes on the same PE. \triangleleft

Implementation

Eden's implementation follows a microkernel approach. The kernel implements a few, general control mechanisms and provides them as primitive operations (see Fig. 5). The more complex high-level language constructs are implemented in libraries using the primitive base constructs. Apart from simplifying the implementation, this approach has important advantages with respect to productivity and maintainability of the system.

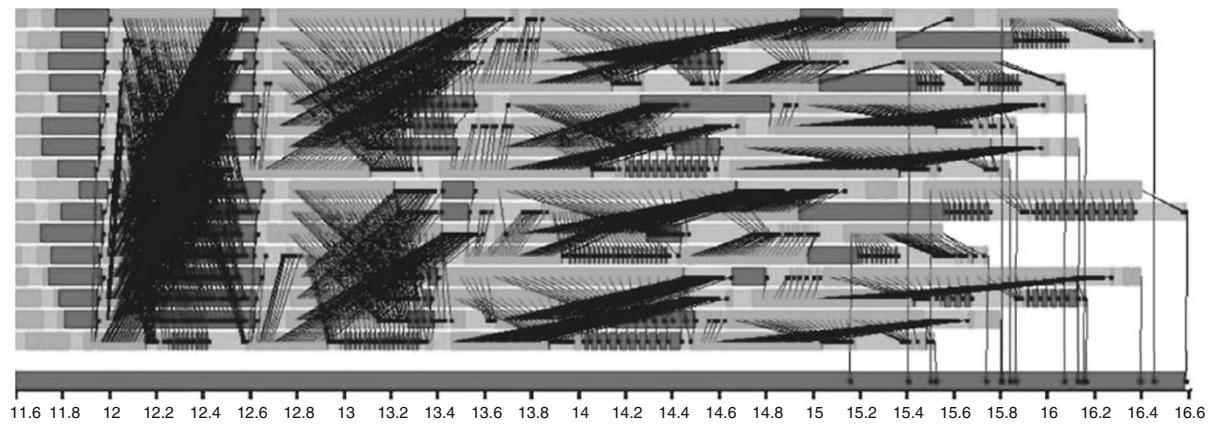
Details on Eden's implementation, especially on primitives provided by the interface of the parallel runtime system (PRTS) and the implementation of the Eden module can be found in [5, 7].

EdenTV – The Eden Trace Viewer Tool

The Eden trace viewer tool (EdenTV) [8] provides a *postmortem analysis* of program executions on the level of the computational units of the PRTS. The latter is instrumented with special trace generation commands activated by a runtime option. In the space-time diagrams generated by EdenTV, machines (i.e., processor elements), processes, and threads are represented by horizontal bars, with time on the x -axis. The diagram bars have segments in different colors, which indicate the activities of the respective logical unit in a period during the execution. Bars are grey, when the logical unit is running, light grey, when it is runnable, but currently not running, and dark grey, when the unit is blocked. Messages between processes or machines are optionally shown by gray arrows which start from the sending unit bar and point at the receiving unit bar. The representation of messages is very important for programmers, since they can observe

Eden program		Language
Eden module and libraries	Haskell libraries	System
Eden primitives		Kernel
Parallel RTS	Sequential RTS	
Suitable middleware		

Eden. Fig. 5 Implementation layers



Zoom of the final communication phase of the hyperquicksort algorithm when sorting 5M elements on 17 machines of a Beowulf cluster. Messages are shown as black arrows.

Eden. Fig. 6 Trace of hypercube communication topology of dimension 4

hot spots and inefficiencies in the communication during the execution as well as control communication topologies.

Figure 6 shows in grayscale (light grey = yellow, grey = green, dark grey = red) the trace of a recursive parallel quicksort in a hypercube of dimension 4 with message traffic, exposing the well-known butterfly communication pattern imposed by the hypercube structure.

Future Directions

Lines of future studies are

- It is planned for the future to maintain a *common parallel runtime environment* for Eden, Glasgow parallel Haskell (GpH), and other parallel Haskells. Such a common environment is highly desirable and

would improve the availability and acceptance of parallel Haskells.

- Recent results [9] show that the Eden system, although tailored for distributed memory machines, behaves equally well on workstation clusters and on *multi-core machines*. It needs to be investigated whether the Eden system can be optimized for multi-core machines by exploiting the shared memory and thereby avoiding unnecessary communications.
- We plan to design and investigate further skeletons. The flexibility of defining individual skeletons for several algorithm classes has not been completely exploited yet. Much more experience in the design and use of skeletons is necessary.
- Recent studies on remote data and the Grace system have shown that high-level parallel programming notations simplify parallel programming a lot. More

research is necessary to find even more abstract parallel programming constructs.

- Supporting tools like profiling, analysis, visualization, and debugging tools are essential for real-world parallel program development. Much more work is needed to further develop existing, and to design new powerful tools which help the programmers to analyze and tune parallel programs.

Related Entries

- ▶ [Concurrent ML](#)
- ▶ [Functional Languages](#)
- ▶ [Glasgow Parallel Haskell \(GpH\)](#)
- ▶ [Multilisp](#)
- ▶ [NESL](#)
- ▶ [Parallel Skeletons](#)
- ▶ [Sisal](#)

Glasgow parallel Haskell (GpH) is another parallel Haskell extension that is less explicit about parallelism than Eden. It provides a simple parallel combinator `par` to annotate sub-expressions that should be evaluated in parallel if free processing elements were available. In contrast to Eden, GpH assumes a global shared memory (at least virtually), and the parallel runtime system decides whether an annotated sub-expression will be evaluated in parallel or not.

NESL is a strict data-parallel functional language that supports nested parallel array comprehensions. Parallelism is hidden and concentrated in data-parallel operations.

SISAL (Streams and Iteration in a Single Assignment Language) and *MultiLisp* are early strict parallel functional languages. SISAL is a first-order functional language with implicit parallelism, a rich set of array operations and stream support. It has been designed for numerical applications. MultiLisp extends the LISP dialect Scheme with explicit parallel constructs, in particular so-called futures. Futures initiate parallel evaluations without forcing the main computation to wait for the result. In this respect, futures lead to a special form of laziness within a strict computation language.

Concurrent ML extends the strict functional language SML (Standard ML) with concurrency primitives like thread creation and synchronous message passing over explicit channels. Blocking send and

receive functions are provided for channel communication. First-class synchronous operations, called events, however, allow to hide channels and complex communication and synchronization protocols behind appropriate abstractions. In contrast to Eden and other parallel functional languages, Concurrent ML has been designed for concurrent programming, that is, with a focus on structuring software and not with the goal of speeding up computations. Recently, however, a shared-memory parallel implementation has been developed [30].

Manticore is a strict parallel functional language that supports different kinds of parallelism. In particular, it combines explicit concurrency in the style of Concurrent ML with various implicitly parallel constructs which provide fine-grain data parallelism.

Bibliographic Notes and Further Reading

The seminal book on research directions in parallel functional programming [17] edited by Hammond and Michaelson covers not only fundamental issues but also provides summaries about selected research areas. Various parallel and distributed variants of the functional programming language Haskell are discussed in a journal paper by Trinder et al. [33]. A comprehensive overview on patterns and skeletons for parallel and distributed programming can be found in a book edited by Gorlatch and Rabhi [29]. The three parallel functional languages, Glasgow parallel Haskell, PMLS (a parallel version of ML), and Eden, have been compared with respect to programming methodology and performance in [25].

Comprehensive and up-to-date information on Eden is provided on its web site

<http://www.mathematik.uni-marburg.de/~eden>

Basic information on its design, semantics, and implementation as well as the underlying programming methodology can be found in [27]. Details on the parallel runtime system and Eden's concept of implementation can best be found in [5, 7]. The technique of layered parallel runtime environments has been further developed and generalized by Berthold et al. [1, 10]. The Eden trace viewer tool EdenTV is available on Eden's web site. A short introductory description is given in [8]. Another tool for analyzing the behavior of Eden programs has been developed by de la Encina et al. [14, 15]

by extending the tool Hood (Haskell Object Observation Debugger) for Eden. Extensive work has been done on skeletal programming in Eden. An overview on various skeleton types have been presented as a chapter in the mentioned book by Gorlatch and Rabhi [26]. Definitions and applications of specific skeletons can, for example, be found in the following papers: parallel map [23], topology skeletons [6], adaptive skeletons [16], Google map-reduce [3], hierarchical master-worker systems [4], divide-and-conquer schemes [2]. Special skeletons for computer algebra algorithms are developed with the goal to define the kernel of a computer algebra system in Eden [24]. An operational and a denotational semantics for Eden have been defined by Ortega-Mallén and Hidalgo-Herrero [18, 19]. These semantics have been used to analyze Eden skeletons [20, 21]. A non-determinism analysis has been presented by Segura and Peña [28].

Eden is intensively being used in teaching parallel programming and algorithms and as a platform for investigating high-level parallel programming concepts, techniques, methodology, and parallel runtime systems. Up to now, its main use in practice has been the implementation of parallel computer algebra components [24, 34].

Acknowledgments

The author is grateful to Yolanda Ortega-Mallén, Jost Berthold, Mischa Dieterle, Thomas Horstmeyer, and Oleg Lobachev for their helpful comments on previous versions of this essay.

Bibliography

1. Berthold J (2004) Towards a generalised runtime environment for parallel Haskell s. Computational Science – ICCS’04, LNCS 3038. Springer (Workshop on practical aspects of High-level parallel programming – PAPP 2004)
2. Berthold J, Dieterle M, Lobachev O, Loogen R (2009) Distributed memory programming on many-cores - a case study using Eden divide-&-conquer skeletons. ARCS 2009, Workshop on Many-Cores. VDE Verlag
3. Berthold J, Dieterle M, Loogen R (2009) Implementing parallel google map-reduce in Eden. Europar’09, LNCS 5704, Springer, pp 990–1002
4. Berthold J, Dieterle M, Loogen R, Priebe S (2008) Hierarchical master-worker skeletons. Practical Aspects of Declarative Languages (PADL 2008), LNCS 4902, Springer, pp 248–264
5. Berthold J, Klusik U, Loogen R, Priebe S, Weskamp N (2003) High-level process control in Eden. EuroPar 2003 – Parallel Processing, LNCS 2790, Springer, pp 732–741
6. Berthold J, Loogen R (2006) Skeletons for recursively unfolding process topologies. Parallel Computing: Current & Future Issues of High-End Computing, ParCo 2005, NIC Series, vol 33, pp 835–842
7. Berthold J, Loogen R (2007) Parallel coordination made explicit in a functional setting. Implementation and Application of Functional Languages (IFL 2006), Selected Papers, LNCS 4449, Springer, pp 73–90 (awarded best paper of IFL’06)
8. Berthold J, Loogen R (2007) Visualizing parallel functional program runs – case studies with the Eden trace viewer. Parallel Computing: Architectures, Algorithms and Applications, ParCo 2007, NIC Series, vol 38, pp 121–128
9. Berthold J, Marlow S, Zain AA, Hammond K (2009) Comparing and optimising parallel Haskell implementations on multi-core. 3rd International Workshop on Advanced Distributed and Parallel Network Applications (ADPNA-2009). IEEE Computer Society
10. Berthold J, Zain AA, Loidl H-W (2008) Scheduling light-weight parallelism in ArtCoP. Practical Aspects of Declarative Languages (PADL 2008), LNCS 4902, Springer, pp 214–229
11. Cole M (1989) Algorithmic skeletons: structured management of parallel computation. MIT Press, Cambridge
12. Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Commun ACM 51(1):107–113
13. Dieterle M, Horstmeyer T, Loogen R (2010) Skeleton composition using remote data. Practical Aspects of Declarative Programming 2010 (PADL 2010), LNCS 5937, Springer, pp 73–87
14. Encina A, Llana L, Rubio F, Hidalgo-Herrero M (2007) Observing intermediate structures in a parallel lazy functional language. Principles and Practice of Declarative Programming (PPDP 2007), ACM, pp 109–120
15. Encina A, Rodríguez I, Rubio F (2009) pHood: a tool to analyze parallel functional programs. Implementation of Functional Languages (IFL’09), Seton Hall University, New York, pp 85–99. Technical Report, SHU-TR-CS-2009-09-1
16. Hammond K, Berthold J, Loogen R (2003) Automatic skeletons in template Haskell. Parallel Process Lett 13(3):413–424
17. Hammond K, Michaelson G (eds) (1999) Research directions in parallel functional programming. Springer, Heidelberg
18. Hidalgo-Herrero M, Ortega-Mallén Y (2002) An operational semantics for the parallel language Eden. Parallel Process Lett 12(2):211–228
19. Hidalgo-Herrero M, Ortega-Mallén Y (2003) Continuation semantics for parallel Haskell dialects. Asian Symposium on Programming Languages and Systems (APLAS 2003), LNCS 2896, Springer, pp 303–321
20. Hidalgo-Herrero M, Ortega-Mallén Y, Rubio F (2006) Analyzing the influence of mixed evaluation on the performance of Eden skeletons. Parallel Comput 32(7–8):523–538
21. Hidalgo-Herrero M, Ortega-Mallén Y, Rubio F (2007) Comparing alternative evaluation strategies for stream-based parallel functional languages. Implementation and Application of Functional Languages (IFL 2007), LNCS 4732, Springer, pp 23–36

- Languages (IFL 2006), Selected Papers, LNCS 4449, Springer, pp 55–72
22. Horstmeyer T, Loogen R (2010) Graph-based communication in Eden. In: Trends in Functional Programming, vol 10. Intellect
 23. Klusik U, Loogen R, Priebe S, Rubio F (2001) Implementation skeletons in Eden – low-effort parallel programming. Implementation of Functional Languages (IFL 2000), Selected Papers, LNCS 2011, Springer, pp 71–88
 24. Lobachev O, Loogen R (2008) Towards an implementation of a computer algebra system in a functional language. AISC/Calculemus/MKM 2008, LNAI 5144, pp 141–174
 25. Loidl H-W, Rubio Diez F, Scaife N, Hammond K, Klusik U, Loogen R, Michaelson G, Horiguchi S, Peña Mari R, Priebe S, Portillo AR, Trinder P (2003) Comparing parallel functional languages: programming and performance. Higher-Order Symbolic Computation 16(3):203–251
 26. Loogen R, Ortega-Mallén Y, Peña R, Priebe S, Rubio F (2003) Parallelism abstractions in Eden. In [29], chapter 4, Springer, pp 95–128
 27. Loogen R, Ortega-Mallén Y, Peña-Marí R (2005) Parallel functional programming in Eden. J Funct Prog 15(3):431–475
 28. Peña R, Segura C (2001) Non-determinism analysis in a parallel-functional language. Implementation of Functional Languages (IFL 2000), LNCS 1268, Springer
 29. Rabhi FA, Gorlatch S (eds) Patterns and skeletons for parallel and distributed computing. Springer, Heidelberg
 30. Reppy J, Russo CV, Yiao Y (2009) Parallel concurrent ML. International Conference on Functional Programming (ICFP) 2009, ACM
 31. The GHC Developer Team. The Glasgow Haskell Compiler. <http://www.haskell.org/ghc>
 32. Trinder P, Hammond K, Loidl H-W, Peyton Jones S (1998) Algorithm + strategy = parallelism. J Funct Prog 8(1):23–60
 33. Trinder PW, Loidl HW, Pointon RF (2002) Parallel and distributed Haskells. J Funct Prog 12(4 & 5):469–510
 34. Zain AA, Berthold J, Hammond K, Trinder PW (2008) Orchestrating production computer algebra components into portable parallel programs. Open Source Grid and Cluster Conference (OSGCC)

Eigenvalue and Singular-Value Problems

BERNARD PHILIPPE¹, AHMED SAMEH²

¹Campus de Beaulieu, Rennes, France

²Purdue University, West Lafayette, IN, USA

Definition

Given a matrix $A \in \mathbb{R}^{n \times n}$ or $A \in \mathbb{C}^{n \times n}$, an **eigenvalue problem** consists of computing eigen-elements of A which are:

either eigenvalues: these are the roots λ of the n -degree characteristic polynomial:

$$\det(A - \lambda I) = 0. \quad (1)$$

The set of all eigenvalues is called the spectrum of A : $\Lambda(A) = \{\lambda_1, \dots, \lambda_n\}$. Eigenvalues can be real or complex, whether the matrix is real or complex.

or eigenpairs: in addition to an eigenvalue λ , one seeks for the corresponding eigenvector $x \in \mathbb{C}^n - \{0\}$ which is the solution of the singular system

$$(A - \lambda I)x = 0. \quad (2)$$

When the matrix A is real symmetric or complex hermitian, the eigenvalues are real.

Given a matrix $A \in \mathbb{R}^{m \times n}$ (or $A \in \mathbb{C}^{m \times n}$), the **singular-value decomposition (SVD) of A** consists of computing orthogonal matrices $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ (or unitary matrices $U \in \mathbb{C}^{m \times m}$ and $V \in \mathbb{C}^{n \times n}$) such that the $m \times n$ matrix

$$\tilde{\Sigma} = U^T A V \quad (3)$$

(or $\tilde{\Sigma} = U^H A V$) has zero entries except for its diagonal leading $p \times p$ real submatrix $\Sigma = \text{diag}(\sigma_1, \dots, \sigma_p)$ with $\sigma_1 \geq \dots \geq \sigma_p \geq 0$. The first p columns of $U = [u_1, \dots, u_m]$, and of $V = [v_1, \dots, v_n]$, are called the *left* and *right singular vectors*, respectively. The *singular triplets* are (σ_i, u_i, v_i) for $i = 1, \dots, p$.

Discussion

Mathematical Fundamentals

Eigenproblems. Equation 1 has n complex roots, some of them possibly multiple. For every eigenvalue $\lambda \in \mathbb{C}$, the dimension of the eigenspace $\mathcal{X}_\lambda = \ker(A - \lambda I)$ is at least 1. The algebraic multiplicity of λ (its multiplicity as root of (1)) is larger or equal to its geometric multiplicities (the dimension of \mathcal{X}_λ). When all the geometric multiplicities are equal to the algebraic ones, a basis $X = [x_1, \dots, x_n] \in \mathbb{C}^{n \times n}$ of eigenvectors (say *right eigenvectors*) can be built by concatenating bases of the invariant subspaces \mathcal{X}_λ where $\lambda \in \Lambda(A)$. In such a situation, the matrix A is said to be *diagonalizable* since by expressing the same operator in the basis X ,

$$D = X^{-1} A X = \text{diag}(\lambda_1, \dots, \lambda_n). \quad (4)$$

By denoting $X^{-H} = Y = [y_1, \dots, y_n]$, Eq. 4 becomes $D = Y^H A X$ with $Y^H Y = I$. For $i = 1, \dots, n$, the vector y_i is the left eigenvector corresponding to the eigenvalue λ_i .

The well-known *Jordan normal form* generalizes the diagonalization when Eq. 4 does not hold. However, the problem of determining such a decomposition is ill-posed. Transforming it into a well-posed problem can be done by proving the existence of matrices with an assigned block structure in a neighborhood of the matrix. For more details see Kagström and Ruhe's contribution in [25].

A decomposition which is numerically computable is called the *Schur decomposition*, in which a unitary matrix U exists such that the complex matrix

$$T = U^H A U = \begin{pmatrix} \lambda_1 & \tau_{12} & \cdots & \tau_{1n} \\ \ddots & \ddots & & \vdots \\ & \ddots & \tau_{n-1,n} \\ & & & \lambda_n \end{pmatrix}, \quad (5)$$

is upper triangular. Eigenvectors can then be computed from T . When the matrix A is Hermitian – or real symmetric – so is the matrix T which becomes diagonal and real. The *symmetric eigenvalue problem* deserves a lot of attention since it arises frequently in practice.

When the matrix A is real unsymmetric, pairs of complex conjugate eigenvalues may exist. In order to avoid complex arithmetic in that case, the *real Schur decomposition* extends the Schur decomposition by allowing 2×2 diagonal blocks with pairs of conjugate eigenvalues.

SVD problems. Since the complex case is similar to the real one and since the SVD of A^T is obviously obtained from the SVD of A , the study is restricted to the situation $A \in \mathbb{R}^{m \times n}$ with $m \leq n$.

Let A have the singular-value decomposition (3), then, the symmetric matrix

$$B = A^T A \in \mathbb{R}^{n \times n}, \quad (6)$$

has eigenvalues $\sigma_1^2 \geq \dots \geq \sigma_n^2 \geq 0$, with corresponding eigenvectors (v_i), ($i = 1, \dots, n$). B is called the *normal*

matrix, while the augmented symmetric matrix

$$A_{\text{aug}} = \begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix} \quad (7)$$

has eigenvalues $\pm \sigma_1, \dots, \pm \sigma_n$, corresponding to the eigenvectors

$$\frac{1}{\sqrt{2}} \begin{pmatrix} u_i \\ \pm v_i \end{pmatrix}, \quad i = 1, \dots, n.$$

Every method for computing singular values of A is based on computing the eigenpairs of B or A_{aug} .

E

The Symmetric Eigenvalue Problem

Three class of methods are available for solving problems (1) and (2) when A is symmetric:

Jacobi iterations: The oldest method, introduced by Jacobi in 1846 [23], consists of annihilating successively off-diagonal entries of the matrix via orthogonal similarity transformations: $A_0 = A$ and $A_{k+1} = R_k^T A_k R_k$ where R_k is a plane rotation. The scheme is organized into sweeps of $n(n-1)/2$ rotations to annihilate every off-diagonal pairs of symmetric entries once. One sweep involves $6n^3 + O(n^2)$ operations when symmetry is exploited in the computation. The method was abandoned due to high computational cost but has been revived with the advent of parallelism. However, depending on the parallel architecture, it remains more expensive than other solvers.

QR iterations: The aim of the QR algorithm relies on the following iteration:

$$A_0 = A \text{ and } Q_0 = I$$

$$\begin{cases} \text{For } k \geq 0, \\ (Q_{k+1}, R_{k+1}) = qr(A_k) \text{ (QR Factorization)} \\ A_{k+1} = R_{k+1} Q_{k+1}, \end{cases} \quad (8)$$

where the QR factorization is defined in [linear least squares and orthogonalization 12]. Under weak assumptions, matrix A_k approaches a diagonal matrix for sufficiently large k . A direct implementation of (8) would involve $O(n^3)$ arithmetic operations at each iteration. If,

however, one obtains the tridiagonal matrix $T = Q^T A Q$ via orthogonal similarity transformations at a cost of $O(n^3)$ arithmetic operations, the number of arithmetic operations of a QR iteration is reduced to $O(n^2)$ or even $O(n)$ if only the eigenvalues are needed. To accelerate convergence, a shifted version of the QR algorithm is given by

$$\left\{ \begin{array}{l} T_0 = T \text{ and } Q_0 = I \\ \text{For } k \geq 0, \\ (Q_{k+1}, R_{k+1}) = qr(T_k - \mu_k I) \text{ (QR Factorization)} \\ T_{k+1} = R_{k+1} Q_{k+1} + \mu_k I, \end{array} \right. \quad (9)$$

where μ_k is an approximation of the smallest eigenvalue of T_k .

Sturm sequence evaluation: The Sturm sequence in $\lambda \in \mathbb{R}$ is defined by $p_0(\lambda) = 1$ and for $k = 1, \dots, n$ by $p_k(\lambda) = \det(A_k - \lambda I_k)$ where A_k and I_k are the principal matrices of order k of A and of the identity matrix. The number $N(\lambda)$ of sign changes in the sequence is equal to the number of eigenvalues of A which are smaller than λ . $N(\lambda)$ can also be computed as the number of negative diagonal entries of the matrix U in the LU factorization of the matrix $A - \lambda I = LU$. As for the QR iteration, the algorithm is applied to the tridiagonal matrix T , so that each Sturm sequence computation has complexity $O(n)$. Sturm sequence allows one to partition a given interval $[a, b]$ as desired to obtain each of the eigenvalues in $[a, b]$ to a desired accuracy. When needed, the eigenvectors are then obtained by inverse iterations.

Many references exist which describe the theory underlying the three methods outlined above, e.g., see [20].

Parallel Jacobi algorithms. A parallel version of the cyclic Jacobi algorithm was given by Sameh [30]. It is obtained by the simultaneous annihilation of several off-diagonal elements by a given orthogonal matrix U_k , rather than only one rotation as is done in the serial version. For example, let A be of order 8 (see Table 1) and consider the orthogonal matrix U_k as the direct sum of four independent plane rotations simultaneously determined. An example of such a matrix is

$$U_k = R_k(1, 3) \oplus R_k(2, 8) \oplus R_k(4, 7) \oplus R_k(5, 6),$$

Eigenvalue and Singular-Value Problems. Table 1

Annihilation scheme as in [30] (first regime)

x	3	6	2	5	1	4	7
	x	2	5	1	4	7	6
		x	1	4	7	3	5
			x	7	3	6	4
				x	6	2	3
					x	5	2
						x	1
							x

where $R_k(i, j)$ is that rotation which annihilates the (i, j) off-diagonal element (\oplus indicates that the rotations are assembled in a single matrix and extended to order n by the identity). Let one sweep be the collection of such orthogonal similarity transformations that annihilate the element in each of the $\frac{1}{2}n(n-1)$ off-diagonal positions (above the main diagonal) only once, then for a matrix of order 8, the first sweep will consist of seven successive orthogonal transformations with each one annihilating distinct groups of maximum four elements simultaneously as described in the Table 1. For the remaining sweeps, the structure of each subsequent transformation U_k , $k > 8$, is chosen to be the same as that of R_j where $j = 1 + (k-1) \bmod 8$. In general, the most efficient annihilation scheme consists of $(2r-1)$ similarity transformations per sweep, where $r = \lfloor \frac{1}{2}(n+1) \rfloor$, in which each transformation annihilates different $\lfloor \frac{1}{2}n \rfloor$ off-diagonal elements (see [30]). Several other annihilation schemes are possible which are based on round-robin techniques. Luk and Park [27] have demonstrated that various parallel Jacobi rotation ordering schemes are equivalent to the sequential row-ordering scheme, and hence share the same convergence properties.

The above algorithms are well-suited for shared memory computers. While they can also be implemented on distributed memory systems, their efficiency on such systems may suffer due to communication costs. In order to increase the granularity of the computation (i.e., to increase the number of floating point operations between two communications), block algorithms are considered. A parallel Block-Jacobi algorithm is given by Gimenez et al. in [19]. This algorithm takes advantage of the symmetry of the matrix.

Tridiagonalization of a symmetric matrix by Householder reductions. The goal is to compute a symmetric tridiagonal matrix T , orthogonally similar to A : $T = Q^T A Q$ where $Q = H_1 \dots H_{n-2}$ combines Householder reductions. These transformations are defined in [linear least square and orthogonalization 12]. The transformation H_1 is chosen such that $n - 2$ zeros are introduced in the first column of $H_1 A$, the first row being unchanged:

$$H_1 A = \begin{pmatrix} * & * & * & * & * & * \\ * & * & * & * & * & * \\ 0 & * & * & * & * & * \\ 0 & * & * & * & * & * \\ 0 & * & * & * & * & * \\ 0 & * & * & * & * & * \end{pmatrix}.$$

Therefore, by symmetry, applying H_1 on the right results in the following pattern:

$$A_1 = H_1 A H_1 = \begin{pmatrix} * & * & 0 & 0 & 0 & 0 \\ * & * & * & * & * & * \\ 0 & * & * & * & * & * \\ 0 & * & * & * & * & * \\ 0 & * & * & * & * & * \\ 0 & * & * & * & * & * \end{pmatrix} \quad (10)$$

In order to take advantage of the symmetry, the reduction $A_1 = H_1 A H_1$ can be implemented by a symmetric rank-two update which is a routine of the Level 2 BLAS [6]. The tridiagonal matrix T is obtained by repeating successively similar reductions on columns 2 to $n - 2$. The total procedure involves $4n^3/3 + O(n^2)$ arithmetic operations. Assembling the matrix $Q = H_1 \dots H_{n-2}$ requires $4n^3/3 + O(n^2)$ additional operations.

As indicated in [linear least square and orthogonalization 12], successive applications of Householder reductions can be compacted into blocks which allow use of routines of Level 3 BLAS [6]. Such reduction is implemented in routine DSYTRD of LAPACK [2] which takes advantage of the symmetry of the process.

A parallel version of the algorithm is implemented in routine PDSYTRD of ScaLAPACK [5].

Parallel QR for the symmetric tridiagonal eigenvalue problem. The QR scheme is the method of choice when all the eigenvalues and the eigenvectors are sought. A Divide-and-Conquer algorithm was introduced by Dongarra and Sorensen in [16]. It consists of partitioning the triadiagonal matrix by rank-one tearings.

A tridiagonal matrix T of order n can be partitioned

$$\text{as follows: } T = \begin{pmatrix} T_1 & 0 \\ 0 & T_2 \end{pmatrix} + \rho f f^T \text{ where } T_1 \text{ and } T_2$$

are two tridiagonal matrices of order n_1 and n_2 such that $n = n_1 + n_2$ and where f is a vector with zero entries except in position n_1 and $n_1 + 1$. If the Schur decompositions $T_1 = Q_1 D_1 Q_1^T$ and $T_2 = Q_2 D_2 Q_2^T$ are already known, the problem is reduced to computing the eigenvalues of $\tilde{D} = Q^T T Q = D + \rho z z^T$ where

$$Q = \begin{pmatrix} Q_1 & 0 \\ 0 & Q_2 \end{pmatrix}, D = \begin{pmatrix} D_1 & 0 \\ 0 & D_2 \end{pmatrix}, \text{ and } z = Q f.$$

The eigenvalues of T are interleaved with the diagonal entries of D and satisfy a simple secular equation obtained from the characteristic polynomial of \tilde{D} . The eigenvectors are then explicitly known but they must be expressed back in the original basis by pre-multiplying them by Q . The back transformation may even include the initial orthogonal transformation which reduces the original symmetric matrix to the tridiagonal form.

By p recursive tearings, the problem is reduced to 2^p independent eigenvalue problems of order $m = n/2^p$. Then, the technique described above is recursively applied in parallel to update the eigendecompositions. The total number of arithmetic operations is $O(n^3)$.

Parallel Sturm sequences. Sturm sequences are often used when only the part of the spectrum, in an interval $[a, b]$, is sought. Several authors discussed the parallel computation of Sturm sequences for a tridiagonal matrix T , and among them are Lo et al. in [26]. It is hard to parallelize the LU factorization of $(T - \lambda I)$ which provides the number of sign changes in the Sturm sequences, because the diagonal entries of U are obtained by a nonlinear recurrence. One

possible approach is to consider the linear three-term recursion which computes the Sturm sequence terms $p_k(\lambda) = \det(T_k - \lambda I_k)$, for $k = 0, \dots, n$. Since this computation corresponds to solving a triangular banded system of three diagonals, it can be parallelized, as for instance, by the algorithm described by Chen et al. in [10]. An analysis of the situation is given in [26]. Because the sequence $(p_k(\lambda))_{0,n}$ could suffer from overflow or underflow, Zhang proposed the computation of a scaled sequence in [33] and provided the backward error of the sequence.

To parallelize the computation, a more beneficial approach is to consider simultaneous computation of Sturm sequences by replacing bisection of intervals by multisection. However, multisections are efficient only when most of the created subintervals contain eigenvalues. Therefore, in [26] a strategy in two steps is proposed: (1) isolating all the eigenvalues with disjoint intervals, (2) extracting each eigenvalue from its interval. Multisections are used for step (1) and bisections or other root finders are used for step (2). This approach proved to be very efficient. When the eigenvectors are needed, they are computed independently by Inverse Iterations. A difficulty could arise if one wishes to compute all the eigenvectors corresponding to a cluster of very poorly separated eigenvalues.

Demmel et al. discussed in [11] the reliability of the Sturm sequence computation in floating point arithmetic where the sequence is no longer monotonic. Therefore, in very rare situations, it is possible to obtain a wrong answer with regular bisection. A robust algorithm, called MRRR developed by Dhillon et al. [13], is implemented via LAPACK with routine DSTEMR for the computation of high-quality eigenvectors. Several approaches are discussed in [13] for possible parallel implementations of the scheme in DSTEMR.

The Unsymmetric Eigenvalue Problem

Reduction to the Hessenberg form. Similar to the symmetric case, the reduced form is obtained by Householder reductions. But here, lack of symmetry implies that, unlike the picture in (10), zeros are not introduced in the upper part of the matrix. Therefore, the process results in an upper Hessenberg matrix. The total procedure involves $10n^3/3 + O(n^2)$ arithmetic operations. Assembling the matrix $Q = H_1 \dots H_{n-2}$ requires $4n^3/3 + O(n^2)$ additional operations.

Also, successive applications of Householder reductions can be compacted into blocks which allow use of Level 3 BLAS [6]. The algorithm is implemented via routine DGEHRD of LAPACK [2]. A parallel version of the algorithm is implemented via routine PDGEHRD of ScaLAPACK [5].

Parallel solution of the Hessenberg eigenproblem. Several attempts for generalizing the Divide-and-Conquer approach used in the symmetric case have been considered, including Dongarra and Sidani in [15], Adams and Arbenz in [1]. In these algorithms, the Hessenberg matrix is partitioned into a sum of a two-block upper-triangular matrix and a rank-one update. Compared to the symmetric case, there are two major drawbacks that limit the benefit of the tearing procedure: (1) the condition number of the eigenproblems defined by the two diagonal blocks of the first matrix can be much higher than that of the original eigenproblem as shown by Jessup in [24]. (2) the Newton iterations involved in the updating process include solving triangular or Hessenberg linear systems rendering the procedure rather more expensive than without tearing.

Henry et al. discussed in [21] existing approaches for realizing a parallel QR procedure and ended up considering an improved implicit multishifted strategy which allows the use of Level 3 BLAS [6]. The algorithm is implemented in subroutine PDLAHQR of ScaLAPACK [5].

The Singular-Value Problem

As mentioned above, methods for solving the symmetric eigenvalue problems can be reconsidered as methods for solving SVD problems. In this section, the study is restricted to the case where $A \in \mathbb{R}^{m \times n}$ with $m \geq n$.

Parallel Jacobi algorithms for SVD. In the special case where $m \gg n$, one should first perform an initial “skinny” QR factorization of $A = QR$ where $Q \in \mathbb{R}^{m \times n}$ is orthogonal and $R \in \mathbb{R}^{n \times n}$ is triangular. Next, the SVD of $R = \tilde{U}\Sigma V^T$ provides the SVD of $A = U\Sigma V^T$ with $U = Q\tilde{U}$. Computing the QR factorization in parallel has been outlined in [linear least squares and orthogonalization 12]. The complete process is obtained at the cost of $4mn^2 + O(mn + n^3)$ arithmetic operations.

One-sided and two-sided Jacobi algorithms. The first one-sided algorithm was introduced by Hestenes in [22]. Applying a rotation R to both sides of $B = A^T A$ for annihilating the entry β_{ij} is equivalent to post-multiply A by R for making the columns i and j of AR orthogonal. Therefore, any parallel Jacobi algorithm for solving a symmetric eigenvalue problem can be considered as a parallel one-sided Jacobi algorithm for solving a SVD problem. Each sweep is a sequence of orthogonal transformations V_k made of a set of independent rotations that are successively applied on the right side by $A_{k+1} = A_k \tilde{V}_k^T$ ($A_0 = A$). When the right singular vectors are needed, they are accumulated by $V_{k+1} = V_k \tilde{V}_k = (V_0 = I)$. At convergence, the singular values are given by the norms of the columns of A_k and the normalized columns are the corresponding left singular vectors. Each sweep performs $3mn^2 + O(mn)$ arithmetic operations. Brent and Luk [8] and Sameh [31] described possible implementations of this algorithm on multiprocessors.

Charlier et al. [9] demonstrate that an implementation of Kogbetliantz's algorithm for computing the SVD of upper-triangular matrices is quite effective on a systolic array of processors. Therefore, the first step in a Jacobi SVD scheme is the QR factorization of A , followed by computing the SVD of $\in \mathbb{R}^{n \times n}$. The Kogbetliantz's method for computing the SVD of a real square matrix A mirrors the method for computing the eigenpairs of symmetric matrices, in that the matrix A is reduced to the diagonal form by an infinite sequence of plane rotations

$$A_{k+1} = U_k A_k V_k^T, \quad k = 1, 2, \dots, \quad (11)$$

where $A_1 \equiv A$, and $V_k = V_k(i, j, \phi_{ij}^k)$, $U_k = U_k(i, j, \theta_{ij}^k)$ are plane rotations. It follows that A_k approaches the diagonal matrix $\Sigma = \text{diag}(\sigma_1, \sigma_2, \dots, \sigma_n)$, where σ_i is the i th singular value of A , and the products $(U_k \dots U_2 U_1)$, $(V_k \dots V_2 V_1)$ approach matrices whose i th column is the respective left and right singular vector corresponding to σ_i .

Block Jacobi algorithms. The above algorithms are well-suited for shared memory architectures. While they can also be implemented on distributed memory systems, their efficiency on such systems will suffer due to communication costs. In order to increase the granularity of the computation (i.e., to increase the number of floating

point operations between two communications), block algorithms are considered. For one-sided algorithms, each processor is allocated a block of columns instead of a single column. The computation remains the same as discussed above with the ordering of the rotations within a sweep is as given in [27].

For the two-sided version, the allocation manipulates 2-D blocks instead of single entries of the matrix. Bećka and Vajtešić propose in [4] a modification of the basic algorithm in which one annihilates, in each step, two off-diagonal blocks by performing a full SVD on a small-sized matrix. While reasonable efficiencies are realized on distributed memory systems, this block strategy increases the number of sweeps needed to achieve convergence.

We note that parallel Jacobi algorithms can only surpass the speed of the bidiagonalization schemes of ScaLAPACK when the number of processors available are much larger than the size of the matrix under consideration.

Bidiagonalization of a matrix by Householder reductions. The classical algorithm which reduces A into an upper-bidiagonal matrix B , via the orthogonal transformation $B = U^T A V$, was introduced by Golub and Kahan (see for instance [20]). The algorithm requires $4(mn^2 - n^3/3) + O(mn)$ arithmetic operations. U and V can be assembled in $4(mn^2 - n^3/3) + O(mn)$ and $4n^3/3 + O(n^2)$ additional operations, respectively. A parallel version of this algorithm is implemented in routine PDGEQR of ScaLAPACK.

The one-sided reduction for bidiagonalizing a matrix was proposed by Raha [29]. It appears to be better suited for parallel implementation. It consists of determining in a first step the orthogonal matrix V which would appear in the tridiagonalization of $A^T A$: $T = V^T A^T A V = B^T B$. The matrix V can be obtained by using Householder or Givens reductions, without first forming $A^T A$ explicitly. The second step performs a QR factorization of $F = AV$: $F = QR$. Since $F^T F = R^T R = T$ is tridiagonal, its Cholesky factor R is upper-bidiagonal which means that any two nonadjacent columns of F are orthogonal. This allows a simplified QR factorization. Bosner and Barlow introduced in [7] two adaptations of Raha's approach: a block version which allows use of Level 3 BLAS routines and a parallel version.

Once the bidiagonalization is performed, all that remains is to compute the SVD of B . When only

the singular values are sought, it can be done by the iterative Golub and Kahan algorithm where the number of arithmetic operations is $O(n)$ per iteration (see for instance [20]) which is performed only on uniprocessor. The whole Singular-Value Decomposition is implemented in routine PDGESVD of **ScalAPACK**.

Bibliographic Notes and Further Reading

Eigenvalue problems and singular-value decomposition: [28]

Algorithms: [3]

Linear algebra: [12, 20, 32]

Parallel algorithms for dense computations: [14, 18]

Bibliography

1. Adams L, Arbenz P (1994) Towards a divide and conquer algorithm for the real nonsymmetric eigenvalue problem. *SIAM J Matrix Anal Appl* 15(4):1333–1353
2. Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D (1999) LAPACK users' guide, 3rd edn. Society for Industrial and Applied Mathematics, Philadelphia, Library available at <http://www.netlib.org/lapack/>
3. Bai Z, Demmel J, Dongarra J, Ruhe A, van der Vorst H (eds) (2000) Templates for the solution of algebraic eigenvalue problems – a practical guide. SIAM, Software–Environments–Tools, Philadelphia
4. Bećka M, Vajtešić M (1999) Block-Jacobi SVD algorithms for distributed memory systems. Parallel algorithms and applications, Part I in 13, 265–267, Part II in 14, 37–56
5. Blackford LS, Choi J, Cleary A, D'Azevedo E, Demmel J, Dhillon I, Dongarra J, Hammarling S, Henry G, Petitet A, Stanley K, Walker D, Whaley RC (1986) ScaLA-PACK users' guide. Society for Industrial and Applied Mathematics, Philadelphia, Available on line at <http://www.netlib.org/lapack/lug/>
6. Blackford LS, Demmel J, Dongarra J, Duff I, Hammarling S, Henry G, Heroux M, Kaufman L, Lumsdaine A, Petitet A, Pozo R, Remington K, Whaley RC (2002) An updated set of basic linear algebra subprograms (BLAS). *ACM Trans Math Soft* 28:2–135–151
7. Bosner N, Barlow JL (2007) Block and parallel versions of one-sided bidiagonalization. *SIAM J Matrix Anal Appl* 29:927–953
8. Brent RP, Luk FT (1985) The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays. *SIAM J Sci Stat Comput* 6:69–84
9. Charlier J-P, Vanbegin M, Van Dooren P (1988) On efficient implementations of kogbetliantz's algorithm for computing the singular value decomposition. *Numer Math* 52:279–300
10. Chen SC, Kuck DJ, Sameh AH (1978) Practical parallel band triangular system solvers. *ACM Trans Math Software* 4:270–277
11. Demmel J, Dhillon I, Ren H (1995) On the correctness of some bisection-like parallel eigenvalue algorithms in floating point arithmetic. *ETNA* 3:116–149
12. Demmel J (1997) Applied numerical linear algebra. SIAM, Philadelphia
13. Dhillon D, Parlett B, Vömel C (2006) The design and implementation of the MRRII algorithm. *ACM Trans Math Software* 32:533–560
14. Dongarra JJ, Duff IS, Sorensen DC, Van der Vorst H (1998) Numerical linear algebra for high performance computers. SIAM, Philadelphia
15. Dongarra JJ, Sidani M (1993) A parallel algorithm for the nonsymmetric eigenvalue problem. *SIAM J Sci Comput* 14:542–569
16. Dongarra JJ, Sorensen DC (1987) A fully parallel algorithm for the symmetric eigenvalue problem. *SIAM J Sci Stat Comput* 8:s139–s154
17. Dongarra JJ, Van De Gein R (1996) Parallelizing the QR algorithm for the unsymmetric algebraic eigenvalue problem: myths and reality. *SIAM J Sci Comput* 17:870–883
18. Gallivan KA, Plemons RJ, Sameh AH (1990) Parallel algorithms for dense linear algebra computations. In: Gallivan KA, Heath MT, Ng E, Ortega JM, Peyton BW, Plemons RJ, Romine CH, Sameh AH, Voigt RG (eds) *Parallel algorithms computations*. SIAM, Philadelphia
19. Gimenez D, Hernandez V, van de Gein R, Vidal AM (1997) A Block Jacobi method on a mesh of processors. *Conc Pract Exp* 9–5:391–411
20. Golub GH, Van Loan CF (1996) *Matrix computations*, 3rd edn. Johns Hopkins University Press, Baltimore
21. Henry G, Watkins D, Dongarra J (2002) A parallel implementation of the nonsymmetric QR algorithm for distributed memory architectures. *SIAM J Sci Comput* 24:284–311
22. Hestenes MR (1958) Inversion of matrices by biorthogonalization and related results. *J Soc Ind Appl Math* 6:51–90
23. Jacobi CGJ (1846) Über ein Leiches Vehfahren Die in der theorie der Sacular-storungen Vorkom-mendern Gleichungen Numerisch Aufzulosen. *Crelle's J für reine und angewandte Mathematik* 30:51–94
24. Jessup ER (1993) A case against a divide and conquer approach to the nonsymmetric eigenvalue problem. *J Appl Num Math* 12: 403–420
25. Kagström B, Ruhe A (1980) An algorithm for numerical computation of the Jordan normal form of a complex matrix. *ACM Trans Math Software* 6:398–419
26. Lo SS, Philippe B, Sameh A (1987) A multiprocessor algorithm for the symmetric tridiagonal eigenvalue problem. *SIAM J Sci Stat Comput* 8:s155–s165
27. Luk F, Park H (1989) On parallel Jacobi orderings. *SIAM J Sci Stat Comput* 10:18–26
28. Parlett BN (1998) *The symmetric eigenvalue problem*. SIAM, Classics in Applied Mathematics, Philadelphia
29. Ralha R (2003) One-sided reduction to bidiagonal form. *Linear Algebra Appl* 358:219–238

30. Sameh A (1971) On Jacobi and Jacobi-like algorithms for a parallel computer. *Math Comp* 25:579–590
31. Sameh A (1985) Solving the linear least squares problem on a linear array of processors. In: Snyder L, Gannon DB, Siegel HJ (eds) Algorithmically specialized parallel computers. Academic, H.J. Siegel
32. Trefethen LN, Bau D III (1997) Numerical linear algebra. SIAM, Philadelphia
33. Zhang J (2003) The scaled sequence computation. In: Proceedings of the SIAM Conference on Applied Linear Algebra. July 15–19, 2003

EPIC Processors

DAVID I. AUGUST, ARUN RAMAN
Princeton University, Princeton, NJ, USA

Definition

Explicitly Parallel Instruction Computing (EPIC) refers to architectures in which features are provided to facilitate compiler enhancements of instruction-level parallelism (ILP) in all programs, while keeping hardware complexity relatively low. Using ILP-enhancement techniques such as speculation and predication, the compiler identifies the operations that can execute in parallel in each cycle and communicates a plan of execution to the hardware.

Discussion

Introduction

The performance of modern processors is dependent on their ability to execute multiple instructions per cycle. In the early 1990s, instruction-level parallelism (ILP) was the only viable approach to achieve higher performance without rewriting software. Programs for ILP processors are written in a sequential programming model, with the compiler and hardware being responsible for automatically extracting the parallelism in the program. *Explicitly Parallel Instruction Computing* (EPIC) refers to architectures in which features are provided to facilitate compiler enhancements of ILP in all programs. These architectures allow the compiler to generate an effective *plan of execution* (POE) of a given program, and provide mechanisms to communicate the compiler's POE to the hardware.

EPIC processors combine the relative hardware simplicity of *Very Long Instruction Word* (VLIW) processors and the runtime adaptability of *superscalar* processors. VLIW processors require the compiler to specify a POE consisting of a cycle-by-cycle description of independent operations in each functional unit, and the hardware follows the POE to execute the operations. While VLIW considerably simplifies the hardware, it also results in machine-code incompatibility between implementations of a VLIW architecture. Due to the strict adherence to the compiler's POE, code compiled for one implementation will not execute correctly on other implementations with different hardware resources or latencies. Superscalar processors discover ILP and construct a POE entirely in hardware. This means that code compiled for a particular superscalar architecture will execute correctly on all implementations of that architecture. However, ILP extraction in hardware entails significant complexity, resulting in diminishing performance returns and an adverse impact on clock rate. EPIC processors retain VLIW's philosophy of statically exposing ILP and constructing the POE, thereby keeping the hardware complexity relatively low, but incorporate the ability of superscalar processors to cope with dynamic factors such as variable memory latency. Combined with techniques to provide binary compatibility for correctness, EPIC processors provide a means to achieve high levels of ILP across diverse applications while reducing the hardware complexity needed for the same.

Design Principles

In EPIC architectures, the compiler is responsible for designing a POE. This means that the compiler must identify the inherent parallelism of a sequential program and map it efficiently to the parallel hardware resources in order to minimize the program's execution time. The main challenge faced by the compiler in determining a good POE is on account of dynamic events that determine the program's runtime behavior. Will a load-store pair access the same memory location? If yes, then the operations must be executed sequentially; otherwise, they may be scheduled in any order. Will a conditional branch be taken? If yes, then operations along the taken path may be scheduled in parallel with operations before the branch; otherwise, they must not be executed. In such situations, the compiler can

speculate that a load-store pair will not alias, or that a branch will be taken, with the compiler's confidence in the speculation typically being determined by the outcomes of profiling runs on representative inputs. EPIC architectures support compiler speculation by providing mechanisms in hardware to ensure correctness if speculation fails. Finally, the architecture must be rich enough for the compiler to express its POE; specifically, which operations are to be issued in parallel and which hardware resources they must use.

Section "ILP Enhancement" presents the major techniques used to enhance ILP of programs. Section "Enabling Architectural Features" describes the micro-architectural features that enable these and other ILP enhancement techniques.

ILP Enhancement

The code example in Fig. 1 is used to illustrate the main compiler transformations for ILP enhancement that are made possible by EPIC architectures. In the example, a condition is evaluated to alternatively update a memory location `p4` with the contents of location `p2` or `p3`. Outside the if-construct, the variable `val` is updated with the value at location `p5`, with the latter potentially aliasing with `p4`. The processor model assumed for illustration purposes, shown in Table 1, is a six-issue processor capable of executing one branch per cycle, with no further restrictions on the combination of operations that may be concurrently issued. Conditional branches require separate comparison and

EPIC Processors. Table 1 Assumed processor model to illustrate ILP enhancement techniques

Issue width	Six instructions
Latency of conditional branches	Two cycles
Latency of memory loads	Two cycles
Latency of other instructions	One cycle

```

if (*p1 == 0)
    *p4 = *p2;
else
    *p4 = *p3;
val += *p5;

```

EPIC Processors. Fig. 1 Example C code to illustrate ILP enhancement techniques

control transfer operations. All operations are assumed to have a latency of one cycle, with the exception of memory loads that have a latency of two cycles.

Figure 2a shows the scheduled, assembly code. The compiler conservatively respects all dependences, resulting in a rather sparse schedule with several slots going to waste. The execution time along either branch is ten cycles.

Speculation

Compiler-controlled speculation refers to breaking inherent programmatic dependences by guessing the outcome of a runtime event at compile time. As a result, the available ILP in the program is increased by reducing the height of long dependence chains and by increasing the scheduling freedom among the operations.

Control Speculation

Many general-purpose applications are branch-intensive. The latency to resolve a branch directly affects ILP since potentially multiple issue slots may go to waste. It is crucial to overlap other instructions with branches. Control speculation breaks control dependences which occur between branches and other operations. An operation is control dependent on a branch if the branch determines whether the operation will be executed at runtime. By guessing the direction of a branch, the control dependence may be broken effectively making the operation's execution independent of the branch. Control speculation allows operations to move across branches, thus reducing dependence height and resulting in a more compact schedule.

Data Speculation

Many general-purpose applications exhibit irregular memory access patterns. Often, a compiler is conservative because it respects a memory dependence that occurs infrequently or because it cannot determine that the dependence does not actually exist. In either case, data speculation breaks data flow dependences between memory operations. Two memory operations are flow dependent on one another if the first operation writes a value to a memory location and the second operation *potentially* reads from the same location. By guessing that the two memory operations will access different locations, the load operation may be hoisted above the

0	(1) r1 = MEM[p1]	
1		
2	(2) c1 = (r1 == 0)	
3	(3) JUMP c1, ELSE	
4	(4) r2 = MEM[p2]	
5		
6	(5) MEM[p4] = r2	(6) JUMP CONTINUE
ELSE:		
0	(7) r3 = MEM[p3]	
1		
2	(8) MEM[p4] = r3	
CONTINUE:		
0	(9) r5 = MEM[p5]	
1		
2	(10) r4 = r4 + r5	

E

0	(1) r1 = MEM[p1]	(4) r2 = MEM[p2] <CS>	(7) r3 = MEM[p3] <CS>	(9) r5 = MEM[p5] <DS>
1				
2	(2) c1 = (r1 == 0)			(10) r4 = r4 + r5 <DS>
3	(3) JUMP c1, ELSE			
4	(4') CHECK r2			
5	(5) MEM[p4] = r2			(6) JUMP CONTINUE
ELSE:				
0	(7') CHECK r3			
1	(8) MEM[p4] = r3			
CONTINUE:				
b 0	(9') CHECK r5			

0	(1) r1 = MEM[p1]	
1		
2	(11) pT, pF = (r1 == 0)	
3	(4) r2 = MEM[p2] <pT>	(7) r3 = MEM[p3] <pF>
4		
5	(5) MEM[p4] = r2 <pT>	(8) MEM[p4] = r3 <pF>
6	(9) r5 = MEM[p5]	
7		
c 8	(10) r4 = r4 + r5	

0	(1) r1 = MEM[p1]	(4) r2 = MEM[p2] <CS>	(7) r3 = MEM[p3] <CS>	(9) r5 = MEM[p5] <DS>
1				
2	(11) pT, pF = (r1 == 0)			(10) r4 = r4 + r5 <DS>
3	(4') CHECK r2 <pT>		(7') CHECK r3 <pF>	
4	(5) MEM[p4] = r2 <pT>		(8) MEM[p4] = r3 <pF>	
d 5				(9') CHECK r5

EPIC Processors. Fig. 2 Predication and speculation have a synergistic relationship: applying speculation after predication results in the best improvement in ILP. (a) Sequential schedule. (b) Schedule using control and data speculation alone. (c) Schedule using predication alone. (d) Schedule using predication and speculation combined

store. EPIC provides for more aggressive code motion: operations dependent on the load may also be hoisted above potentially aliasing stores.

Applying speculation to the code in Fig. 1 results in the tighter schedule shown in Fig. 2b. $\langle CS \rangle$ and $\langle DS \rangle$, respectively, denote operations which have been speculated with respect to control or data. The resultant increase in ILP is achieved primarily by applying speculation to the loads (operations 4, 7, and 9). Note that loads from either side of a branch may be speculatively executed. If the branch is highly biased, then the compiler may choose to speculatively execute just the loads from the biased path. Operations 4 and 7 are control dependent on operation 3. Control speculation enables the compiler to break the dependences and move 4 and 7 to the top of the block. Operation 9 is memory dependent on operations 5 and 8: these ambiguous memory dependences arise because the compiler is unable to prove that pointer p_5 does not point to the same location as pointer p_4 . The net result of applying speculation is that the dependence height of the code segment is reduced to seven cycles.

While speculation enhances ILP, it requires hardware support to handle exceptions. Exceptions generated by speculated operations can either be genuine, reflecting exception conditions present in the original code, or spurious, resulting from unnecessary execution of speculative operations. Since speculative operations, like ordinary operations, may cause long-latency exceptions such as page faults and TLB misses, time is wasted if spurious exceptions are repaired. Spurious exceptions are eliminated by taking exceptions only when the results of speculative operations are used nonspeculatively, since it is guaranteed at that point that the speculated code would have executed in the original program. A symbolic operation, called a *check*, is responsible for detecting any problems that occurred in a previous speculative execution. When an error is detected by a check instruction, either an exception is reported or repair is initiated. By positioning the check at the point of the original operation, the error detection and repair is guaranteed to occur only when the original operation would have been executed by a nonspeculated version of the program.

For data speculation, repair is necessary when an actual data dependence existed between the speculated load and one or more stores predicted to be independent at compile time. The check queries the hardware

to detect if a dependence actually existed for this execution and initiates repair if required. In general, all data-speculative and all potentially excepting control-speculative operations require checks. In Fig. 2b, operations 4', 7', and 9' are the previously discussed symbolic check operations.

Predication

Predicated execution is a mechanism that supports conditional execution of individual operations based on Boolean guards, which are implemented as predicate register values. A compiler converts control flow into predicates by applying *if-conversion* [1]. If-conversion translates conditional branches into predicate defining operations and guards operations along alternative paths of control under the computed predicates. A predicated operation is fetched regardless of its predicate value. An operation whose predicate is TRUE is executed normally. Conversely, an operation whose predicate is FALSE is prevented from modifying the processor state. With if-conversion, complex nests of branching code can be replaced by a straight-line sequence of predicated code. Predication increases ILP by allowing separate control flow paths to be overlapped and simultaneously executed in a single thread of control.

Figure 2c illustrates an if-conversion of the code segment from Fig. 1. The predicate for each operation is shown within angle brackets. For example, operation 4 is predicated on pT . The absence of a predicate indicates that the operation is always executed.

Predicates are computed using predicate define operations, such as operation 11. The predicate pT is set to 1 if the condition evaluates to true, and to 0 if the condition evaluates to false. The predicate pF is the complement of pT . Operations 4 and 7, and operations 5 and 8 are executed concurrently with the appropriate ones taking effect based on the predicate values. The net result of applying predication is that the dependence height of the code segment is reduced to nine cycles.

Combining Speculation and Predication

Both speculation and predication provide effective means to increase ILP. However, the example shows that their means of improving performance are fundamentally different. Speculation allows the compiler to break control and memory dependences, while predication allows the compiler to restructure program control flow

and to overlap separate execution paths. The problems attacked by both techniques often occur in conjunction; therefore, the techniques can be mutually beneficial.

Figure 2d illustrates the use of predication and speculation in combination. If-conversion removes the branch resulting in a stream of predicated operations. As before, data speculation breaks the dependences between operations 5 and 9, and operations 8 and 9, allowing the compiler to move operation 9 to the top of the block. Even though no branches remain in the code, control speculation is still useful to break dependences between predicate definitions and guarded instructions. In this example, the control dependences between operations 11 and 4, and operations 11 and 7 are eliminated by removing the predicates on operations 4 and 7. This form of speculation in predicated code is called *promotion*. As a result of this speculation, the compiler can hoist operations 4 and 7 to the top of the block to achieve a more compact schedule of height six cycles.

In summary, predication is only 11% faster and speculation is only 43% faster than the original code segment. As the example shows, speculation in the form of promotion can have a greater positive effect on performance after if-conversion than before. This synergistic relationship between predication and speculation yields a combined speedup of 67% over the original code segment.

Enabling Architectural Features

Since EPIC delegates the responsibilities of enhancing ILP and planning an execution schedule to the compiler, the architecture must provide features:

1. To overcome the worst impediments to a compiler's ability to enhance ILP, namely, frequent control transfers and ambiguous memory dependences
2. To specify an execution schedule

Speculation Support

As discussed in Section "ILP Enhancement", an architecture that supports speculation must provide mechanisms to detect potential exceptions on control-speculative operations as they occur, to record information about data-speculative memory accesses as they occur, and then to check at an appropriate time whether an exception should be taken or data-speculative repair should be initiated.

First, speculative operations must be distinguished from nonspeculative operations, since the latter should report exceptions immediately. Also, loads that have not been speculated with regard to data dependence need not interact with memory conflict detection hardware. For this, each operation that can be speculated has an extra bit, called the *S-bit*, associated with it. This bit is set on operations that are either control speculated or are data dependent on a data-speculative load. Each load has an additional bit, called the DS-bit. This bit is set only on data-speculative loads.

Second, to defer the reporting of exceptions arising from speculative operations until it is clear that the operation would have been executed in the original program, a 1-bit tag called the *NaT* (Not a Thing) bit is added to each register. When a speculative load causes an exception, the NaT bit is set instead of immediately raising the exception. Speculative instructions that use a register with its NaT bit set propagate the bit through their destination registers until a check operation determines whether speculation failed in which case the deferred exception is suppressed.

Third, a mechanism must be provided to store the source addresses of data-speculative loads until their independence with respect to intervening stores can be established. This functionality can be provided by the Memory Conflict Buffer (MCB) [2] or the Advanced Load Address Table (ALAT) [3]. The MCB temporarily associates the destination register number of a speculative load with the address from which the value in the register was speculatively loaded. Destination addresses of subsequent stores are checked against the addresses in the buffer to detect memory conflicts. The MCB is queried by explicit data speculation check instructions, which initiate recovery if a conflict was detected.

Predication Support

A set of predicate registers are used to store the results of compare operations. For example, the Intel Itanium processor family has a set of 64 (1-bit) predicate registers. An EPIC processor must also implement some form of predicate squash logic, which prevents instructions with false predicates from committing their results. The IMPACT EPIC architecture [4] also provides predication-aware bypass/interlock logic, which forwards results based on the predicate values associated with the generating instructions.

Static Scheduling Support

EPIC provides many features for enabling the compiler to specify a high-quality Plan Of Execution (POE). These features include the ability to specify multiple operations per instruction, the notion of architecturally visible latencies, and an architecturally visible register structure.

Multiple Operations per Instruction (MultiOp)

EPIC architectures allow the compiler to specify explicit information about independent operations in a program. A MultiOp is a specification of a set of operations that are issued simultaneously to be executed by the functional units. Exactly one MultiOp is issued per cycle of the *virtual time*. The virtual time is the schedule built by the compiler, and may differ from the actual time if the hardware inserts stalls due to dynamic events.

For example, the Itanium architecture [3] consists of *instruction groups* that have no read-after-write (RAW) or write-after-read (WAR) register dependences, delimited by *stops*. The Itanium implementation consists of 128-bit instruction bundles comprising three 41-bit opcodes and a 5-bit template that includes intra-bundle stop bits. The use of template bits greatly simplifies instruction decoding. The use of stops enables explicit specification of parallelism.

Architecturally Visible Latencies

A lot of the hardware complexity of a superscalar processor arises from the need to maintain an illusion of sequential execution (i.e., each operation completes before another begins) in the face of non-atomicity of the operations. To avoid the complexity, EPIC processors expose operations as being nonatomic; instead, read-and-write events of an operation are viewed as the atomic events. There is a contractual specification of *assumed latencies* of operations that must be honored by both the hardware and the compiler. A MultiOp that takes more than one cycle to generate all the results is said to have *non-unit assumed latency* (NUAL). The guarantee of NUAL affords multiple benefits to EPIC processors. Since the hardware is certain that the read event of an operation will not occur before the write event of the operation that produces the value to be read, there is no need for stall/interlock capability. Since the compiler is certain that an operation will not write its result before its assumed latency has elapsed, it

can schedule other operations that are anti- or output-dependent on the operation in question, resulting in tighter schedules.

As mentioned earlier, EPIC processors may still have interlocking to account for the deviation of actual latencies of operations from their assumed latencies. However, MultiOp and NUAL permit an in-order implementation of the architecture while still extracting substantial ILP.

Register Structure

ILP processors require a large number of registers. Specifically in EPIC processors, techniques such as control and data speculation cause increased register pressure. Fundamentally, since multiple operations are executing in parallel, there is a demand for a larger store for temporary values. Superscalar processors architecturally expose only a small fraction of the physical registers, and rely on register renaming to generate parallel schedules. However, this requires complex hardware for dynamic scheduling. Since EPIC demands a schedule from the compiler in order to simplify the hardware, it becomes necessary to expose a larger set of architectural registers to generate the same parallel schedule as achieved through hardware register renaming.

EPIC architectures also provide a mechanism to rotate the register file to speed up inner-loop execution. Speeding up loops involves executing instructions from multiple loop iterations in parallel. One way to do this is by using *Modulo Scheduling* [5]. Modulo scheduling schedules one iteration of the loop such that when this schedule is repeated at regular intervals, no intra- or inter- iteration dependence is violated, and no resource conflict arises between the dynamic operations within and across loop iterations. While generating code, it is important that false dependences do not arise owing to dynamic instances on different iterations of the same operation writing to the same registers. While the loop could be unrolled, followed by static register renaming, this typically results in substantial code growth. EPIC architectures implement a *rotating register file* that provides register renaming such that successive writes to the same virtual register actually go to different physical registers. This is accomplished through the use of a *rotating register base* (RRB) register. The physical register number accessed by each operation is the sum of the virtual register number specified in the instruction

and the number in the RRB register, modulo the number of registers in the rotating register file. The RRB is updated at the end of each iteration using special operations. In this manner, EPIC architectures provide compiler-controlled dynamic register renaming.

Another interesting feature is the *register stack* that is used to simplify and speed up function calls. Instead of spilling and filling all the registers at a procedure call and return sites, EPIC processors enable the compiler to rename the virtual register identifiers. The physical register accessed is determined by renaming the virtual register identifier in the instruction through a base register. The callee gets a physical register frame that may overlap with the caller's frame; the overlap constitutes the registers that are passed as parameters.

Cache Management

EPIC architectures allow the compiler to explicitly control data movement through the cache hierarchy by exposing the hierarchy architecturally. Generating a high-quality schedule often requires the compiler to be able to accurately predict the level in the cache hierarchy where a load operation will find its data. This is normally achieved by profiling or analytical means, and the source-cache specifier is encoded as part of the load instruction. This informs the hardware of the expected location of the datum as well as the assumed latency of the load.

EPIC architectures also provide a target cache specifier for load/store operations that allow a compiler to instruct the hardware of where the datum should be put in the cache hierarchy for subsequent references to that datum. This allows the compiler to explicitly control the content of the caches. The compiler could, for example, prevent pollution of the caches by excluding data with poor temporal locality from the higher-level caches, and remove data from the last level cache after the last use.

Many streaming programs operate on data without accessing them again. To improve the latency of such accesses, EPIC architectures provide a *data prefetch cache*. Prefetch load instructions can be used to prefetch data with poor temporal locality into the data prefetch cache. This prevents the displacement of first-level data cache contents that may potentially have much better temporal locality.

Related Entries

- [Cydra 5](#)
- [Dependences](#)
- [Instruction-Level Parallelism](#)
- [Modulo Scheduling and Loop Pipelining](#)
- [Multiflow Computer](#)
- [Parallelization, Automatic](#)

Bibliographic Notes and Further Reading

The EPIC philosophy was primarily developed by HP researchers in the early 1990s [6, 7]. As mentioned in the introduction, the EPIC philosophy was fundamentally influenced by the VLIW philosophy developed in the 1980s in the Multiflow and Cydrome processors [8, 9]. EPIC inherited various features from Cydrome, most significantly:

- Predicated execution
- Support for software pipelining of loops in the form of rotating register files
- A rudimentary MultiOp instruction format
- Hardware support for speculation including speculative opcodes, a speculative tag bit in each register, and deferred exception handling

Many of the ideas on speculation and its hardware support were developed independently by researchers in the IMPACT group at the University of Illinois [10] and IBM Research [11]. The most famous commercial EPIC processor is the Itanium processor family jointly developed by HP and Intel. The first processor in the family was released in 2001, with successors having improved memory subsystems, multiple cores on the same die, and improved power efficiency.

More information on the EPIC philosophy and implementation can be found in the technical reports and manuals on the topic [3, 6].

Bibliography

1. Allen JR, Kennedy K, Porterfield C, Warren J (1983) Conversion of control dependence to data dependence. In: Proceedings of the 10th ACM Symposium on Principles of Programming Languages, pp 177–189, January 1983
2. Gallagher DM, Chen WY, Mahlke SA, Gyllenhaal JC, Hwu WW (1994) Dynamic memory disambiguation using the memory conflict buffer. In: Proceedings of 6th International Conference on

- Architectural Support for Programming Languages and Operating Systems, pp 183–193, October 1994
3. Intel Corporation (2006) Intel Itanium architecture software developer's manual. Application architecture, vol 1, Revision 2.2. Santa Clara, CA
 4. August DI, Connors DA, Mahlke SA, Sias JW, Crozier KM, Cheng B, Eaton PR, Olaniran QB, Hwu WW (1998) Integrated predication and speculative execution in the IMPACT EPIC architecture. In: Proceedings of the 25th International Symposium on Computer Architecture, pp 227–237, June 1998
 5. Rau BR (1994) Iterative modulo scheduling: an algorithm for software pipelining loops. In: Proceedings of the 27th International Symposium on Microarchitecture, pp 63–74, December 1994
 6. Schlansker MS, Rau BR (2000) “EPIC: An architecture for instruction-level parallel processors,” Technical Report HPL-1999-111, Compiler and Architecture Research HP Laboratories Palo Alto, February 2000
 7. Schlansker MS, Rau BR (2000) EPIC: Explicitly Parallel Instruction Computing. Computer 33(2):37–45
 8. Fisher JA (1983) Very long instruction word architectures and the ELI-512. In: ISCA '83: Proceedings of the 10th Annual International Symposium on Computer Architecture, ACM, New York, pp 140–150
 9. Beck GR, Yen DW, Anderson TL (1993) The Cydra 5 minisupercomputer: architecture and implementation. J Supercomput 7:143–180
 10. Mahlke SA, Chen WY, Hwu WW, Rau BR, Schlansker MS (1992) Sentinel scheduling for VLIW and superscalar processors. In: Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, pp 238–247, October 1992
 11. Ebcioiglu K (1988) Some design ideas for a VLIW architecture for sequential-natured software. In: Cosnard M, Barton MH, Vanneschi M (eds) Parallel Processing (Proc. IFIP WG 10.3 Working Conference on Parallel Processing, Pisa, Italy), pp 3–21, North Holland, 1988

developed, mainly by W. Händler. Such a system is a collection of nodes where each node consists of one processor and one shared memory unit at least. The nodes are organized in several hierarchically ordered planes (layers) and one top-node. Each plane is a collection of m^2 identical nodes arranged in an $m \times m$ grid, where each node is connected to its four nearest neighbors via shared memory. At the border of the plane, the nodes are interconnected in such a way that the layer forms a torus. Each upper plane is smaller than the layer immediately below. It has a quarter of the nodes of the lower plane. Each node, which is not part of the lowest plane, has access to the shared memory of four child nodes of the layer immediately below. The base layer, called working plane, is that part of the system where most of the real work is to be done. The upper layers and the top-node feed the working plane with data and offer support functions. In larger systems, only some of the lower planes are realized. In this case, the top-node is connected to the highest realized plane via bus.

A special EGPA system is MEMSY. In such a system, only the two lower planes and the top-node are realized. Furthermore, by adding a bus system, long distance communication via messages is possible. Consequently, several programming models, not only shared memory, are supported.

From 1978 to 1994, three such systems of the EGPA class have been realized: the Pilot Pyramid (4+1 nodes), the DIRMU Pyramid, and the MEMSY Prototype (16 + 4 + 1 nodes both).

Discussion

Introduction

In 1974, W. Händler published first ideas [A] and proposed one year later together with F. Hofmann and H.J. Schneider a new subclass of multiprocessors [16]: the Erlangen General Purpose Array (EGPA, Erlangen is an university city near Nürnberg in Bavaria). (Many of the following text is taken from [14, 15, 18].) The essential features and design objectives of these MIMD-computers are:

1. Homogeneity: There is only one identical type of element – the node often called PMM (processor-memory-module).
2. Memory-coupling: Connection between nodes is via memory.

Erlangen General Purpose Array (EGPA)

JENS VOLKERT
Johannes Kepler University Linz, Linz, Austria

Synonyms

MEMSY

Definition

During the 1970s, the concept of the multiprocessor class Erlangen General Purpose Array (EGPA) was

3. Restricted neighborhood: No node has access to the whole memory, and no memory block can be accessed by all nodes.
4. Hierarchy: The PMMs are arranged in several layers and these layers form a total order.
5. Regularity: The nodes of one layer are ordered like a grid.

W. Händler explained these features as follows: The reasons for these demands are speed (memory coupling), programmability (regularity, homogeneity), control (hierarchy), technical limits (restricted neighborhood), and expandability (all).

In the following years, several EGPA systems were built. The first one, the pilot pyramid, was assembled in 1978 and was used until 1984. It consisted of four PMMs at the working plane level 0 (A-processors) and one PMM at level 1 (B-processor).

From 1985 to 1988, DIRMU (Distributed Reconfigurable Multiprocessor Kit [4]) was used for testing applications written for EGPA systems. DIRMU was a kit consisting of PMMs, 8086 based, where processor-ports of one PMM could be interconnected with memory-ports of other PMMs by cables. Using several of these PMMs, different topologies could be realized: rings, trees, and so on. Especially an EGPA pyramid with three layers (16 + 4 + 1 PMMs) could be built.

From 1989 to 1994, a new EGPA system started running in Erlangen. It was called MEMSY (Modular Expandable Multiprocessor System). In this multiprocessor, three layers were realized. It consisted of 16 nodes in the working plane and 4 nodes in the next layer.

These machines were used to demonstrate the feasibility and usefulness of EGPA systems.

In the following, the principles of EGPA systems are discussed, and the pilot pyramid and MEMSY serve as examples.

Motivation

EGPA systems are general purpose machines, but they were especially designed for numerical simulations. With the emergence of powerful computer systems, the interest in computational science was awakened. This computational approach in natural science and engineering sciences was and is very successful in context with fluid dynamics, condensed matter physics, plasma

physics, applied geophysics, and others. The corresponding models of physical phenomena are either continuum models or many-body models.

Essential is that there is an inherent “locality” in these approaches. The designer of EGPA kept this fact in mind when they developed their machines according to the possibilities the computer technology offered at that time, the 1980s.

E

Design Goals

The EGPA architecture was defined with the following design goals in mind:

Scalability

The architecture should be scalable with no theoretical limit. The communication network should grow with the number of processing elements in order to accommodate the increased communication demands in larger systems.

Portability

Algorithms should run on EGPA systems of any size.

Cost effectiveness

As far as possible, off-the-shelf components should be used.

Flexibility

The architecture should be usable for a great variety of user problems.

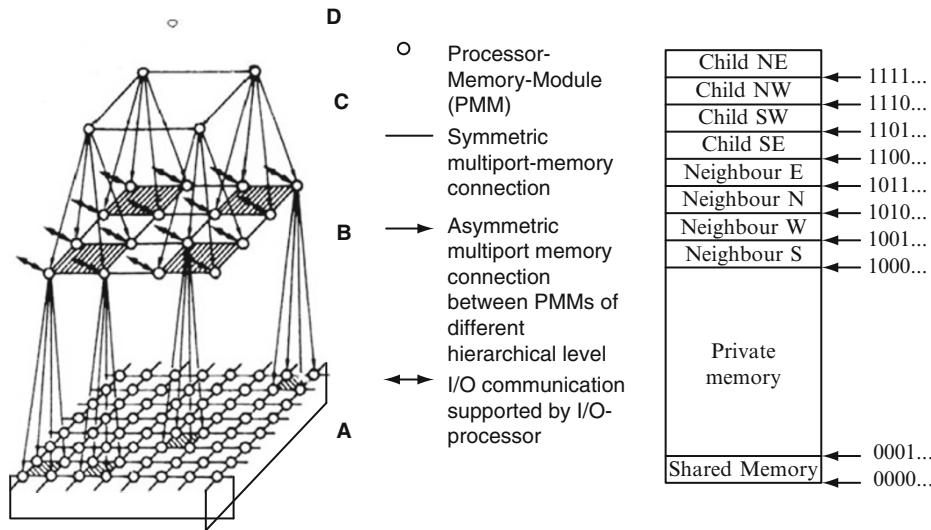
Efficiency

The computing power of the system should be big enough to handle real problems which occur in scientific research.

System Topology

An EGPA system consists of identical PMMs (nodes): one processing unit and one attached communication (multiport) memory form one PMM (Processor Memory Module). The processing element consists of one or more processors. Optionally, there can exist a coprocessor and/or a private memory only accessible by the processing unit itself.

The PMMs are hierarchically arranged in several layers (planes). An example with four layers (A, B, C, D) is given in Fig. 1a. At the topmost layer, there is only a single PMM which serves as a front-end to the system. At each lower layer, each PMM is connected to exactly four neighbors at the same layer, e.g., each processor has access to the communication memories of its neighboring PMMs. Thus, at each plane, the hardware has a



Erlangen General Purpose Array (EGPA). Fig. 1 (a) EGPA with 64 PMMs in the working plane (b) address space of a node

grid-like structure. As in each row, the most left node is connected to the most right PMM, and in each column, the corresponding nodes are connected too, each layer forms a torus. (In Fig. 1a, for clarity, only two of these turn-around connections are drawn.) In addition to the horizontal accesses, each node, except the very lowest, has access to four PMMs at the next lower layer. Consequently, the address space of a processor looks like in Fig. 1b. Each access with an address lying in the address space part of a child or neighbor will be translated by hardware to a shared memory address of that child or neighbor and that location of the child or neighbor will be accessed (e.g., the address 1010 01... causes an access to the cell 0000 01... of the neighbor in the north).

The vertical connections are equipped to broadcast data downward to all four of the lower PMMs simultaneously, say, to transmit code segments. On the other hand, though the lower nodes do not have memory access to those at higher layers, they are able to interrupt their supervising PMM. These substructures, consisting of four nodes and their supervisor, are called elementary pyramid. Several of them are highlighted in Fig. 1a. Each elementary pyramid has an I/O processor with corresponding I/O devices, mainly disks. The I/O processor which has access to all of its elementary pyramid memories is controlled by the supervising PMM. The topmost PMM is connected to a network containing a software development system.

The overall arrangement of an EGPA system is pyramidal. Each pyramid can be extended downward to any size by adding new levels. Such an extension significantly increases the computing power of the system. An EGPA system is a distributed memory multiprocessor. Remote communication between nodes is done via chains of copies and not via messages as usual.

System Software

Each node has its own local operating system that exactly renders those services the resources of which are available. Now, in case a process needs a system service, this service either will be rendered at once or the addressed operating system cannot perform it and now informs his superior processor, i.e., its operating system, of the order.

In the pilot pyramid, this technique was realized as follows. For each A-processor, a shadow process existed in the B-processor. At the four A-processors, only a user interface to the operating system of the B-processor was implemented. If there was an order to the operating system of the B-processor, this interface communicated with its shadow process. Then this shadow process sent the request to the proper operating system which executed the desired function afterward.

This approach can be extended to larger EGPA computer; since each such system is composed of elementary pyramids (see Fig. 1a). Each B-processor is

part of an elementary pyramid one level higher and so on. Consequently, the technique of shadow processes could be used in upper elementary pyramid as well.

The proposed user interface for programming such a multiprocessor was in addition to several tools for testing parallel programs the EGPA monitor. This monitor consists of an EGPA frame program and a pool of subroutines. If a parallel program is executed, the frame program runs first installing the necessary system processes and then starting the user routines. The subroutines of the monitor are tools for controlling user programs. They allow to distribute programs and data to the system, to control the processes and to execute coordinating tasks (see [25]). Two very important tools are EXECUTE (PROGNAME, PROC) (starts routine PROGNAME at the indicated processor – child or neighbor) and WAIT (PROGNAME, PROC) (wait until the routine has finished at the processor).

The EGPA monitor was realized for the Pilot Pyramid. But the concept can be carried to larger systems. It supports only one parallel program at a time in contrast to MEMSOS (see later) used in the MEMSY pilot.

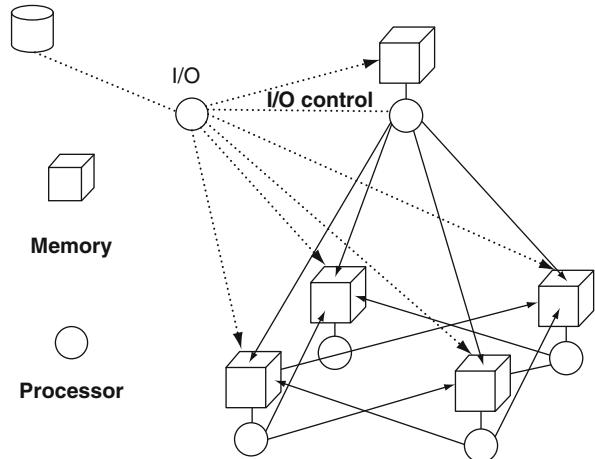
The Pilot Pyramid

The pilot pyramid, equivalent to an elementary pyramid, was assembled in 1978 and was used until 1984 (see Fig. 2). It consisted of four PMMs at the working plane level 0 (A-processors) and 1 PMM at level 1 (B-processor). All nodes were commercially available computers AEG 80/60 which offered a multiport memory. The characteristics of these processors were internal processor cycles of 80, 120, and 240 ns and a word length of 32 bits. The access time to a word was 1–2 μ s. Each A-processor had a memory of 128 KB and the B-processor 512 KB. For more details, see [20].

The operating system was an expansion of the original available operating system MARTOS. The EGPA monitor could be called from user programs written in FORTRAN or SL3 (a subset of ALGOL68).

Application Programming with the EGPA Monitor

EGPA was designed for mainly solving problems using data partitioning. The corresponding programming model is known as SPMD (Single Program Multi Data).



Erlangen General Purpose Array (EGPA). Fig. 2 Pilot Pyramid

Before implementing a parallel program, the user has to partition his problem domain in such a way that parallel work can be done as much as possible. Furthermore, he has to assign himself the data to the processing elements and, in contrast to a global shared memory systems, in EGPA, the user had to take into account that needed data should not only exist within the system but also in the right memory at the right time. The synchronization and the control of data were done by using the EGPA monitor. The allocation of data was effected by segments, and most of data transports happened under user control. There were two possibilities for synchronization: under system control using messages or under user control by special routines of the monitor. The latter was a factor thousand faster, but deadlocks were possible if the user made a mistake. Nevertheless, it was preferred by nearly all users and the gained speedups were much better.

One very important area for using EGPA systems are partial differential equations. Under the assumption that the region of interest is a square (or a cube) represented by a grid, for numerical purposes, and a grid-point-wise relaxation method is performed in red black order, the partitioning is very simple. To each processor one part is assigned. Each processor handles its portion of the grid and needs only values produced by neighboring processors. In Fig. 3a, the access area of one processor is hatched.

On an EGPA system, one iteration step performed at a processor p is as follows:

1. Relax red points which do not depend on points of any neighbor.
2. Wait until all neighbors have signaled that their values can be used, e.g., the western neighbor has set the variable “ready” to iteration number (p accesses this variable using the name “ready&western”).
3. Relax red points which depend on black points of neighbors. These are points at the border of p.
4. Relax black points.
5. Ready := Iteration number +1; /* signal to neighbors own borders are ready for next iteration.

For this simple case, speedups near the number of processors in the working layer were achieved. If it was possible to map a rectangle onto a non-square physical space (see, e.g., Fig. 3b), the results were equivalent. If the domain under investigation is not so optimal, the programmer had to search for other solutions. If, e.g., the domain is a circle, it is not possible to give all nodes the same load and consequently, the efficiency of the system is smaller. Sometimes it helps to divide the region into several blocks. Then the grid is so folded onto the multiprocessor that each processor operates on a part of several blocks and each block is calculated by several processors simultaneously (Fig. 3c). Thereby, the torus structure of the layer is necessary. In Fig. 3d, this is illustrated for a circle. Each square is distributed to all nodes of the working plane.

In many cases, the application designer has to search for new algorithm to use the power of the system. Thereby, he has to take into account the interconnection structure of an EGPA system. Matrix multiplication ($H = F \cdot G$) with $m \times m$ nodes in the working

plane shall serve as an example. At the first glance, the interconnection topology of an EGPA seems to be a serious restriction for calculation of any new element, since data stored in remote memories (in the sense of restricted neighborhoods) are needed and have to be transported using the communication memories. The following algorithm is based on an idea of Cannon [2].

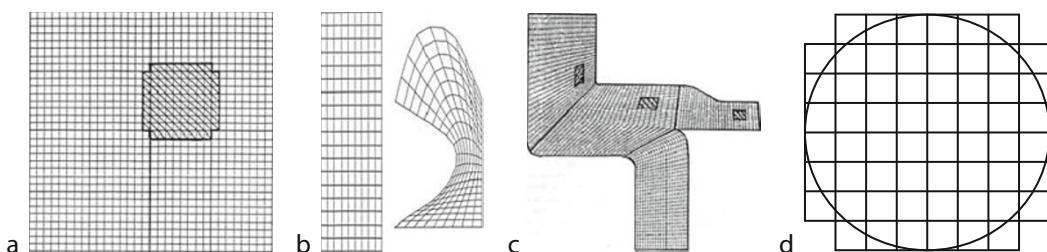
At the beginning, all matrices are equally distributed over the lowest plan as well as possible and H is set to 0. H_{ij} , F_{ij} , and G_{ij} are stored in node p_{ij} (see Fig. 4 Start position for $m = 4$). The start position is transformed by more or less rotation of the partial matrices F_{ij} and G_{ij} in rows respectively columns of the working plane until the data are distributed as depicted in Fig. 4 step 1 for $m = 4$. For example, p_{34} reads G_{44} and copies it into the communications memory of processor p_{24} , this data is read and written into its own shared memory by p_{14} . These data transports can be done in parallel by the hardware. Thereby, the toroidal connections are used.

Step 1: Each processor p_{ij} multiplies the partial matrices F_{ik} and G_{kj} stored at its memory at the beginning of step i and adds the resulting matrix to H_{ij} . Subsequently, the processor writes the matrix F_{ik} into the memory of the processor “west” of it and the matrix G_{ij} into the memory of the processor “north” of it.

After m steps, the result H is distributed on the working plane as shown in Fig. 4. Result:

Parts of the program for demonstration purposes written in SL3 (subset of ALGOL68).

It is one SPMD program per layer: The same code runs at each processor of one layer. For that reason, relative names with the following syntax were used.



Erlangen General Purpose Array (EGPA). Fig. 3 Data grid partitioning for pointwise relaxation method

Start position:	Step 1:	Step 2:
F ₁₁ G ₁₁ F ₁₂ G ₁₂ F ₁₃ G ₁₃ F ₁₄ G ₁₄	F ₁₁ G ₁₁ F ₁₂ G ₂₂ F ₁₃ G ₃₃ F ₁₄ G ₄₄	F ₁₂ G ₂₁ F ₁₃ G ₃₂ F ₁₄ G ₄₃ F ₁₁ G ₁₄
F ₂₁ G ₂₁ F ₂₂ G ₂₂ F ₂₃ G ₂₃ F ₂₄ G ₂₄	F ₂₂ G ₂₁ F ₂₃ G ₃₂ F ₂₄ G ₄₃ F ₂₁ G ₁₄	F ₂₃ G ₃₁ F ₂₄ G ₄₂ F ₂₁ G ₁₃ F ₂₃ G ₂₄
F ₃₁ G ₃₁ F ₃₂ G ₃₂ F ₃₃ G ₃₃ F ₃₄ G ₃₄	F ₃₃ G ₃₁ F ₃₄ G ₄₂ F ₃₁ G ₁₃ F ₃₂ G ₂₄	F ₃₄ G ₄₁ F ₃₁ G ₁₂ F ₃₂ G ₂₃ F ₃₃ G ₃₄
F ₄₁ G ₄₁ F ₄₂ G ₄₂ F ₄₃ G ₄₃ F ₄₄ G ₄₄	F ₄₄ G ₄₁ F ₄₁ G ₁₂ F ₄₂ G ₂₃ F ₄₃ G ₃₄	F ₄₁ G ₁₁ F ₄₂ G ₂₂ F ₄₃ G ₃₃ F ₄₄ G ₄₄
Step 3:	Step 4:	Result:
F ₁₃ G ₃₁ F ₁₄ G ₄₂ F ₁₁ G ₁₃ F ₁₂ G ₂₄	F ₁₄ G ₄₁ F ₁₁ G ₁₂ F ₁₂ G ₂₃ F ₁₃ G ₃₄	H ₁₁ H ₁₂ H ₁₃ H ₁₄
F ₂₄ G ₄₁ F ₂₁ G ₁₂ F ₂₂ G ₂₃ F ₂₃ G ₃₄	F ₂₁ G ₁₁ F ₂₂ G ₂₂ F ₂₃ G ₃₃ F ₂₄ G ₄₄	H ₂₁ H ₂₂ H ₂₃ H ₂₄
F ₃₁ G ₁₁ F ₃₂ G ₂₂ F ₃₃ G ₃₃ F ₃₄ G ₄₄	F ₃₂ G ₂₁ F ₃₃ G ₃₂ F ₃₄ G ₄₃ F ₃₁ G ₁₄	H ₃₁ H ₃₂ H ₃₃ H ₃₄
F ₄₂ G ₂₁ F ₄₃ G ₃₂ F ₄₄ G ₄₃ F ₄₁ G ₁₄	F ₄₃ G ₃₁ F ₄₄ G ₄₂ F ₄₁ G ₁₃ F ₄₂ G ₂₄	H ₄₁ H ₄₂ H ₄₃ H ₄₄

Erlangen General Purpose Array (EGPA). Fig. 4 Matrix multiplication for m = 4 = 16 nodes: partial matrix distribution

```
<relative name> ::= <name>&<neighbour> |
    <name>&<child>
<neighbour> ::= NORTH | WEST | SOUTH | EAST /* the 4
    neighbours
<child> ::= NORTHWEST | NORTHEAST | SOUTHWEST |
    SOUTHNORTH /* the 4 children */
```

Allocation of data at each A-processor:

```
/* installation of a global (shared)
   segment with name "MATRIX" in
   its own shared memory */
&SEG (MATRIX,GLOBAL);
/* consequently there exist segments
   "MATRIX" installed in the shared
   memory of all its neighbour. It
   needs only access to those
   segments of its western and
   northern neighbour. Therefore*/
&SEG (MATRIX&WEST,GLOBAL);
    &SEG (MATRIX&NORTH,GLOBAL);
/* declaration part */
GLOBAL (MATRIX) [] REAL FMATRIX, GMATRIX,
    HMATRIX, INT FSEM, GSEM;
VALGLOB (MATRIX&WEST) REF [] REAL
    FMATRIX&WEST, REF INT FSEM&WEST;
VALGLOB (MATRIX&NORTH) REF [] REAL
    GMATRIX&NORTH, REF INT
    GSEM&NORTH;
```

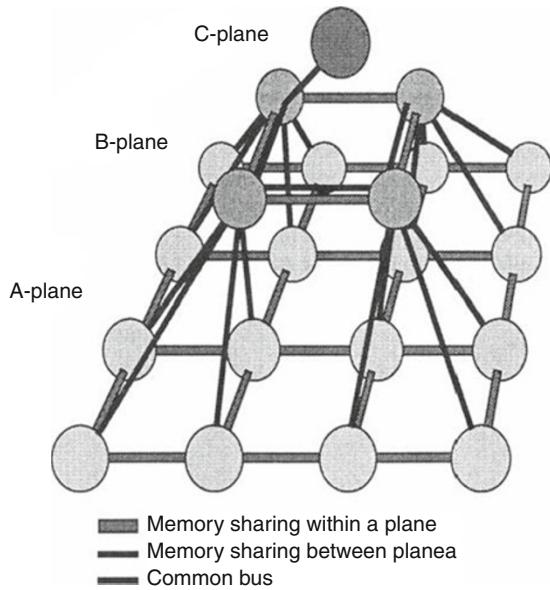
Essential part of the main program running at each A-processor:

```
/* P x P is the number of A-processors */
P is the FSEM := GSEM := 1;
FOR STEP FROM 1 TO m
DO /* FSEM = STEP and GSEM = STEP are
```

```
indicators that the data needed
for STEP i are already stored */
WHILE FSEM/= STEP OR GSEM/= STEP
    DO WAIT (T) OD; /* T for
        avoiding too much memory accesses,
        if coherent caches are used - not
        in the Pilot Pyramid, WAIT is not
        necessary */
    MULT (HMATRIX, FMATRIX, GMATRIX)
        /* local matrix multiplication
           followed by addition of the
           result to HMATRIX */
    FSEM := GSEM := -STEP /* indicator
        that the data are used and can be
        overwritten */
    WHILE FSEM& WEST/= -STEP
        DO WAIT(T) OD;
            /* western neighbour has
               not finished */
    COPY (FMATRIX, FMATRIX&WEST);
        /* subroutine which copies
           FMATRIX into FMATRIX of the
           western neighbour.*/
    FSEM&WEST := STEP + 1; /* indicator
        that data is transported */
    WHILE GSEM&NORTH/= -STEP
        DO WAIT (T) OD;
    COPY (GMATRIX, GMATRIX&NORTH);
    GSEM&NORTH := STEP + 1
```

OD

The codes for gaining the data distribution at the beginning of Step 1 (see Fig. 4) as well as for COPY and for MULT are not represented. These routines need the size of the matrices which depend on the position of



Erlangen General Purpose Array (EGPA). Fig. 5 MEMSY prototype

the processor. Therefore, the EGPA monitor provides each processor with its position (column, row) in the working plane together with the value of m .

MEMSY

One essential result of all investigations concerning EGPA was the following. For many kinds of computations, the main computing load was carried by the lowest layer. The higher levels were mainly used for distributing partial tasks and data and for the realization of communication mechanisms. The consequence was that the higher layers were left underutilized. For removing this problem, the first idea was to taper the pyramid so that each lower level in an elementary pyramid has 9 or even 16, rather than merely four PMMs. Experiences with a test system showed that excessive tapering (more than nine nodes in the elementary pyramid) can lead to bottlenecks at the higher layers. During these investigations, the best solution was found: The plane B must be mighty enough (elementary pyramid with five nodes) that the upper levels can be removed if the topmost node is connected to the B-plane by a bus. Such a special EGPA system is called MEMSY.

A prototype with 16 nodes in the A-plane and 4 nodes in the B-plane had been realized. The theoretical peak performance was 2 GMIPS or 960 MFLOPS (double precision), the measured performance was up to 500 MFLOPS [3, 4]. (Most of the following information presented in this chapter is taken from [21].)

The node structure of MEMSY differed from that one of the pilot pyramid. Each node consisted of a Motorola multiprocessor board MVEME188 (4 MC88100, 25 MHz clock rate) and additional hardware. It contained a local memory, an attached shared memory, and an I/O-bus with access to different devices.

The programming model of MEMSY was defined as a set of library calls which could be called from C and C++. In contrast to the EGPA model, it was not only based on the shared memory paradigm of EGPA. Instead, the model defined a variety of different mechanisms for communication and coordination. From these mechanisms, the application programmer could pick those calls which were best suited for his particular problem.

The use of the shared memory was based on the concept of “segments,” very much like the original shared memory mechanism provided by UNIX System V. A process which wants to share data with another process (possibly on another node) first has to create a shared memory segment of the needed size. To have the operating system, select the correct location for this memory segment; the process has to specify with which neighboring nodes this segment needs to be shared. After the segment has been created, other processes may map the same segment into their address space by the means of an “attach” operation. The segments may also be unmapped and destroyed dynamically.

For coordination purposes, three mechanisms had been provided: short (two word) messages, semaphores, and spinlocks. There was a second message type for transports of high-volume and fast data transfer between any two processors of the system.

To express parallelism within a node, the programmer had to create multiple processes on each processing element by a special variant of the system call “fork.” Parallelism between nodes was handled by a configuration description, which forced the application environment to start one initial process on each node that the application should run on. In this starting phase, the application was consistent with the SPMD

model. Through the execution of fork-calls, a configuration could be established which is beyond this model.

The operating system of the MEMSY prototype was MEMSOS, it was based on UNIX V/88 Release 3 of Motorola, which was adapted to the multiprocessor architecture of the processor board [20].

In MEMSOS, most of the users' needs were supported by the implementation of the “*application concept*.” An application was defined as the set of all processes belonging to one single user program. A process, belonging to an application, could be identified by (a) a globally unique application id, (b) a task id which was assigned to each process (called *task*) of an application and which was locally unique for this application and processor node and (c) a globally unique node id which identified any node in the system. This concept allowed a very simple steering and controlling of applications. MEMSOS could distinguish different applications and was able to run several of them concurrently in “space sharing” as well as in “time sharing” style.

Additionally, a very important new concept, called “*gang scheduling*,” had been implemented. The purpose of this was to assure the concurrent execution of tasks of one application on all nodes at the same time. This was achieved by creating a runqueue for each application on each allocated node. A global component, called *applserv*, retrieved information about the nodes and the applications started and running there. It established a ranking of applications which were to be scheduled.

Exemplary Applications

In order to validate the concepts of EGPA, certain applications from different scientific areas were adapted to the programming models of the EGPA monitor and of MEMSOS. In some cases, it was necessary to develop new parallel algorithms.

The corresponding programs were written a lot of years ago. The listings of them are no more available. Therefore, it is not possible to give an impression of the amount for writing such a program, e.g., in lines of code. Only the matrix multiplication (see page 7) gives an idea: 50 lines (an estimation of the not shown part is included).

The authors of such parallel programs wrote in their papers mainly about the principles of the underlying parallel algorithms and the runtimes. In many cases, they only published the maximal speedups received by

examples which were well-dimensioned for the used system. This means the problem size is maximal and, e.g., in case of the relaxation, the region under investigation is a square and not a circle or something else (see Fig. 3d). In a few cases, the dependency of the speedup on the problem size is depicted (see Gauss-Seidel and Table 1).

Using the Pilot pyramid, the maximum speedup was 4. Here are some of the measured speedups. The example “text formatting” demonstrates that not all types of tasks will run well on an EGPA system.

Linear algebra [9, 17]:

Matrix inversion: Gauss-Jordan (3.8), Column Substitution (3.95)

Matrix multiplication (3.7)

Gauss-Seidel iteration (4).

Speedup/dimension of the dense matrix: 1/10,
3.4/50, 3.8/200, 3.9/300, 3.99/500

Differential equations [21]:

Relaxation (3.5)

Image processing and graphics [10]:

Topographical representation (3.6)

Illumination of topographical model (2.9),

Vectorizing a grey level matrix (2.9),

distance transformation

- Each processor is working on a fixed part of data (3.0),
- Dynamic assignment of varying parts of data (3.3).

Nonlinear programming [8]:

Search for a minima of a multi-dimensional objective function (3.2)

Graph theory

Network flow with neighborhood aid (3.5)

Text formatting [19] (2.6)

Erlangen General Purpose Array (EGPA). Table 1 Results with 16 processors in the working plane [4]

Poisson equation with Dirichlet conditions: (speedup/number of cells in finest grid)	(11/1024), (14.4/16641), (15.3/65025)
Steady state Stokes equation with staggered grid: (speedup/number of cells in finest grid)	(10.3/1024), (13.7/4096), (15.3/16384)

In order to have a larger system, the DIRMU (Distributed Reconfigurable Multiprocessor based on 8086 with coprocessor 8087) was used for building an EGPA with three layers. Among other things, that multiprocessor was used to investigate multigrid methods which were established in the industry at that time. Though in coarser grids the work per processor is very small, the results were very good (see Table 1). In this context, it is worth mentioning that on a system, which only realized a working plane consisting of 25 DIRMU-PMMs, adaptive multigrid approaches were implemented with high efficiency [5].

On the MEMSY Prototype, several algorithms were realized with great success. The measured efficiency was mostly greater than 80%. This is true for all the above-mentioned algorithms which were ported from pilot or DIRMU to MEMSY [7], but also for the following real world applications [23]:

- Calculation of the electronic structure of polymers
- Parallel molecular dynamic simulation
- Parallel processing of seismic data by wave analogy of the common depth point
- Mapping parallel program graphs into MEMSY by the use of a hierarchical parallel. Algorithm
- Parallel application for calculating one-/two- and ecp-electron integrals for polymers

Only for sparse matrix algorithms, efficiencies near 60% were measured.

Outlook

At the same time when the mentioned EGPA systems were realized in Japan, a very similar systems, named PAX [22], has been built. Also in Japan, the System HAP represented a lot of the principles of EGPA systems [24].

In the 1990s, large commercial multiprocessors came into the market. Distributed memory MIMDs were now available which consisted of much more processors than, e.g., the MEMSY prototype. Even global shared memory systems with cache coherence were larger with respect to the number of processors. Especially the latter multiprocessors were easier to program than EGPA systems with their restricted neighborhoods. Consequently, the research on EGPA machines was stopped even in Erlangen.

Related Entries

- Computational Sciences
- Distributed-Memory Multiprocessor
- Metrics
- SPMD Computational Model
- Topology Aware Task Mapping

Bibliographic Notes and Further Reading

Further papers dealing with the EGPA pilot are [1, 6, 12, 13, 19, 20, 27]. Publications concerning MEMSY can be found on the page: www4.informatik.uni-erlangen.de/Projects/MEMSY/papers.html

Bibliography

1. Bode A, Fritsch F, Händler W, Henning W, Hofmann F, Volkert J (1985) Multi-grid oriented computer architecture. ICPP, 89–95
2. Cannon LE (1969) A cellular computer to implement the Kalman filter algorithm. PhD thesis, Montana State University
3. Dal Cin M, Hohl W, Dalibor S, Eckert T, Grygier A, Hessenauer H, Hildebrand U, Höning J, Hofmann F, Linster C-U, Michel E, Pataricza A, Sieh V, Thiel T, Turowski S (1994) Architecture and realization of the modular expandable multiprocessor system MEMSY. In: Proceedings of parallel systems fair of the 8th international parallel processing symposium (IPPS '94), Cancun, pp 21–28
4. Finemann H, Brehm J, Michel E, Volkert J (1988) Solution of the neutron diffusion equation through multigrid methods implemented on a memory-coupled 25-processor system. Parallel comput 8(1-3):391–398
5. Finemann H, Volkert J (1988) Parallel multigrid algorithms implemented on memory-coupled multiprocessors. Nucl Sci Eng 100:226–236
6. Fritsch G, Jens Volkert (1988) Multiprocessor systems for large numerical applications. Parcella 266–273
7. Fritsch G, Henning W, Hessenauer H, Klar R, Linster CU, Oelrich CW, Schlenk P, Volkert J (1989) Distributed shared-memory architecture MEMSY for high performance parallel computations. Comput Archit News 17(6):22–35
8. Fritsch G, Müller H (1981) Parallelization of a minimization problem for multiprocessor system. In: CONPAR 81. Lecture Notes Computer Science, vol III. Springer, Berlin/Heidelberg/New York, pp 453–463
9. Geus L, Henning W, Vajtersic M, Volkert J (1988) Matrix inversion algorithms for the a pyramidal multiprocessor system. Comput Artif Intell 7(Nr 1):65–79
10. Goosmann M, Volkert J, Zischler H (1982) Matrix image processing and graphics on EGPA. EGPA-Bericht, IMMD, Universität Erlangen-Nürnberg

11. Händler W (1974) A unified associative and von-Neumann processor EGPP and EGPP array. In: Sagamore Computer Conference, pp 97–99
12. Händler W, Klar R. (1975) Fitting processors to the needs of a general purpose array (EGPA). MICRO 8: 87–100
13. Händler W, Herzog U, Fridolin Hofmann, Hans Jürgen Schneider: Multiprozessoren für breite Anwendungsbereiche: Erlangen General Purpose Array. ARCS 1984:195–208
14. Händler W, Bode A, Fritsch G, Henning W, Volkert J (1985) A tightly coupled and hierarchical multiprocessor architecture. Comput Phy Commun 37:87–93
15. Händler W, Fritsch G, Volkert J (1985) Applications implemented on the erlangen general purpose array. In: Parcella'84. Akademie-Verlag, Berlin, pp 12–31
16. Händler W, Herzog U, Hofmann F, Schneider HJ (1975) A general purpose array with a broad spectrum of applications. In: Informatik Fachberichte vol 4. Springer, Berlin/Heidelberg/New York, pp 311–335
17. Händler W, Maehle E, Wirl K (1985) Dirmu multiprocessor configurations. In: Proceedings of the international conference on parallel processing, pp 652–656
18. Henning W, Volkert J (1985) Programming EGPA Systems. In: Proceedings of the 5th International conference on distributed computing systems, Denver, pp 552–559
19. Henning W, Volkert J (1986) Multi grid algorithms implemented on EGPA multiprocessor. ICPP, pp 799–805
20. Hercksen U, Klar R, Kleinöder W (1980) Hardware-measurements of storage access conflicts in the processor array EGPA. In: ISCA '80, Proceedings of the 7th annual symposium on computer architecture, ACM Press, New York, La Baule, pp 317–324
21. Hofmann F, Dal Cin M, Grygier A, Hessenauer H, Hildebrand U, Linster CU, Thiel T, Turowski S (1993) MEMSY – a modular expandable multiprocessor system. In: Proceedings of parallel computer architectures: theory, hardware, software, applications. Lecturer notes computer science, vol 732. Springer, London, pp 15–30
22. Hoshino T, Kamimura T, Kageyama T, Takenouchi K, Abe H (1983) Highly parallel processor array “PAX” for wide scientific applications. In: Proceedings of the international conference on parallel processing, IEEE Computer Society, Columbus, pp 95–105
23. Linster CU, Stukenbrock W, Thiel T, Turowski S (1994) Das Multiprozessorsystem MEMSY. In: Wedekind (Hrsg.): verteilte systeme. Wissenschaftsverlag, Mannheim, Leipzig Wien Zürich, pp 206–228
24. Momoi S, Shamada S, Koboyashi M, Tshikawa T (1986) Hierarchical array processor system (HAP). In: Proceedings of the conference on algorithms and hardware for parallel processing on COMPAR 86, pp 311–318
25. Rathke M (1985) Parallelisieren ordnungserhaltender Programmssysteme für hierarchische Multiprozessorsysteme. PhD thesis. Arbeitsberichte des IMMD, Universität Erlangen 18(2)
26. Rusnock K, Raynoha P (1990) Adapting the Unix operating system to run on a tightly coupled multiprocessor system. VMEbus Syst 6(5):8–28
27. Vajtersic M (1982) Parallel Poisson and biharmonic solvers implemented on the EGPA multiprocessor. In: ICPP 1982

ES

► [Earth Simulator](#)

Ethernet

MIGUEL SANCHEZ

Universidad Politécnica de Valencia, Valencia, Spain

Synonyms

[IEEE 802.3](#); [Interconnection network](#); [Network architecture](#); [Thick ethernet](#); [Thin ethernet](#)

Definition

Ethernet is the name of the first commercially successful Local Area Network technology invented by Robert Metcalfe and David Boggs while working at Xerox's Palo Alto Research Center (PARC) in 1973. While the prototype developed worked at 2.96 Mbps, the commercial successor was called Ethernet and worked at 10 Mbps. Several faster versions have been developed and marketed since then.

Ethernet networks are based on the idea of a shared serial bus and the use of Carrier Sense Multiple Access with Collision Detection (CSMA/CD) technique for network peers to communicate. Ethernet media access is fully distributed. Ethernet evolution brings the use of switching instead of CSMA/CD and full-duplex links. At the same time, coaxial cable is first replaced by twisted pair and then by fiber optics.

Discussion

Introduction

The world of Computing has always been a scenario of fast changes where state of the practice is challenged on a daily basis. Most of the advances in our field have come from this attitude of challenging the existing wisdom.

Thanks to that, we moved from the mainframe to the mini, and from it to the Personal Computer and now to embedded systems.

When we lived in a world of a few computers, networking seemed not to be in big demand, but as the number of computers in enterprise grew, a new need started to develop to make efficient use of companies' computing resources. It was the dawn of the Local Area Network (LAN).

In the early days of computing, most of the interconnection focused on adding dumb terminals to the existing mainframe computers. For that purpose, different variations of serial connections sufficed. However, as first minicomputers and then personal computers started to spread on the enterprise there was a need of another type of interconnection. In this new environment, the computing power started to move from a central location to be distributed to each desk. Together with that, the need of keeping systems interconnected became apparent, especially at those locations that have a significant number of mini or personal computers.

Xerox Corporation played an important role in the development of both the personal computer and LANs. More specifically, Xerox founded Palo Alto Research Center (PARC) in 1970 to conduct independent research. Many current technologies can be traced back to PARC, from the personal computer to the Graphical User Interface to Ethernet LANs. Xerox Alto was the first personal computer. It was developed in 1973 ([Fig. 1](#)).

The advent of Ethernet as a successful wired networking technology is based on a wireless precursor

developed at the University of Hawaii and first deployed in 1970 by Normal Abramson and others. They used amateur radios to create a computer network so teletypes at different locations could access a time-sharing system. It was called the ALOHA network. All terminals use a shared frequency to transmit to the host. Simultaneous transmissions destroy each other but if a transmission was successful then it was acknowledged from the host.

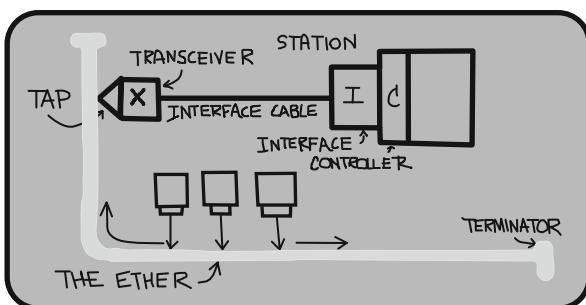
A network technology was needed to link together Alto computers with printers and other devices. No existing technology was found suitable so a team at PARC was set to develop a new technology. That was the embryo of Ethernet.

Collision Sense Multiple Access with Collision Detection

The basic principle of the ALOHA network that allowed a node to transmit at any time was simple but not very efficient. Harvard University student Robert Metcalfe sought to improve the basic idea, and so he did his PhD on a better system and joined PARC. The main improvement was that network nodes should check for an ongoing transmission before starting their own. The way to do this was to check whether a carrier signal was present or not. This technique is also called Listen Before Talk (LBT). This gave way to a technique called Carrier Sense Multiple Access (CSMA).

CSMA could be improved further by adding another mechanism called Collision Detection (CD). It consisted of stopping an ongoing transmission as soon as a collision was detected. A collision happens when two or more network nodes transmit at the same time. The purpose of this second improvement was to reduce the duration of collisions. This newer version of the access method was called Carrier Sense Multiple Access with Collision Detection or CSMA/CD and it was the base for the Ethernet network [1].

A third improvement over ALOHA was what to do after an unsuccessful transmission (collision). While ALOHA called for a random (but short) timer, Metcalfe's proposal was to set the retransmit timer as a random slot number chosen from a special set. Ethernet's Binary Exponential Backoff algorithm calls for random numbers to be selected from a range that starts as {0, 1} and doubles after each collision. The maximum range is {0 ... 1023}. This algorithm ensures that the



Ethernet. Fig. 1 Original Ethernet drawing by Robert Metcalfe

probability of a new collision after a collision is halved. While CSMA/CD works on continuous time, Binary Exponential Backoff works on slotted time, each slot being the time required to transmit 512 bits.

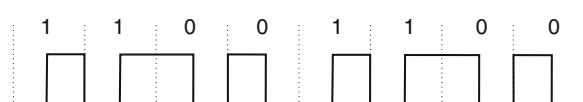
Robert Metcalfe was helped by Stanford student Dave Boggs and they created a working system by 1973 networking together several Alto computers.

Ethernet Topologies

Original Ethernet design used a bus topology with a coaxial cable as the shared media for all the transmissions. Data transmission used a baseband signal using Manchester encoding. A bus was found as a suitable way to broadcast information to all the network nodes and no switching was needed. The network bus was terminated with a resistive load on each end to reduce signal reflection. Maximum segment length was up to 500 m. Repeaters could be used to connect two or more segments together. No more than four consecutive repeaters were allowed. Maximum network diameter was 2,500 m (Fig. 2).

Coaxial cable, specially the thick cable, was difficult to use in an office setup due to its size and it was expensive. This topology had the main drawback of a bus being a single point of failure.

In order to reduce the cost of the network some revisions of Ethernet called for a thinner cable. This so-called Thin Ethernet was still based on a bus topology but using a thinner cable would make it not only cheaper but also easier to install cable runs. Unfortunately, the new setup with thinner cable was not without its own troubles as the attachment of each station was made using so-called T-connectors that required the coaxial cable to be cut at each joint and required two connectors to be added. Both the number of connectors and the physical characteristic of the cable (RG-58) limited the length of Thin Ethernet segments to 185 m.



Ethernet. Fig. 2 Manchester encoding as used on Ethernet

The availability of broadband coaxial cable on some premises for cable TV distribution enabled another variation for Ethernet, but now using broadband signals and these call for a change as transmission is here unidirectional. This broadband version of Ethernet was never very successful but it allowed a maximum distance of 3,600 m.

In a quest for cheaper alternatives, coaxial cable was replaced by twisted pair and the bus topology gave way to a star topology. While the first attempt at this only worked at a fraction of the coaxial data rate, the same speed was achieved with a later version. The main advantage of using twisted pair was its wide availability in office setups as it was used for voice communications. Twisted pair was also cheaper and easier to install and to handle. But the use of star topology called for a central multi-port repeater or hub. Hub device was, in essence, a bus-in-a-box. However, the star topology meant that if a cable was severed only one network node was affected. In a bus topology, if the coaxial cable is severed the whole network stops working. The star topology proved to be much more robust than the bus counterpart but each cable run was limited to 100 m giving a maximum distance of 200 m for each segment.

Support for fiber optic media came from the need for both longer distance links and electrically isolated links. The first use of fiber optic media in Ethernet networks was for the interconnection of repeaters. It later developed as an alternative media to coaxial or twisted pair. While more expensive than copper, fiber optic media allowed longer distances of up to 1,000 m. Ethernet fiber optic links were point to point and that meant a star topology. A passive-star over fiber was proposed but it was never implemented.

The First Ethernet, and the First Commercial Ethernet Success

The first version of Ethernet was deployed in PARC in 1973 and it ran at 2.94 Mbps and used 8-bit values for source and destination addresses. The first 16 bits of the data field were reserved for a type identifier, so different protocols could be used over the same network each one using a different value for the type identifier. Data frames were protected against errors by means of a Cyclic Redundancy Check of 16 bits placed at the end of the frame.

The original name for this network was the Aloha Alto Network, as it was developed for interconnecting Alto computers together and with laser printers. One of the original prototype boards can be found now at the Smithsonian.

However, it was soon recognized that such a name suggested it would only work with Alto computers. The *Ethernet* name was a vendor-neutral name that was based on an old concept of wireless transmissions. During nineteenth century, it was believed that wireless signals propagate using an invisible media called *ether*, although physicists Michelson and Morley rejected that misconception in 1887.

DEC, Intel and Xerox

In 1979, Xerox, DEC and Intel joined to develop a common specification for Ethernet products: The first commercially successful Ethernet network was born. The specification was published as the “blue book” [2].

The new Ethernet specification, also known as DIX (acronym based on the three companies names), called for 10 Mbps data rate, 48-bit addresses, 16-bit type field, and 32-bit Cyclic Redundancy Check (CRC) field (Fig. 3). The data field was of variable size of up to 1,500 bytes. It was a new and improved version of the first Ethernet and now it was set to become a viable commercial product. At this same time, the market for personal computers blossomed with many different models. While this first commercially available version of Ethernet sold well, there were some concerns about the potential lock-in by the three companies. The Institute of Electrical and Electronic Engineers (IEEE) came into play. Ethernet would be standardized within the networking group 802, who met for the first time on February 1980. Ethernet became the IEEE 802.3 standard in 1983 [3]. The availability of the 802.3 standard eased the doubts about the maturity level of Ethernet networks. At the same time, many different companies started selling compatible products many of them intended for the — by then — new IBM PC computer.

IEEE 802.3 also developed several choices for physical media and used a naming convention that prepended the data rate to a word describing the modulation scheme (baseband or broadband) and next the physical media used. 10BASE5 described a 10 Mbps version of Ethernet using Manchester encoding over RG-8X coaxial cable. 10BASE-T described a similar network but over twisted-pair cable.

Eventually twisted pair became the cabling of choice, mostly using two of the four pairs of Category 3 and Category 5 cable. While twisted-pair setups required the use of expensive active hubs, the savings for using cheaper cable and the improved reliability of the network compensated for the additional costs.

CSMA/CD Ethernet Modeling and Problems

One of the early criticisms of Ethernet was the fact that media access is probabilistic (nondeterministic). That means transmission latency cannot be known beforehand. The deterministic behavior of other media access algorithms, like IEEE 802.5 (Token Ring) was considered superior, but average transmission latency is much lower on Ethernet than on a token-passing network (except under very heavy network loads).

Ethernet (and CSMA/CD by extension) was found to exhibit what was called “capture effect,” that happens when two stations contending for the medium have a collision and both have more frames to transmit. The winner will transmit successfully and it will keep an edge on future transmissions against the station that lost the transmission slot after the collision. This effect causes a transient channel-access unfairness that in turn can interfere with higher layer protocols, like TCP, that can amplify the effect.

A second cause of concern was the behavior of heavy loaded Ethernet networks [4, 5]. As traffic grows beyond a certain limit, the chances of a transmission attempt to end up in a collision grow too. Having an increased chance of a collision when traffic load is high introduces a positive feedback that can bring the network to a congestion state. But as Ethernet data rate goes up, so



Ethernet. Fig. 3 Ethernet frame format

does the overload limit. Again, the use of full-duplex Ethernet avoids this problem.

Next Stop: 100 Mbps Ethernet

Once the 10 Mbps Ethernet IEEE standard was approved, the 802 group was tasked to develop a 100 Mbps version. This faster version was known as *Fast Ethernet*. Different standards allowed this faster speed over different types of physical media using both copper and fiber optic media. Twisted-pair cable is not very well suited for high speed communication so different modulation schemes were used over different types of twisted-pair cables to give way to several physical layer specifications: 100BASE-TX use 4B5B MLT-3 coded signals over Category 5 twisted-pair cable while 100BASE-T4 uses 8B6T OAM-3 coded signals over four twisted pairs of Category 3. Similarly, 100BASE-FX is Fast Ethernet media access for multimode fiber. Depending on whether half-duplex or full-duplex links are used, 400 m or 2 km is the maximum link distance respectively for fiber media.

Switched Ethernet

Ethernet evolved not only becoming faster but also adding new components to the original architecture. A shared bus proved to be not enough for large networks. The maximum distance constraints together with traffic considerations opened the way to bridges. One way to increase the capacity of a network segment was to split it into two smaller segments, each one now having fewer computers competing for the available capacity. But these two segments need now to be interconnected so distributed applications can still work as before.

The original bus topology gave way to a star topology using twisted-pair cable. Hubs provided the interconnection between multiple stations but there was a problem: All the stations had to work at the same data rate on the same collision domain. Bridges could be used to split one collision domain into several interconnected network segments where only intersegment traffic was shared.

Therefore, both hubs and repeaters enabled the interconnection of two or more network segments. The term “hub” is used only for the interconnection of point-to-point links while repeaters could be used for interconnecting two coaxial-cable Ethernet segments.

From the logical point of view, both exhibit the same behavior working at the physical layer. A repeater retransmits a digital signal so it can reach a longer distance. A repeater does not create a separation of the collision domain.

A bridge is an interconnection device that works at the Link Layer. Bridges connect two or more cable runs called network segments. Bridges understand MAC addresses and create several independent collision domains. Only traffic intended for other network segment is actually forwarded by a bridge. The aggregate bandwidth of a network interconnected by a bridge is higher than if the interconnection uses a repeater, but so is latency.

The term “bridge” was usually associated with the interconnection of large and usually coax-based segments while the term “switch” referred mostly to interconnection devices using point-to-point links, where each port would have a workstation or server. From the logical point of view, bridges and switches are similar.

Little by little, bridges were replaced by new devices called switches that offered similar functionality but many ports per unit. Switches started to replace network hubs. There was a performance increase but the cost was high too.

The different topologies of Ethernet had to be loop-free. But given that bridges or switches were used to interconnect network segments there was a risk of creating network loops. On the other hand, having network loops is a way to introduce redundancy to the topology of a network, a feature that could be very useful if it did not interfere with Ethernet media access. The problem was that a new component was needed for Ethernet not to have problems in the presence of loops in the network topology or frames would be forwarded forever within the loops.

The brilliant solution to the problem was the Spanning-Tree protocol [6], invented by Radia Perlman while working at Digital Equipment Corporation and ratified as the IEEE 802.1d standard. This protocol running on switches and bridges creates a spanning tree out of the existing network topology. Frames are only forwarded on network segments that belong to the spanning tree and, being a tree a loop-free topology, the problem of forwarding loops is solved. Whenever one of the links in use stops working, a new spanning

tree is calculated making use of the available redundant links. Bridges and switches exchange control frames periodically to maintain the spanning tree.

A switch, like a bridge, works on Link Layer and creates an independent collision domain on each port. Switches are store-and-forward devices that can easily handle different data rates on different ports. Switches can handle simultaneous frame forwarding on different ports, an event that would have caused a collision should a hub were used. Different proprietary flow-control mechanisms were developed for half-duplex switches based on back pressure *congestion management*: A switch may transmit a carrier signal on those ports it wants to alleviate congestion.

Ethernet switches may use different approaches to switching. The simpler way was to use the store-and-forward approach where a frame is stored in a buffer to be retransmitted once it is received entirely by the switch. Cut-through switches only store the very first bytes of the frame and once the destination address is known the forwarding of the frame can start. This second approach leads to a significant reduction of the switching latency experienced by frames.

An Ethernet network that uses switches instead of hubs enables a new improvement: full-duplex links. Provided that the physical media has two different circuits for simultaneous transmission and reception (e.g., two different copper pairs on 100BASE-TX), it is possible for a station to transmit and receive at the same time. While full-duplex Ethernet does work at the same data rate as half-duplex, higher throughput can be achieved with full-duplex. Besides, collisions do not occur in a full-duplex point-to-point link. A collision-free link is going to be more effective as the time devoted to collisions can now be put to use to transmit more data. However, network adapters still need to keep the CSMA/CD functionality for backward compatibility.

Together with full-duplex operation came the 802.3x flow-control mechanism. Sometimes, a receiver on a full-duplex link may not be able to handle frames fast enough. Ethernet flow-control is based on the receiver transmitting a special *pause* frame to a multicast MAC address. Such a frame reports the amount of time (using 512 bit-time units) the sender on that full-duplex link must wait before the next transmission. If a receiver is

able to accept data sooner than expected, a new pause frame with zero wait-time can be transmitted.

Ethernet Gets Faster

In May 1996, the Gigabit Ethernet Alliance was founded by eleven companies interested in the development of Gigabit Ethernet (i.e., 1,000 Mbps Ethernet). The original aim of this new Ethernet version, faster than Fast Ethernet, was to make server and backbone connections even faster. Gigabit Ethernet was covered in IEEE 802.3z and 802.ab standards and it was approved in 1999. Gigabit Ethernet can use either multi-mode fiber, single-mode fiber, or twisted-pair copper cable.

Gigabit Ethernet over copper (known as 1000BASE-T) poses a special challenge for twisted-pair connections that could use Category 5 cable only if it meets some restrictions. This is the reason for Category 5E or Category 6 copper cable. Different techniques were used for making possible to get gigabit speed over copper (e.g., QAM-modulation to more efficiently utilize the channel bandwidth).

Many changes were needed beyond speed. Half-duplex operation was still allowed and so a technique called *carrier extension* was added to allow Ethernet to keep CSMA/CD and the ordinary 200 m maximum network distance. An extension field is appended as needed to ensure a minimum frame length of 512 bytes. Alternatively, another optional improvement was *frame bursting*: It allows the transmission of consecutive data frames after one MAC arbitration on half-duplex channels.

In June 2002, 10 Gigabit Ethernet was ratified. With this new version of Ethernet the performance level is reaching or surpassing other Metropolitan Area Network (MAN) and Wide Area Network (WAN) technologies such as OC-12 and OC-48, SONET or SDH technologies. This is the first version of Ethernet only supporting fiber optic media (10GBASE-CX4 barely supports 15 m of copper cable, just for attaching a switch to a router). A 10 Gbps link range is between 65 m and 40 km. Both multi-mode and single-mode fibers are supported.

Ethernet and Beyond

New versions of Ethernet are coming as 40 Gbps Ethernet and 100 Gbps Ethernet. The IEEE 802.3ba standard

has been ratified in June 2010. A bit unusual was the fact that an interim 40Gbps version was developed largely as a step towards 100Gbps Ethernet. The 802.3ba calls for full-duplex fiber optic links.

Data Center Ethernet (DCE) extends Ethernet architecture to improve networking and management in data centers. Several features like Priority-Based Flow-Control, Enhanced Transmission Selection (standardized as 802.1Qaz), Multipathing or Congestion Management (802.1Qau) focus on improving network features and manageability. The vision of DEC is a unified fabric where LAN, storage area network (SAN), and inter-process communication traffic (IPC) converge.

Technologies like Fiber Channel over Ethernet (FCoE) enable data centers to consolidate interconnection and internetworking technologies into a single network, which may simplify connectivity and reduce data center power consumption.

Whether this is the last stop of Ethernet's network technology or not, it is unclear. However, if past is an indication of what may be ahead, an even faster version of Ethernet should be expected in a few years time.

Related Entries

- ▶ [Buses and Crossbars](#)
- ▶ [Clusters](#)
- ▶ [Distributed-Memory Multiprocessor](#)
- ▶ [Network Interfaces](#)
- ▶ [Network of Workstations](#)

Bibliographic Notes and Further Reading

Access to many standards has been fee-based. IEEE 802 standards started a move a few years ago by opening their documents to the public, for free. The documents of published standards are available for free download 12 months after they are published. That means all details about any approved Ethernet standard version is easily available visiting <http://standards.ieee.org/getieee802/802.3.html>

Unfortunately, standards are usually written so all details are covered but many times they are not reader-friendly. For getting a good understanding of Ethernet technology, the Charles Spurgeon's [7] guide is a very appropriate text.

Bibliography

1. Metcalfe RM, Boggs DR (1976) Ethernet: distributed packet switching for local computer networks. *Commun ACM* 19(7):395–405
2. The Ethernet (1980) A local area network: data link layer and physical layer specifications (version 1.0). Digital Equipment Corporation, Intel, Xerox
3. ANSI/IEEE Standard 802.3-1985 Carrier sense multiple access with collision detection. IEEE, October 1985
4. Boggs DR, Mogul JC, Kent CA (1988) Measured capacity of an Ethernet: myths and reality. In: Proceedings of ACM sigcomm '88 symposium of communications architecture and protocols, Stanford, CA, (Also as Computer Commun Rev 18(4), August 1988)
5. Takagi H, Kleinrock L (1985) Throughput analysis for persistent CSMA systems. *IEEE Trans Commun COM-33(7):627–638*
6. Perlman R (1999) Interconnections: bridges, routers, switches, and internetworking protocols (2^a ed. edición). Addison-Wesley, Reading, MA
7. Spurgeon CE (2000) Ethernet—the definitive guide. O'Reilly & Associates, Inc, Sebastopol

Event Stream Processing

- ▶ [Stream Programming Languages](#)

Eventual Values

- ▶ [Futures](#)

Exact Dependence

- ▶ [Dependence Abstractions](#)

Exaflop Computing

- ▶ [Exascale Computing](#)

Exaop Computing

- ▶ [Exascale Computing](#)

Exascale Computing

RAVI NAIR

IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA

Synonyms

[Exaflop computing](#); [Exaop computing](#)

Definition

Exascale computing refers to computing with systems that deliver performance in the range of 10^{18} (exa) operations per second.

Discussion

Introduction

In computing literature, it is customary to measure progress in factors of 1000. The term exa, standing for 10^{18} or 1000^6 , is derived from the Greek word ἕξ, meaning six. Thus exascale computing refers to systems that execute between 1000^6 and 1000^7 operations per second. This performance level represents a 1000-fold increase in capability over petascale computing, which itself represents a 1000-fold increase over terascale computing.

There is often confusion over how to denote the performance of a system. The most common measure is the number of double-precision floating-point operations that are completed every second (flops) when the system is running the Linpack dense linear algebra benchmark. By this measure, an exascale system is one which can execute Linpack at the rate of 1 exaflops or higher. For a given system, this is usually less than the peak double-precision floating-point execution rate, typically by 10–30%, because of practical limitations in exploiting the full floating-point capability in a system. In many systems that implement only single-precision floating point in hardware, the peak single-precision floating-point capability may be more than double the flops as measured for Linpack. Petascale computing was ushered in with the introduction of the IBM Roadrunner system in May 2008, when it ran Linpack at 1.1 petaflops. If trends continue as in the past, it is reasonable to expect an exascale system to be operational around 2018.

A list of the Top 500 supercomputers in the world has been maintained since June 1993 and is a useful indicator of trends in the capabilities of the largest scientific machines. The list shows that there has indeed been a trend of doubling of Linpack performance every year, but that the manufacturer of the highest-performing system and the architecture of the system have been changing over time. While shared-memory computers and even uniprocessors made the list in earlier days, today the list is dominated by cluster supercomputers and massively parallel processors, the latter being distinguished from the former by their highly customized interconnection between nodes.

With growing interest in the use of large-scale machines for applications that are considered commercial (or, more appropriately, nonscientific), the term exascale is also being used to refer to systems which can carry out 1000^6 operations per second (1 exaops). There continues to be ambiguity in what constitutes an operation in this terminology. An operation is variously defined as an instruction, an arithmetic or logical operation, an operation intrinsic to the execution of an algorithm, and so on. This has led to one abstract definition of an exascale system to be one that has a performance capability roughly 1000 times more than a system solving the same problem in 2008.

In any case, the needs of the two worlds, scientific and commercial, appear to be converging especially from the points of view of compute capability, system footprint, energy requirements, and cost. It remains to be seen whether, in comparison to systems of 2008, the first exascale system will execute only Linpack 1000 times faster, or whether it will solve a variety of problems, both scientific and commercial, faster by a factor of 1000.

The Importance of Exascale

Large supercomputers are already being used to solve important problems in a variety of areas. The International Exascale Software Project (IESP) has categorized the present areas of application of supercomputers as follows:

- Material Science
- Energy Science
- Chemistry
- Earth Systems

- Astrophysics and Astronomy
- Biology/Life Systems
- Health Sciences
- Nuclear and High Energy Physics
- Fluid Dynamics

Each of these areas has specific problems and a set of algorithms customized to solve those problems. However, there is an overlap in the use of algorithms across these domains, because of similarity in the basic nature of some of the problems across domains. Many of these problems could benefit from even larger supercomputers, but the expectation is that exascale systems will also bring a qualitative change to problem solving, allowing them to be used in new types of problems and in new areas.

There are two ways in which larger supercomputers can help in the solution of existing problems. The first is called strong scaling, where the problem remains the same, but the solution takes less time. Thus, a weather calculation that forecasts the weather a week ahead is more useful if it takes 5 h to solve rather than 5 days. The other form of scaling is called weak scaling, where the size of the problem that can be solved is vastly increased with the help of exascale computers with its significantly larger resources. Thus the quality of a weather forecast can be improved if it is based on global data rather than local data, or if the forecast is made based on data from a larger number of more closely located sensors, or if the model is increased in complexity, for example, through the use of more types of meteorological information.

Beyond just scaling of existing applications, large-scale computing enables new types of computation. It becomes possible to model in a single application a physical phenomenon at different scales in time, space, or complexity simultaneously, for example, at the atomic, molecular, granular, component, or structural scale, or at the molecular, cellular, tissue, organ, or organism scale. This capability is already becoming evident at the petascale level and will be further exploited at the exascale level. Another new and interesting class of applications that will be further exploited in exascale computing is the quantification of uncertainty in mathematical models. These applications involve the use of massive parallel computing to understand the behavior

of physical systems in the presence of both systematic and random variations.

The list of areas that could benefit from the use of supercomputers has been increasing steadily. Applications for supercomputing are also emerging in areas that were not traditionally considered scientific areas. Examples of these are visualization, finance, and optimization. Supercomputers have been used in the past by large movie production companies to render animated movies frame-by-frame over long periods of time. However, for typical scientific applications, the results produced by supercomputers were often packaged and shipped to a significantly smaller computer which the scientist used to analyze and visualize the results. The point where it will be too expensive to ship the vast amount of data produced by supercomputers is fast approaching. Increasingly, supercomputers will be used to process the results of their computation and produce visualization data to be shipped over to the scientist.

The area of business analytics, traditionally considered a commercial application, is fast embracing the use of massive computers. The potential use of massive computation in such areas suggests that exascale computers are likely to represent the point at which supercomputers shift from being the largely simulation-oriented scientific machines that they have traditionally been to more versatile machines solving fundamentally different types of problems for a larger section of humanity.

The Challenges of Exascale

The principal engine for the advances made by supercomputers over the past decades has been Dennard scaling through which the electronics industry has enjoyed the perfect combination of increased density, better performance, and reduced energy per operation with successive improvements in silicon technology. When all dimensions of an electronic circuit on a chip, for example, the transistor gate width, the diffusion depth, and the oxide thickness, are reduced by a factor α , and when the voltage at the circuit is also reduced by a factor α , it is possible to increase the density of packing of devices on the chip by a factor of α^2 and increase the frequency of operation of the device by a factor of α , with a reduction in energy per operation by a factor of α^3 . With process technologists providing lithography improvements that

allowed a doubling of transistor density roughly every 2 years, with microarchitects discovering new ways to expose parallelism at the instruction level, and with system architects engineering sophisticated techniques to combine multiple processors into ever large parallel systems, it was not difficult to maintain a steady doubling of supercomputer system performance every year at almost constant cost.

Going forward from the current 32 nm technology node, it will become increasingly difficult to keep up with Dennard scaling. With oxide thickness down to a few atoms, voltages cannot be scaled proportionally without serious compromise to the reliability and predictability of the operation of devices. It is no longer reasonable to expect the same frequency improvement for constant power density that was enjoyed in the past. At the same time, the nature of typical application programs makes the detection and exploitation of additional parallelism within a single thread of execution difficult. Thus, scaling the total number of threads in a system remains the main option to increase the performance of future large-scale systems. With current petascale systems already sporting hundreds of thousands of cores, the expected number of cores in an exascale system is likely to be in hundreds of millions. Due to various restrictions that will be enumerated below, most of this increase will have to come from using whatever lithography advances are possible to increase the density of cores on a chip.

The task of achieving the goal of building an exascale system by 2018 is daunting. There are challenges in implementing the hardware, in designing the software environment, in programming the system, and in keeping the machine running reliably for long periods at a stretch. Listed below are brief descriptions of some of these challenges.

Power

A quick glance at the Top 500 list shows that the power ratings for the top two petascale systems are, respectively, 4 MW/Petaflops for the Cray system and 2.25 MW/Petaflops for the IBM Roadrunner system. At this rate, an exascale system could consume as much as 2000–4000 MW of power, a quantity that is larger than the amount of power consumed by a medium-sized

US city. The cost of acquiring the largest supercomputer has remained largely constant over the years at around \$100–200 million. This is the cost of about 100–200 MW-years of energy. Thus it is unlikely that exascale installations will be willing to provide more than about 30–50 MW of power, a third of which can be expected to be lost in cooling and power distribution. There is thus a gap of 70x–100x between the power efficiency of today's machines and the desired efficiency of exascale machines. Traditional CMOS scaling would have bridged that gap in about four technology generations, but with the slowing down of CMOS scaling and with longer intervals between introductions of technology generations, it is imperative to look for other avenues to achieve the desired power efficiency.

This gap must be bridged for all components of the system – the processor, the memory system, and the interconnection network. At the processor level, it is likely that exascale systems will include domain-specific accelerators because it is easier to build hardware that is power efficient in a specific domain rather than across a general range of applications. Vector units, which perform the same computation on multiple data simultaneously, could play an even bigger role in exascale systems precisely for this reason. Unlike what is seen in desktop and server processors today, processor designers will be forced to take a minimalist view in every part of the processor, paring overhead down to a minimum without significant impact on performance. Power-efficient optics technology will need to be deployed in interconnection networks if the total system bandwidth has to be scaled proportionally from what exists today. Memory system designs, both at the main memory level and at the cache level, will have to be reinvented with the above stringent power-efficiency requirements in mind.

Memory

One of the trends seen in the top supercomputers is that the fraction of the total cost of the system and the fraction of the total system power going into memory has been steadily rising, even though the ratio of memory capacity to the computational performance (bytes/flop) has been decreasing. In the Top 500 machines, this ratio has decreased from 1 byte/flop to as low as 0.1 byte/flop in some cases. The result of this is that there is a greater pressure on applications



to adapt to the reduced amount of memory by carefully managing the locality of applications, and a pressure on processor designers to provide ever-growing cache capacities and deeper cache hierarchies, both of which add to the challenge of streamlining the processor design to achieve a low power-performance ratio for exascale systems.

Another problem is the bandwidth at the memory interface. DRAM designs have been catering mainly to the commodity market and hence, while their capacities have been keeping up with Moore's Law, their bandwidths have not kept up with increases in processor performance. Cost considerations have prevented supercomputer vendors from designing custom DRAM solutions. However, the rampant data explosion in all types of information processing is forcing even commodity memory vendors to rethink the designs of memory interfaces and modules. It is conceivable that such new designs along with new storage-class memory technologies and new packaging technologies will do a better job in meeting the needs of exascale systems.

Communication

An important aspect of all large-scale scientific applications is the amount of time spent communicating between computation nodes. Applications impose a certain communication pattern which often reflects the locality of interactions of objects in the physical world. Thus the topologies of interconnection networks in large systems tend to be optimized for such communication patterns. Increasingly, however, the nature of algorithms and the types of problems being solved are imposing greater randomness in the pattern of communications and hence greater pressure on the available bandwidth.

One measure used to characterize the behavior of systems for nonlocal communication patterns is the bisection bandwidth. This is the least bandwidth available between any two partitions of the system each having half the total number of computing nodes. Petascale systems for the High Productivity Computing Systems (HPCS) program are required to have a bisection bandwidth of at least 0.5 PB/s. But few organizations will either need or will be able to afford such a high bandwidth per flops ratio.

At the exascale level, costs will force the bandwidth per flops ratio to be even lower. The ExaScale Computing Study sponsored by DARPA estimated the bisection bandwidth requirements at the exascale level to be in a range between 10 PB/s and 1 EB/s. Such a high bandwidth will force a predominant use of optics technology across the system, not just between racks, but also at the board level and perhaps even at the chip level. Exascale computing would benefit from lower costs through commoditization of optics components and from further advances in optics technology, like silicon photonics.



Packaging

As mentioned earlier, the power goal of 30–50 MW for an exascale system will impose challenging requirements in all aspects of the system. One such aspect is packaging. The number of racks in an exascale system cannot be allowed to increase dramatically from today because of limitations in the size and cost of installations. Thus each rack will have to accommodate more boards and each board will have to accommodate more chips than at present. Increased packaging density will increase power density at various levels of the system and will make testing, debugging, and servicing of the system more challenging.

Two technologies that are maturing and that could help in this respect are 3D stacking of silicon dies and the use of silicon carriers. 3D stacking not only helps in providing a compact footprint but also reduces the latency of communication between circuits on different layers. With judicious use of through-silicon vias (TSVs), 3D technology could also provide greater bandwidth between sections of the processor design without having to resort to large, expensive dies.

In comparison to the 1 mm pitch of printed-circuit board technology, silicon carrier technology allows mounting of dies closer than 100 um, wiring pitch of 1–10 um, and an I/O pitch of 10–100 um. The higher signaling rates, the bandwidth, and the higher I/O density promised by silicon carrier technology at lower power and in a compact footprint will go a long way toward meeting exascale goals.

Supercomputers have always been at the forefront in packaging; exascale promises to be no different.

Reliability

The number of components in a system has a direct bearing on the reliability of the system. Today's petascale systems, even with their redundancy and error-correcting techniques at the cache and memory levels, have a mean time between failures (MTBF) of less than a month. A direct extrapolation for a 1000-fold increase in the number of components brings this number down to a few minutes. The conventional method of handling failures is to take a checkpoint of the relevant state of an application at an interval well under the MTBF of the system. When a failure occurs, the system is interrupted, the state reverted back to a previously checkpointed state, and execution resumed. The maintenance of checkpoints requires the system to be periodically interrupted and typically involves copying of processor and memory state to some other part of memory or to storage. The overhead of this checkpointing process could impede forward progress of the application if it has to be performed too often. New techniques for detection of faults and for recovery from faults are needed for future exascale systems.

While coding and redundancy techniques could be used to improve the MTBF of the system, they come at a cost – in area, in power, and often in performance. With the exascale goal already a significant challenge on all these fronts, there is a growing belief that reliability techniques can no longer be application agnostic, and that applications will need to guide when and how checkpointing needs to be done in future large systems.

Usability

The eventual success of exascale computing will be measured not by whether it can execute the Linpack benchmark at an exaflops rate but by how successfully it is embraced by the community and how fundamentally it transforms both science and commerce. The preceding discussion about the challenges of implementing an exascale system suggests that the role of software will be important in the effective exploitation of such a system. On the one hand, life for application developers will be made easier if the system supports today's programming models without change, but on the other hand, software will have to work around the necessary restrictions in hardware due to reasons mentioned above.

Perhaps the most effective way in which this can be accomplished is through widespread adoption of a

new exascale programming model. Neither MPI, the standard message-passing paradigm, nor OpenMP, the standard shared-memory paradigm, alone can exploit a system almost certainly likely to include a cluster of shared-memory SIMD processors. There is growing interest in a hybrid model, for example, one that uses OpenMP at the local chip level and MPI at higher levels. New programming languages classified as Partitioned Global Address Space (PGAS) languages like X10, UPC, Co-Array Fortran, and Titanium provide the programmer the view of one large shared address space for the entire system, but still allow the programmer to specify partitions of the address space local to each thread.

Various parts of the software stack will also be expected to incorporate features to tolerate the expected high rate of failures as discussed in the previous section. While this could be handled directly through the programming language, ease-of-use considerations are likely to force reliability to be taken care of either at the compiler level or in the runtime system, as in the Map-Reduce programming model.

The role of compilers and runtime systems also becomes important if the system is heterogeneous like some of today's petascale systems or if domain-specific accelerators and field-programmable gate-arrays (FPGAs) become part of the system hardware. While a small number of proficient users, the so-called top-gun programmers, would enjoy programming with such a diverse array of components, the vast majority of users would prefer a compiler or the system to decide when and how to employ these components to deliver optimal performance for their programs.

A new organization, the International Exascale Software Project (IESP) has been formed with the recognition that unless a coordinated global effort is begun now, the programming community will not be ready to utilize exascale systems effectively when they are deployed. The IESP has taken upon itself the task of outlining a roadmap for the development of systems software (including operating systems, runtime systems, I/O systems, systems management software), of development environments (including programming models, frameworks, compilers, libraries, and debugging tools), of algorithm support (including data management, analysis, and visualization), and also of algorithms for common applications.

Conclusion

Exascale systems promise to usher in a new era of computing where massive supercomputers will be used not only for traditional simulations of large scientific problems but also to process the vast amount of data that is being produced especially from new types of ubiquitous devices. The success of such large systems, however, will be determined by how effectively they are utilized. Exascale systems can be expected to blaze new trails in power-efficient hardware design, in a system-wide approach to ensuring availability despite expected failures, and in new software approaches to maximize the utilization of the available computation power.

Related Entries

- ▶ [Chapel \(Cray Inc. HPCS Language\)](#)
- ▶ [Checkpointing](#)
- ▶ [Clusters](#)
- ▶ [Coarray Fortran](#)
- ▶ [Distributed-Memory Multiprocessor](#)
- ▶ [Fault Tolerance](#)
- ▶ [Hybrid Programming With SIMPLE](#)
- ▶ [LINPACK Benchmark](#)
- ▶ [Massive-Scale Analytics](#)
- ▶ [Metrics](#)
- ▶ [Networks, Direct](#)
- ▶ [Networks, Fault-Tolerant](#)
- ▶ [Networks, Multistage](#)
- ▶ [PGAS \(Partitioned Global Address Space\) Languages](#)
- ▶ [Power Wall](#)
- ▶ [Reconfigurable Computers](#)
- ▶ [TOP500](#)
- ▶ [UPC](#)

Bibliographic Notes and Further Reading

The most comprehensive description of the challenges of exascale systems can be found in the report of a study [7] under the sponsorship of the DARPA Information Processing Techniques Office. The software challenges are detailed in a similar study described in [9]. The International Exascale Software Project [5] is also actively pursuing an agenda to ensure adequate software exploitation of future exascale systems. DARPA also sponsored a study on reliability aspects of exascale systems and the resulting whitepaper is available at [2].

The Top500 Web site [10] gives a historical trend of the performance and capabilities of the highest-performing scientific computers. The limits of CMOS scaling and its effect on power consumption in future technologies are described well in [3]. An excellent overview of the architecture, power consumption, and economics of large datacenters may be found in [1].

There are several new technologies that promise to supplant or complement nonvolatile storage. A good description of storage-class memories may be found in [4]. There is recent interest also in the use of such memories as extensions to traditional DRAM main memory [8].

A comprehensive description of challenges in packaging and potential solutions can be found in [6]. The promise of 3D technology is discussed in [11]. An introduction to recent advances in silicon photonics appears in [12].

Bibliography

1. Barroso LA, Hözle U (2009) The datacenter as a computer: an introduction to the design of warehouse-scale machines. In: Hill M (ed) *Synthesis lectures on computer architecture*. Morgan and Claypool Publishers, San Rafael, CA. Available: <http://www.morganclaypool.com/doi/abs/10.2200/S00193ED1V01Y200905CAC006>. Accessed 15 Feb 2010
2. Elnozahy EN (ed) (2010) System resilience at extreme scale. Available: http://www.darpa.mil/ipto/personnel/docs/ExaScale_Study_Resiliency.pdf. Accessed 15 Feb 2010
3. Frank D (2002) Power-constrained CMOS scaling limits. IBM J Res Dev 46(2/3):235–244
4. Freitas RF, Wilcke WW (2008) Storage-class memory: the next storage system technology. IBM J Res Dev 52(4/5):439–447
5. International Exascale Software Project (2009) Main page. Available: http://wwwexascale.org/iesp/Main_Page. Accessed 15 Feb 2010
6. Knickerbocker JU et al (2005) Development of next-generation system-on-package (SOP) technology based on silicon carriers with fine-pitch chip interconnection. IBM J Res Dev 49(4/5):725–753
7. Kogge PM (ed) (2008) ExaScale computing study: technology challenges in achieving exascale systems. Available: http://www.darpa.mil/ipto/personnel/docs/ExaScale_Study_Initial.pdf. Accessed 15 Feb 2010
8. Qureshi MK, Srinivasan V, Rivers JA (2009) Scalable high performance main memory system using phase-change memory technology. In: Proceedings of ISCA-36. Austin, TX, USA
9. Sarkar V (ed) (2009) Exascale software study: software challenges in extreme scale systems. Available: http://www.darpa.mil/ipto/personnel/docs/ExaScale_Study_Software.pdf. Accessed 15 Feb 2010

10. Top500 Supercomputer Sites (2009) Home page. Available: <http://top500.org/>. Accessed 15 Feb 2010
11. Topol AW et al (2006) Three-dimensional integrated circuits. IBM J Res Dev 50(4/5):491–506
12. Tsybeskov L, Lockwood DJ, Ichikawa M (2009) Silicon photonics: CMOS going optical. Proc IEEE 97(7):1161–1165

Experimental Parallel Algorithms

► [Algorithm Engineering](#)

Execution Ordering

► [Scheduling Algorithms](#)

Extensional Equivalences

► [Behavioral Equivalences](#)

F

Fast Fourier Transform (FFT)

► [FFT \(Fast Fourier Transform\)](#)

Fast Multipole Method (FMM)

► [N-Body Computational Methods](#)

Fast Poisson Solvers

► [Rapid Elliptic Solvers](#)

Fat Tree

► [Networks, Multistage](#)

Fault Tolerance

HANS P. ZIMA, ALLEN NIKORA
California Institute of Technology, Pasadena, CA, USA

Definition

Fault tolerance denotes the capability of a system to adhere to its specification and deliver correct service in the presence of faults.

Discussion

Introduction

Modern society relies increasingly on highly complex computing systems across a wide spectrum of applications, ranging from commercial transactions to the control of transportation and communication systems,

the electrical power grid, nuclear reactors, and space missions. In many cases, human lives and huge material values depend on the well-functioning of these systems even under adverse conditions. Despite the successes in verification and validation technology over past decades, theoretical as well as practical studies convincingly demonstrate that large systems typically *do* contain design faults, even when subject to the strictest development disciplines. Moreover, even a perfectly designed system may be subject to external faults, such as radiation effects and operator errors. As a consequence, it is essential to provide methods that avoid system failure and maintain the functionality of a system, possibly with degraded performance, even in the case of faults. This is called *fault tolerance*.

A *fault* is defined as a defect in a system that may cause an *error* during its operation. If an error affects the service to be provided by a system, a *failure* occurs. Fault-tolerant systems were built long before the advent of the digital computer, based on the use of replication, diversified design, and federation of equipment. In an article on Babbage's difference engine published in 1834, Dionysius Lardner wrote [29]: "The most certain and effectual check upon errors which arise in the process of computation is to cause the same computations to be made by separate and independent computers; and this check is rendered still more decisive if they make their computation by different methods." Early fault-tolerant computers include NASA's *Self-Testing-and-Repairing* (STAR) computer, developed for a 10-year mission to the outer planets in the 1960s, and the computers onboard the Jet Propulsion Laboratory's Voyager systems. Today, highly sophisticated fault-tolerant computing systems control the new generation of fly-by-wire aircraft, such as the Airbus and Boeing airliners, protecting against design as well as hardware faults. Perhaps the most widespread use of fault-tolerant computing has been in the area of commercial transactions systems, such as automatic teller machines and airline reservation systems.

The focus of this article is on software methods for dealing with faults that may arise in hardware or software systems. Before entering into a discussion on fault tolerance, the broader concept of *dependability* is explored below.

Dependability

Fault tolerance is one important aspect of a system's **dependability**, a property that has been defined by the IFIP 10.4 Working Group on Dependable Computing and Fault Tolerance as the "*trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers.*" Whereas fault tolerance deals with the problem of ensuring correct system service in the presence of faults, the notion of dependability includes also methods for removing faults during system design, for example, via program verification, or to prevent faults a priori, e.g., by imposing coding rules that outlaw common design faults.

Basic Attributes of Dependability

The *attributes* of dependability specify a set of properties that can be used to assess how a system satisfies its overall requirements. A system's dependability specification needs to include the requirements for each of the individual attributes in terms of the acceptable frequency and severity of service failures for specified classes of faults and the system's operating environment. Short definitions for a typical list of attributes are provided below; a detailed discussion can be found in [3, 4, 31].

The *reliability*, $R(t)$, of a system at a time, t , is specified as the probability that the system will provide correct service up to time t . For example, the reliability required for aircraft control systems has been specified as $R(t) > 1 - 10^{-9}$ for $t = 1\text{ h}$ (this is often informally characterized as "9 nines"). The *mean time to failure (MTTF)* is the expected time that a system will operate correctly before failure. System *availability* at a time, t , is defined as the probability that the system can provide correct service at time t . The *safety* of a system is characterized by the absence of catastrophic consequences on its user and the environment it is operating in. More formally, it can be defined as the probability to fail in a *controlled*, or *safe* manner (see section "►Controlled Failure"). *Confidentiality* is a

system's capability to prevent the unauthorized disclosure of information. *Maintainability* characterizes the ability of a system to undergo modifications, including repairs: one specific aspect is the time it takes to restore a failed system to normal operation.

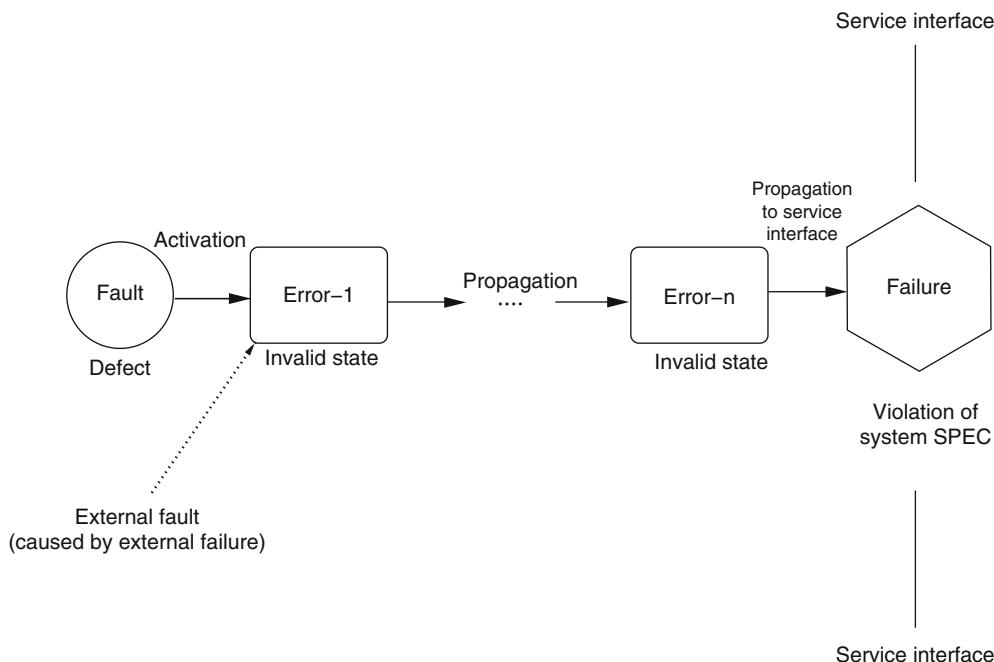
Threats

A *system* is associated with a *specification*, which describes the service the system is intended to provide to its user (a human or another system). It consists of one or more interacting *components*. A *threat* is any fact or event that negatively affects the dependability of a system. Threats can be classified as faults, errors, or failures; their relationship is illustrated by the *fault-error-failure* chain [3] shown in Fig. 1.

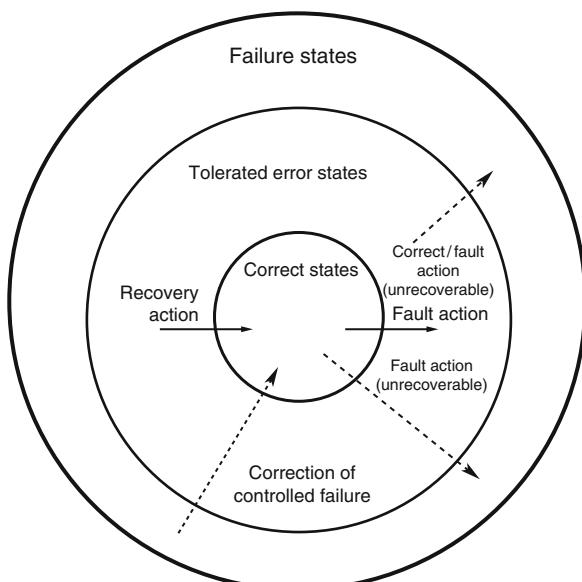
A *fault* is a defect in a system. Faults can be *dormant* – e.g., incorrect program code that is not executed – and have no effect. When *activated* during system operation, a fault leads to an *error*, which is an illegal system state. A fault inside a component is called *internal*; an *external* fault is caused by a failure propagated from another component, or from outside the system. Errors may be propagated through a system, generating other errors. For example, a faulty assignment to a variable may result in an error characterized by an illegal value for that variable; the use of the variable for the control of a for-loop can lead to ill-defined iterations and other errors, such as illegal accesses to data sets and buffer overflows. A *failure* occurs if an error reaches the service interface of a system, resulting in system behavior that is inconsistent with its specification.

The *execution* of a system can be modeled by a sequence of *states*, with state transitions being caused as the result of atomic *actions*. In a first approximation, the set of all system states can be partitioned into *correct states* and *error states*. By separating those error states that allow a recovery from those that represent system failure the set of all error states is further partitioned into *tolerated error states* and *failure states* (Fig. 2). The transitions between these state categories can be described by classifying unambiguously each action as a *correct*, *fault*, or *recovery* action [17]:

- A correct action, executed in a correct state, results in a correct state



Fault Tolerance. Fig. 1 Threats: the fault-error-failure chain



Fault Tolerance. Fig. 2 State-space partitioning

- A fault action, executed in a correct state, results in an error state (tolerated or failure)

- A correct or fault action, executed in a tolerated error state, results in an error state (tolerated or failure)
- A recovery action, executed in a tolerated error state, results in a correct state.
- For a class of systems that are referred to as *fail-controlled* recovery from failure is possible using special protocols (see section “Controlled Failure”).

A system is *fault tolerant* if it never enters a failure state: this means that errors may occur, but they never reach the service boundary of the system and always allow recovery to take place. The implementation of fault tolerance in general implies three steps: error detection, error analysis, and recovery.

The main categories of faults are physical, design, and interaction faults. A *physical fault* manifests itself by malfunctioning of hardware, which is not the result of a design fault. Examples include a *permanent fault* caused by the breakdown of a processor core due to overheating, a *transient fault* caused by radiation and resulting in a bitflip in a register or memory, or a fault caused by a human operator.

Design faults may originate in either hardware or software. An illegal condition in a while loop leading to infinite iteration or an assignment involving incompatible variable types are examples for software design errors.

Interaction faults occur during system operation and are caused by the environment in which the system operates. They include illegal input data or operator errors as well as natural faults caused by radiation hitting the system.

In addition to these main categories, faults can be characterized along a set of additional properties, often across the above classes. This includes the *domain* of a fault – hardware or software, its *intent* – accidental or malicious (e.g., faults caused by viruses, worms, or intrusion), or its *persistence* – hard or soft. A *hard fault* (such as a device breakdown) makes a component permanently unusable. In contrast, a *soft fault* is a temporary fault that can be transient or intermittent. A component affected by such a fault can be reused after the resulting error has been processed. For example, a bitflip in a register caused by a proton hitting a circuit is a transient fault referred to as a *Single Event Upset (SEU)*.

Reflecting the diversity of faults, the failures caused by them can be characterized by a broad range of nonorthogonal properties. These include their *domain* – value or timing, the capability for *control*, *signalling* – signalled or unsignalled failures, and their *consequences*, ranging from minor to catastrophic.

Fault Prevention

Fault prevention addresses methods that prevent faults to being incorporated into a system. In the software domain, such methods include restrictive coding standards that avoid common programming faults, the use of object-oriented techniques such as modularization and information hiding, and the provision of high-level programming languages and abstractions. Firewalls are mechanisms for preventing malicious faults. An example for hardware fault prevention includes shielding against radiation-caused faults [36].

Fault Removal

Despite the existence of fault tolerance methods, a software system's dependability will tend to increase with

the number of faults identified and removed during its development. *Fault removal* refers to the set of techniques that eliminate faults during the design and development process. *Verification and Validation (V&V)* are important in this context: Verification refers to methods for the review, inspection, and test of systems, with the goal of establishing that they conform to their specification. Validation checks the specification in order to determine if it correctly expresses the needs of the system's users.

Verification methods can be classified as either *dynamic* or *static*, depending on whether or not they involve execution of the underlying program. Both techniques require that a system's functional and behavioral specification be transformed into a mathematically based specification language, such as first-order logic [14]. System developers also need to specify a set of *correctness properties*, which must be satisfied. For example, a correctness property for a camera mounted on a robotic planetary exploration spacecraft would be that the camera's lens cover must be closed if the camera is pointed within a given angle of the Sun.

Static verification of a program implies a static proof – performed *before execution* of the program – that for all legal inputs the program conforms to its specification. Early techniques include Hoare's logic and Dijkstra's predicate transformers [12, 20]. Theorem provers use mathematical proof techniques to provide a rigorous argument that the correctness properties are satisfied. Some theorem provers, such as the early versions of the Boyer-Moore theorem prover [6], are designed to operate in a completely automated fashion. Others, such as the Prototype Verification System (PVS) [9], are interactive, requiring the user to "steer" the proof. *Model checkers* determine whether a system's correctness properties can be violated by exploring its computational state space. If a violation is detected, the model checker returns a counterexample describing how the violation can occur. This also means that model checkers can be used to generate test cases for the implemented system [16]. Model checkers are able to identify types of faults associated with multithreaded systems such as deadlocks, lack of progress cycles, and race conditions. The more widely used model checkers include the SPIN model checker [21], UPPAAL [2], and SMV (Symbolic Model Verifier) [32].

Although static verification techniques can identify faults early in a system's development, these techniques face a number of theoretical and practical challenges and limitations. Specifically, many decision problems are undecidable, that is, there is no general method that provides a complete solution [22]. Other problems, for which solution algorithms exist – including many graph problems, are *NP-complete*, that is, computationally intractable due to exponential complexity. Finally, methods such as model checking often need to deal with the scalability challenge, that is, an exploding state space.

Furthermore, substantial effort can be required to create the formal specifications to which these tools will be applied. Learning the specification language can require substantial effort as well; even more important is learning the skill of abstracting from nonessential details of the system's functionality and behavior. For these reasons, these techniques are usually applied only to the critical elements of a system (e.g., the fault detection, identification, and repair component of onboard spacecraft control software, or fault response systems for nuclear power plants).

Dynamic verification technologies are based on the actual execution of the program. Most important are *test* methods. The test of a program for a specific input either demonstrates correctness of the program *for this specific input*, or results in an error. Thus, a specific test execution can either prove that the program operates correctly for the selected input, or it identifies an error. Thus, as already noticed by Edsger Dijkstra in 1972, tests can prove or disprove the *existence* of a fault but never the *absence of all* faults [11].

Controlled Failure

Non-fault tolerant systems may be *fail-controlled*, meaning that they fail only in specific, predefined modes, and only to a manageable extent, avoiding complete disruption. Rather than providing the capability of resuming normal operation, a failure in such a system puts it into a state from which recovery is possible after the failure has been detected and identified.

As an example, the onboard software controlling robotic planetary exploration spacecraft for those portions of a mission during which there is no critical

activity (such as detumbling the spacecraft after launch or descent to a planetary surface) can be organized as a fail-controlled system. When a fault is detected during operation, all active command sequences are terminated, components inessential for survival are powered off, and the spacecraft is positioned into a stable, sun-pointed attitude. Critical information regarding its state is transmitted to ground controllers via an emergency link; rebooting and restoring the health of the spacecraft is then delegated to controllers on Earth.

Systems that preserve continuity of service can be significantly more difficult to design and implement than fail-controlled systems. Not only is it necessary to determine that a fault has occurred, the software must be able to determine the effects of the fault on the system's state, remove the effects of the fault, and then place the system into a state from which processing can proceed.

Fault Tolerance

General Methods for Software Fault Tolerance

Ensuring fault tolerance, that is, continuity of service in the presence of faults, requires replication of functionality. Many of the techniques used to achieve replication of functionality can be categorized as either *design diversity* or as *self-checking software*. Using *design diversity* to tolerate faults in the design of a software system requires that there be at least two *variants* of a system (i.e., different designs and/or implementations based on a common specification), a *decider* to provide an error-free result from the variants' execution, and an explicit specification of decision points in common specification (i.e., specification of when decisions have to be performed, and specification of the data upon which decisions have to be performed). Two well-known types of design diversity are *recovery blocks* and *N-version programming*.

Recovery Blocks

Recovery blocks [34] make use of sequentially applied design diversity. In this case, the decider is an acceptance test that checks the computation results against context-specific criteria to determine whether the result can be accepted as correct. If the acceptance test decides

that the result is correct, the system continues execution and does not attempt to apply any corrective action. On the other hand, if the acceptance test indicates that the result is incorrect, the result is discarded and a variant (termed an “alternate” for recovery blocks) is applied to the inputs to retry the computation. If all of the alternates have been attempted and no results have met the acceptance criteria, additional steps (such as exception handling, see section “Exception Handling”) are required to recover from the fault. A Petri-net model of a recovery block is shown in Fig. 3.

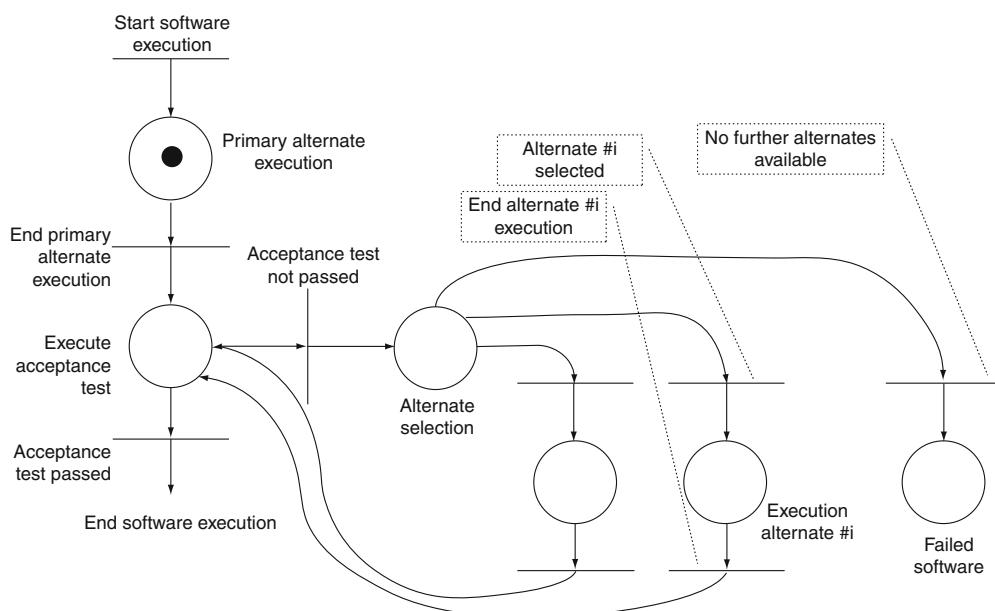
N-Version Programming

For *N-version programming* [5], design diversity is applied by executing all of the variants (termed *versions*) in parallel. The decider then votes on all of the results: if a majority of the results meet the decider’s criteria for correctness, the system is considered to be operating normally and continues execution. Otherwise, further action is required to recover from the fault. A Petri-net model for N-version programming is shown in Fig. 4. A variant of N-version programming, known as back-to-back testing [42], keeps only one of the *N* versions in the fielded system, but uses all of the versions during testing. If a defect is found in one of the *N* versions during test, the other versions are analyzed to determine

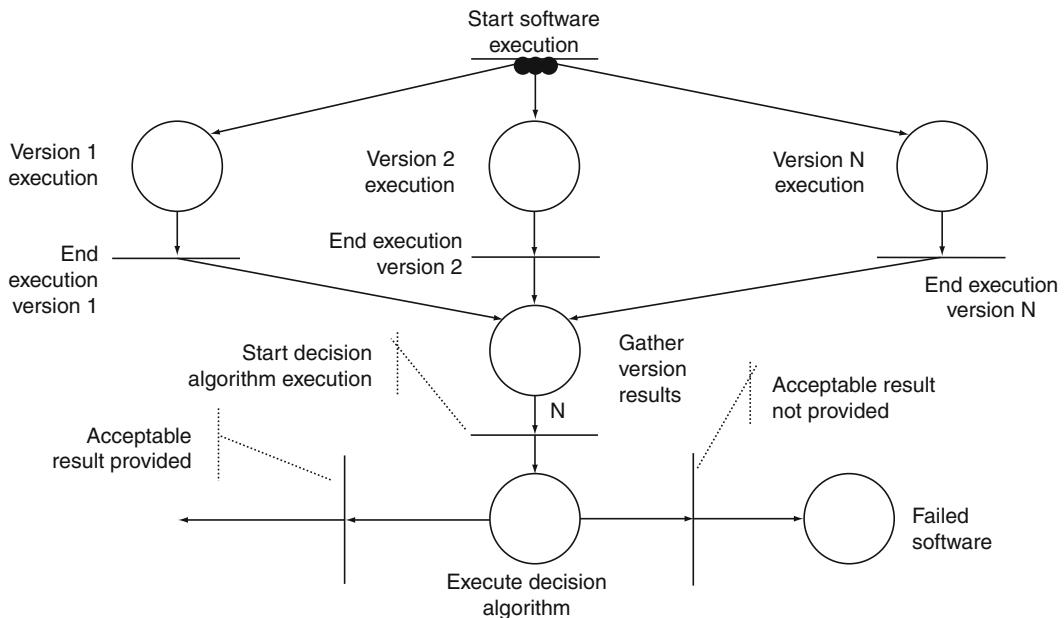
whether they also have that defect. The goal of back-to-back testing is to improve the reliability of a fielded system without incurring the memory space or execution time penalties that might be associated with *N*-version programming. However, fielding only a single version removes the fault-tolerance capability associated with *N*-version programming.

N-Self-Checking Programming

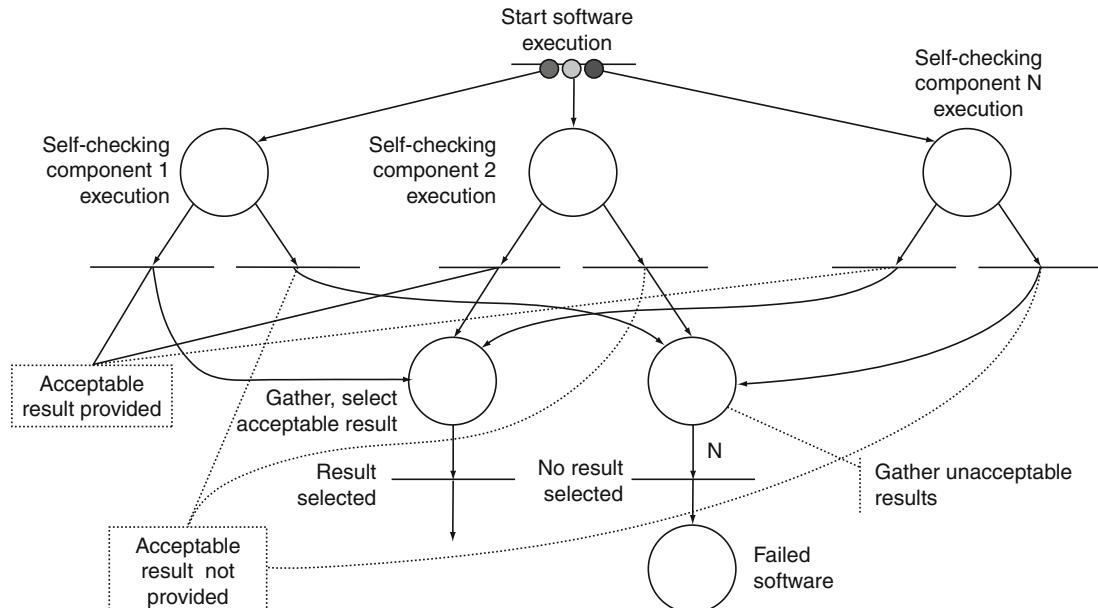
N-self-checking programming [28] combines the idea of an acceptance test from recovery blocks with the parallel execution of $N \geq 2$ variants. In this case, each of the variants has an acceptance test associated with it. The acceptance tests associated to each variant or the comparison algorithms associated to a pair of variants can be the same, or specifically derived for each of them from a common specification. As the variants are executed, the acceptance test associated with each variant determines whether the result meets the criteria for correctness. From the set of correct results, one is selected as the collective output for all the variants and execution proceeds normally. If the set of correct results is empty, the system is considered to have failed and further action is required to restore it to nominal operation. A Petri-net model for *N*-self-checking software is shown in Fig. 5.



Fault Tolerance. Fig. 3 Petri-net model for recovery block



Fault Tolerance. Fig. 4 Petri-net model for N-version programming



Fault Tolerance. Fig. 5 Petri-net model for self-checking programming

For self-checking software components that are based on the association of two variants, one variant only may be written for the purpose of fulfilling the functions expected from the component, while the other variant can be written as an extended acceptance

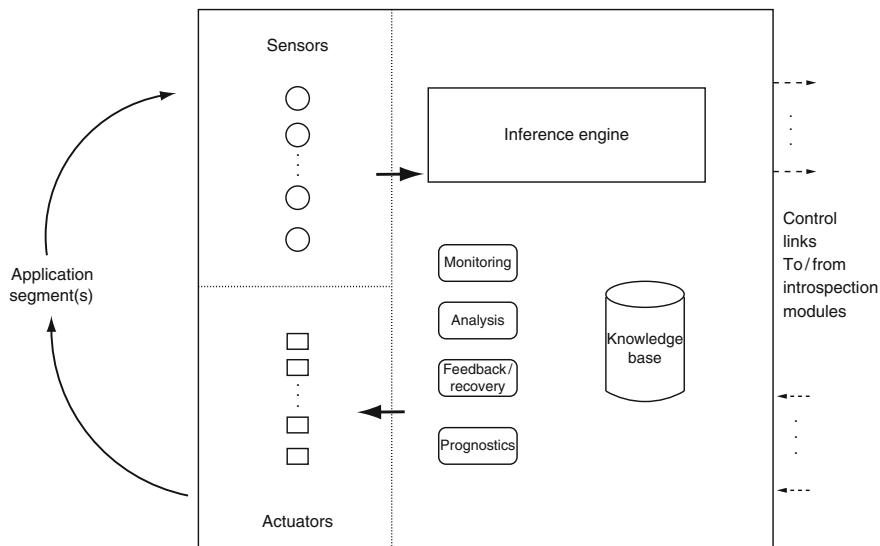
test. For example, it could perform the required computations with a different accuracy, compute the inverse function from the primary variant's results (if possible), or exploit application-specific intermediate results of the primary variant. *Algorithm-based fault tolerance*

(ABFT) belongs to this category [19]. Many of the fault-tolerant systems encountered in real life are based on self-checking software. For example, the software for the Airbus A310 and A320 flight control systems and the Swedish railways' interlocking system are based on parallel execution of two variants – in the latter case, the variants' results are compared, and operation is stopped if the results of the variants disagree.

It is important to note that the actual effectiveness of fault-tolerance techniques based on multiple versions may not be as high as the theoretical limits, because even independent programming teams may make similar errors [25]. However, recent work indicates that in spite of this limitation, N-version techniques can still be a viable way of tolerating faults [7].

systems require support for *adaptive fault tolerance*, based on a *fault model* and a characterization of applications and their components with respect to their overall importance, exposure to faults, and requirements for fault tolerance. An *introspection-based approach* can support such an environment by providing a capability for execution-time monitoring and self-checking, analysis, and feedback-oriented recovery [23]. An introspection module, as illustrated in Fig. 6, is the core element of such a system, connected to the application via a set of software *sensors* for receiving input from, and a set of *actuators* for generating feedback to an application module. An inference engine connected to a knowledge base controls the operation of the introspection module. Introspection modules are organized into an *introspection graph* that reflects the organization and fault-tolerance requirements of the application.

The application-specific error detection, analysis, and recovery actions of an introspection module can be controlled via *assertions* inserted in a program, directions for generation of specific checks, and corresponding recovery actions. This information can be provided by a user or automatically generated from the source program or a pre-existing knowledge basis characterizing properties of the application [23, 44]. A detailed analysis of the range of fault-tolerance requirements for a complex real-time environment is provided in [24].



Fault Tolerance. Fig. 6 Introspection module

Fault Tolerance in Concurrent Systems

Safety and Liveness Properties

A *concurrent system* is introduced as an interconnected collection of a finite number of autonomous *processes*. Processes can either interact via a global *shared memory*, or communicate via *message-passing* over a network. No assumptions are made about individual and relative speeds of processes and delays involved in access to shared memory or message-passing communication, except for the requirement that the speed of active processes is finite and positive, and message delays are finite. (Thus, this model does not cover real-time constraints.) An *execution* of such a system can be modeled by a sequence of system states, with a transition between successive states taking place as a result of an atomic action in a process or a communication step.

Lamport [27] introduced two key types of *correctness properties* for parallel systems: safety properties and liveness properties. A *safety* property can be described by a predicate over the state that must hold for *all* executions of the system. It rules out that “bad things happen.” An example for a safety property is the requirement of an airline reservation system to provide *mutual exclusion* for accesses to any record representing a specific flight on a given day. A more mundane example is the requirement for a system controlling the traffic lights of a street intersection that the lights for two crossing streets may never be green at the same time. A *liveness* property requires that each process, which in principle is able to work, will be able to do so after a finite time. This property can be expressed via a predicate that must be *eventually* satisfied, guaranteeing that “a good thing will finally happen.” For instance, if a set of processes compete for a resource, each of the processes should be able to acquire it after a finite amount of time. Examples for violation of liveness include the prevention of a process to reach regular termination or to provide a specified service. Another example is a *deadlock* involving two or more processes, which cyclically block each other indefinitely in an attempt to access common resources. Alpern and Schneider have shown that *every possible* property of the set of executions can be expressed as a conjunction of safety and liveness properties [1].

The manifestation of faults in a concurrent system, and related methods for their toleration depend largely

on detailed system properties and the programming model under consideration. In the next subsection, a number of typical problems in concurrent systems are shortly discussed; a comprehensive treatment of such issues can be found in [35]. In addition to the applicability of the general methods discussed in section “General Methods for Software Fault Tolerance”, fault tolerance can be often achieved using specialized techniques.

Example Problems

Mutual exclusion constrains access to a shared resource – for example, a variable, record, file, or output device – such that at any time at most one process may access the resource (a safety property). Any correct solution must also guarantee that each process requesting the resource is granted access after a finite waiting period (a liveness property). A simple method for implementing mutual exclusion in a shared memory system is via a binary semaphore [10] that controls access to the resource. The corruption of the semaphore value (by any process or as a result of a hardware fault) destroys the safety property of the algorithm. An error in the semaphore implementation may destroy the liveness property. The mutual exclusion problem is discussed in detail in [10].

State-based synchronization makes the progress of a process dependent on a condition in its environment. For example, in a simple version of the *producer-consumer problem*, the producer and consumer are autonomous processes operating asynchronously, with a cyclic buffer between them acting as a temporary storage for data items: in each cycle of the producer, it generates one data item and places it into the buffer; conversely, the consumer removes in each cycle one item from the buffer and then processes it. Wrong synchronization (as a result of a design error or a corruption of variables used for the coordination) can lead to a number of errors, including the corruption of a buffer element, an attempt of the producer to write into a full buffer, or an attempt of the consumer to read from an empty buffer. The result is a violation of the safety property, and possibly of liveness.

A *deadlock* is a system state where two or more processes block each other indefinitely in an attempt to access a set of resources. In a typical example, process p_1 holds resource r_1 and requests resource r_2 , while process p_2 holds r_2 and requests r_1 . This is a violation of the liveness requirement. It can be addressed in a number

of ways, including deadlock prevention –, by prioritizing resources and allowing access requests only in rising priority order, or by periodically checking for a deadlock and taking a recovery action if it occurs. A different example for a liveness violation is represented by the famous *dining philosophers problem*, where two processes conspire to block a third process indefinitely from accessing a common resource. Problems of this kind can be avoided by defining an upper limit for the number of unsuccessful resource requests, and prioritizing the blocked process once this limit is reached.

In a system where processes communicate over a network, a *network fault* can propagate into a number of different faults in the process system. These include (1) transmission of wrong message content, (2) sending a message to the wrong destination, and, (3) total loss of the message, resulting in a possibly indefinite delay of the intended receiver. A way to deal with the last of these faults is the use of a *watchdog timer* that regularly checks the progress of processes.

Byzantine faults are arbitrary value or timing faults that can occur as a result of radiation hitting a circuit, or of a malicious intruder. Under certain conditions, such faults can be tolerated using a sophisticated form of replication [18].

Checkpointing and Rollback Recovery in Message-Passing Systems

This section deals with fault tolerance for long-running applications in message-passing systems implemented on top of a reliable communication layer (which has to be provided by the underlying hardware and software layers of the network). The state of such systems consists of the set of local states of the participating processes, extended by the state of the communication system, which records the messages that are in-flight at any given time. Consistency of the global state implies that the dependences created between processes as a result of message transmissions must be observable in the state: more specifically, at the time a process records receipt of a message the state of the sender must reflect the previous sending of that message.

A recovery approach based on checkpointing and rollback requires a stable storage device that is not affected by faults in the system. At a *checkpoint*, a process saves recovery information to this storage device during fault-free operation; after occurrence of an error

(e.g., as a result of a failing process), the information stored at the checkpoint can be used to restart the process. Recovery can be managed by the application; here a system-supported approach is discussed.

There are two categories of recovery-based protocols. The first relies only on the use of checkpoints, whereas the second approach in addition performs logging of all nondeterministic events, which allows the *deterministic* regeneration of an execution after the occurrence of a fault (which is not possible if only checkpoints are used). This is important in applications that heavily communicate with the outside world (which cannot be rolled back!). However, from the viewpoint of today's large-scale scientific applications, the first approach, which will just be denoted as *checkpointing* is more relevant, and will be the sole focus in the rest of this section.

There are two main methods for performing checkpointing, which are characterized as uncoordinated or coordinated. In *uncoordinated checkpointing*, each process essentially decides autonomously when to take checkpoints. During recovery, a consistent global checkpoint needs to be found taking into account the dependences between the individual process checkpoints. Uncoordinated checkpointing has the advantage of allowing a process to take into account local knowledge, for example, for minimizing the amount of data that needs to be saved. However, there are serious disadvantages, including the *domino effect* that can result in rollback propagation all the way to the beginning of the computation and the need to record multiple checkpoints in each process, with the necessity of periodical cleanups via garbage collection.

As a result of these shortcomings, the dominating method used in today's systems, and particularly in supercomputers, is *coordinated checkpointing*. In this approach, a global synchronization step involving all processes is used to form a consistent global state. Only one checkpoint is needed in each process, and no garbage collection is required. A straightforward centralized two-stage protocol for the creation of a checkpoint requires blocking of all communication while the protocol executes. In a first step, a *coordinator process* broadcasts a *quiesce message* to all processes requesting them to stop. After receiving an acknowledgment for this request from all processes, the coordinator broadcasts a *checkpoint message* that requests processes to take their checkpoint and acknowledge completion. Finally,

after receipt of this acknowledgment from all processes, the coordinator broadcasts a *proceed message* to all processes, allowing them to continue their work. On a failure in a process, the whole application rolls back to the last checkpoint and resumes execution from there.

Coordinated checkpointing provides a relatively simple approach to recovery. Drawbacks include the overhead for communication in the checkpointing and recovery protocols, and the time required for storing the relevant data sets to a stable storage device, with the obvious consequences for scalability. In fact, a recent study on exascale computing [15] shows that checkpointing along the above protocol seems no longer to be feasible, due to the existence of billions of threads in such a system. A potential solution would require new storage technology for saving the recovery information.

A comprehensive treatment of rollback-recovery protocols, including log-based approaches, can be found in [13]. Coordinated checkpointing for large-scale supercomputers is discussed in detail in [43], including a study of scalability and an approach to deal with failures during the checkpointing and recovery phases.

Fault Injection Testing

Fault injection testing refers to a technique for revealing defects in a system's ability to tolerate faults [40]. Fault injection inserts faults into a system or mimicks their effects, combined with a strategy that specifies the type and number of faults to be injected into system components, the frequency at which faults are to be injected, and dependencies among fault types.

A number of fault injectors have been developed [37, 38, 40], which allow faults to be inserted into a computing system's memory or processor registers while the system is running, in a manner that is transparent to the executing application and system software. This allows developers of fault-tolerant systems to observe the system operating in an environment closely related to the expected environment during fielded use. As faults are injected into the system, system engineers are able to observe the immediate effects of each fault as well as the way in which it propagates through the system. When conducting fault injection testing, it is necessary to design test strategies that will cause the system under test to execute its fault-handling capabilities under a wide variety of conditions. One particular strategy, implemented by the Ballista automated testing

methodology [26], is to change the parameters passed to software modules and observe the result. In Ballista, a test passes if the system does not hang or crash, ignoring the validity of output values produced. The JPL Implementation of a Fault Injector (JIFI) [39] toolset is typical of modern software-controlled fault injectors. It allows automated campaigns of random uniformly distributed bitflip faults into application registers and/or memory space and registers. In order to characterize the fault injection results, it is necessary to verify the final result and classify its effect on the system with respect to its liveness properties and the functional correctness of the output. The main goal of this methodology is to characterize an application's sensitivity to transient faults. For example, it is possible to determine whether the application is more likely to crash or produce incorrect results, or which subroutines of the program are most vulnerable.

Exception Handling

An *exception* is a special event that signals the occurrence of one among a number of predetermined faults in a program. Exception handling can be considered as an additional fault-tolerance technique in that it provides a structured response to such events.

There are two types of exception handling, resumption and termination [8]. In resumption, if an exception is encountered at a given command in a program, the program is temporarily halted and control is transferred to a handler associated with that program. The handler performs the computations required to mitigate the effects of the exception and then transfers control back to the program. The handler may return to the command immediately following the one at which the exception was encountered, or to a different location that may depend on the details of the program's state when the exception was signalled. By contrast, termination simply leads to the (controlled) halting of the program in which the exception was observed.

Although resumption has a potential advantage in that the interrupted program may be resumed upon successful execution of the handler, there are also advantages associated with choosing termination [8]. For example, with termination a programmer is encouraged to recover a consistent state of a module in which an exception is detected before signalling it, so that further calls to the module's procedures find the module state consistent. With resumption, a programmer

does not know if control will come back after signalling an exception. In addition, recovering a consistent state before signalling (e.g., for example by undoing all changes made since the procedure start) defeats the purpose of resumption, which is to save the work done so far between the procedure entry and the detection of the exception. If a consistent state is not recovered, the handler may never resume execution of the module after the signalling command, and the module will remain in the intermediate, most likely inconsistent, state that existed when the exception was detected, meaning that further module calls can lead to additional exceptions and failures.

Future Directions

As computing systems become increasingly embedded in and critical to society, more effective capabilities to deal with faults induced by exceptional environmental conditions, specification and design defects, or both, will be required. As an example, consider implantable medical devices such as insulin pumps or defibrillators. These systems already execute safety-critical software; future versions of these devices intended to respond to a wider set of medical conditions will be substantially more complex than those in operation today. New challenges for fault tolerance will also come from the increased degree of automation in transportation systems (railways, airplanes, automobiles) as well as extreme-scale systems for large-scale simulation with billion-way parallelism [15]. Last but not least, future robotic space missions in our solar system and beyond will demand an unprecedented degree of autonomy when operating in environments in which direct communication with Earth will no longer be possible within practical time intervals. The additional complexity of these systems and the uncertainties of the environments in which they operate will introduce more opportunities for specification and design defects [33], for which effective fault-tolerant techniques must be developed and deployed.

Related Entries

- [Checkpointing](#)
- [Compilers](#)
- [Deadlocks](#)
- [Distributed-Memory Multiprocessor](#)

- [Exascale Computing](#)
- [Race Detection Techniques](#)
- [Synchronization](#)

Bibliographic Notes and Further Reading

The basic concepts of dependability are discussed in detail in the seminal work of Avizienis, Laprie, Randell, and Landwehr [3, 4]. A comprehensive discussion of software fault tolerance can be found in the collection of articles edited by Lyu [30]. In [31] dependability and fault tolerance are discussed with a focus on hardware issues. Faults and fault tolerance for algorithms in distributed systems, with a detailed treatment of provably solvable and unsolvable issues can be found in [41].

The premier conference on dependability is the annual *IEEE/IFIP International Conference on Dependable Systems and Networks* (DSN), with DSN-40 held in Chicago in 2010. This conference was established in 2000 by combining the IEEE-sponsored *International Symposium on Fault-Tolerant Computing* (FTCS), which had been held since 1971, with the *Working Conference on Dependable Computing for Critical Applications* (DCCA) sponsored by IFIP WG 10.4. Other conferences related to dependability include the *International Symposium on Software Reliability Engineering* (ISSRE) organized by the IEEE Computer Society and the *Reliability, Availability, and Maintainability Symposium* (RAMS) organized by the IEEE Reliability Society.

In addition to many journals discussing fault tolerance in the context of more general or related areas such as software engineering, programming languages and systems, and system security, the *IEEE Transactions on Dependable and Secure Computing* (TDSC) have a strong focus on dependability and fault tolerance. The bibliography in [17] provides a good overview of publications in the field until 1999.

Bibliography

1. Alpern B, Schneider FB (1985) Defining liveness. *Inf Process Lett* 21(4):181–185
2. Amnell A, Behrmann G, Bengtsson J, D'Argenio PR, David A, Fehnker A, Hune T, Jeannet B, Larsen KG, MÄoller MO, Pettersson P, Weise C, Wang Yi (2001) UPPAAL—now, next, and future. In: Proceedings of modelling and verification of parallel processes (MOVEP'2k), June 2000. LNCS tutorial 2067, pp 100–125

3. Avizienis A, Laprie JC, Randell B (2000) Fundamental concepts of dependability, UCLA CSD Report No. 010028, University of California, Los Angeles
4. Avizienis A, Laprie JC, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1(1):11–33
5. Avizienis AA (1995) The methodology of N-version programming. In: Lyu MR (ed) *Software fault tolerance*. Wiley, Chichester
6. Boyer RS, Kaufmann M, Moore JS (1995) The Boyer-Moore theorem prover and its interactive enhancement. *Comput Math Appl* 29(2):27–62
7. Cai X, Lyu MR, Vouk MA (2005) An experimental evaluation of reliability features of N-version programming. In: Proceedings of the IEEE international symposium on software reliability engineering, pp 161–170
8. Cristian F (1995) Exception handling and tolerance of software faults. In: Lyu MR (ed) *Software fault tolerance*. Wiley, New York, pp 81–107
9. Crow J, Owre S, Rushby J, Shankar N, Srivas M (1995) A tutorial introduction to PVS. <http://www.csl.sri.com/papers/wift-tutorial/>
10. Dijkstra EW (1968) Co-operating sequential processes. In: Genays F (ed) *Programming languages*. Academic, New York, pp 43–112
11. Dijkstra EW (1972) Notes on structured programming. In: Dahl OJ, Dijkstra EW, Hoare CAR (eds) *Structured programming*. Academic, London, pp 1–82
12. Dijkstra EW (1976) *A discipline of programming*, 2nd edn. Prentice Hall, Englewood Cliffs
13. Elnozahy ENM, Alvisi L, Wang Yi-Min, Johnson DB (2002) A survey of rollback recovery protocols in message-passing systems. *ACM Comput Surv* 34(3):375–408
14. Enderton HB (2001) *A mathematical introduction to logic*, 2nd edn. Academic, San Diego
15. Kogge P et al (2008) ExaScale computing study: technology challenges in achieving ExaScale systems. Technical report, DARPA information processing techniques Office (IPTO). http://www.notur.no/news/inthenews/files/exascale_final_report1002, September 2008
16. Gargantini A, Heitmeyer CL (1999) Using model checking to generate tests from requirements specifications. In: Proceedings of the joint 7th european software engineering conference and the 7th ACM SIGSOFT symposium on the foundations of software engineering, Toulouse, France, September 1999
17. Gärtner FC (1999) Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput Surv* 31(1):1–26
18. Goldberg D, Li M, Tao W, Tamir Y (2001) The design and implementation of a fault-tolerant cluster manager. Technical report CSD-010040, Computer Science Department, University of California, Los Angeles, California, USA, October 2001
19. Gunnels JA, Katz DS, Quintana-Ort ES, van de Geijn RA (2001) Fault-tolerant high-performance matrix multiplication: theory and practice. In: Proceedings of international conference on dependable systems and networks, (DSN-2001), Goteborg, Sweden, June/July 2001
20. Hoare CAR (1969) An axiomatic basis for computer programming. *Commun ACM* 12(10):576–583
21. Holzmann GJ (2003) *The SPIN model checker*. Primer and reference manual. Addison-Wesley, New York
22. Hopcroft JE, Motwani R, Ullman JD (2006) *Introduction to automata theory, languages, and computation*, 3rd edn. Addison Wesley Higher Education, New York
23. James M, Shapiro A, Springer P, Zima H (2009) Adaptive fault tolerance for scalable cluster computing in space. *IJHPCA* 23(3):227–241
24. Kalbarczyk ZT, Iyer RK, Bagchi S, Whisnant K (1999) Chameleon: a software infrastructure for adaptive fault tolerance. *IEEE Trans Parallel Distrib Syst* 10(6):560–579
25. Knight JC, Leveson NG (1986) An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Trans Softw Eng* 12(1):96–109
26. Koopman P (1997) Ballista design and methodology, <http://www.ece.cmu.edu/koopman/ballista/reports/desmthd.pdf>
27. Lamport L (1977) Proving the correctness of multiprocess programs. *IEEE Trans Softw Eng* 3(2):125–143
28. Laprie JC, Arlat J, Beounes C, Kanoun K (1990) Definition and analysis of hardware- and software-fault-tolerant architectures. *Computer* 23(7):39–51
29. Lardner D (1961) Babbage's calculating engine. Edinburgh Review, July 1834. Reprinted. In: Morrison P, Morrison E (eds) *Charles Babbage and his calculating engines*. Dover, New York
30. Lyu MR (ed) (1995) *Software fault tolerance*, vol 12. Wiley, Chichester
31. McCluskey EJ, Mitra S (2004) Fault tolerance. In: Tucker AB (ed) *Computer science handbook*, 2nd edn. Chapman and Hall/CRC Press, Chapter 25, London, UK
32. McMillan KL (1992) Symbolic model checking – an approach to the state explosion problem. Ph.D. Dissertation, Carnegie Mellon University
33. Nikora AP, Munson JC (2006) Building high-quality software predictors. *Softw Pract Exper* 36(9):949–969
34. Randell B, Xu J (1995) The evolution of the recovery block concept. In: Lyu MR (ed) *Software fault tolerance*. Wiley, Hoboken, New Jersey, USA, pp 1–21
35. Schneider FB (1997) *On concurrent programming*. Springer, New York
36. Shirvani PP (2001) Fault-tolerant computing for radiation environments. Technical report 01-6. Ph.D. Thesis, Center for Reliable Computing, Stanford University, Stanford, California 94305, June 2001
37. Critical Software (1995) csXCEPTION. <http://www.criticalsoftware.com/products/services/csexception/>
38. Some RR, Agrawal A, Kim WS, Callum L, Khanoyan G, Shamilian A (2002) Fault injection experiment results in space-borne parallel application programs. In: Proceedings 2002 IEEE aerospace conference, Big Sky, MT, USA, March 2002
39. Some RR, Kim WS, Khanoyan G, Callum L, Agrawal A, Beahan J (2001) A software-implemented fault injection methodology for

- design and validation of system fault tolerance. In: Proceedings of the 2001 international conference on dependable systems and networks (DSN2001), Goteborg, Sweden, pp 501–506, July 2001
40. Stott DT, Floering B, Kalbarczyk Z, Iyer RK (2000) A framework for assessing dependability in distributed systems with lightweight fault injectors. In: Proceedings of the 4th international computer performance and dependability symposium (IPDS'00), IEEE Computer Society, Washington, DC, USA, pp 91–100
41. Gerard Tel (2000) Introduction to distributed algorithms, 2nd edn. Cambridge University Press, Cambridge, UK
42. Vouk MA (1988) On back-to-back testing. In: Proceedings of the third annual conference on computer assurance (COMPASS'88), pp 84–91
43. Wang L, Pattabiraman K, Kalbarczyk Z, Iyer RK (2005) Modeling coordinated checkpointing for large-scale supercomputers. In: Proceedings of the 2005 international conference on dependable systems and networks (DSN'05)
44. Zima HP, Chapman BM (1991) Supercompilers for parallel and vector computers. ACM Press Frontier Series, New York

Fences

- [Memory Models](#)
- [Synchronization](#)

FFT (Fast Fourier Transform)

FRANZ FRANCHETTI¹, MARKUS PÜSCHEL²
¹Carnegie Mellon University, Pittsburgh, PA, USA
²ETH Zurich, Zurich, Switzerland

Synonyms

[Fast algorithm for the discrete Fourier transform \(DFT\)](#)

Definition

A fast Fourier transform (FFT) is an efficient algorithm to compute the discrete Fourier transform (DFT) of an input vector. Efficient means that the FFT computes the DFT of an n -element vector in $O(n \log n)$ operations in contrast to the $O(n^2)$ operations required for computing the DFT by definition. FFTs exist for any vector length n and for real and higher-dimensional data. Parallel FFTs have been developed since the advent of parallel computing.

Discussion

Introduction

The discrete Fourier transform (DFT) is a ubiquitous tool in science and engineering including in digital signal processing, communication, and high-performance computing. Applications include spectral analysis, image compression, interpolation, solving partial differential equations, and many other tasks.

Given n real or complex inputs x_0, \dots, x_{n-1} , the DFT is defined as

$$y_k = \sum_{0 \leq \ell < n} \omega_n^{k\ell} x_\ell, \quad 0 \leq k < n, \quad (1)$$

with $\omega_n = \exp(-2\pi i/n)$, $i = \sqrt{-1}$. Stacking the x_ℓ and y_k into vectors $x = (x_0, \dots, x_{n-1})^T$ and $y = (y_0, \dots, y_{n-1})^T$ yields the equivalent form of a matrix-vector product:

$$y = \text{DFT}_n x, \quad \text{DFT}_n = [\omega_n^{k\ell}]_{0 \leq k, \ell < n}. \quad (2)$$

Computing the DFT by its definition (2) requires $\Theta(n^2)$ many operations. The first fast Fourier transform algorithm (FFT) by Cooley and Tukey in 1965 reduced the runtime to $O(n \log(n))$ for two-powers n and marked the advent of digital signal processing. (It was later discovered that this FFT had already been derived and used by Gauss in the nineteenth century but was largely forgotten since then [9].) Since then, FFTs have been the topic of many publications and a wealth of different algorithms exist. This includes $O(n \log(n))$ algorithms for any input size n , as well as numerous variants optimized for various computing platform and computation requirements. The by far most commonly used DFT is for two-power input sizes n , partly because these sizes permit the most efficient algorithms.

The first FFT explicitly optimized for parallelism was the Pease FFT published in 1968. Since then specialized FFT variants were developed with every new type of parallel computer. This includes FFTs for data flow machines, vector computers, shared and distributed memory multiprocessors, streaming and SIMD vector architectures, digital signal processing (DSP) processors, field-programmable gate arrays (FPGAs), and graphics processing units (GPUs). Just like Pease's FFT, these parallel FFTs are mainly for two-powers n and are adaptations of the same fundamental algorithm to structurally match the target platform.

On contemporary sequential and parallel machines it has become very hard to obtain high-performance DFT implementations. Beyond the choice of a suitable FFT, many other implementation issues have to be addressed. Up to the 1990s, there were many public FFT implementations and proprietary FFT libraries available. Due to the code complexity inherent to fast implementations and the fast advances in processor design, today only a few competitive open source and vendor FFT libraries are available in the parallel computing space.

FFTs: Representation

Corresponding to the two different ways (1) and (2) of representing the DFT, FFTs are represented either as sequences of summations or as factorizations of the transform matrix DFT_n . The latter representation is adopted in Van Loan's seminal book [18] on FFTs and used in the following. To explain this representation, assume as example that DFT_n in (2) can be factored into four matrices

$$DFT_n = M_1 M_2 M_3 M_4. \quad (3)$$

Then (2) can be computed in four steps as

$$t = M_4 x, u = M_3 t, v = M_2 u, y = M_1 v.$$

If the matrices M_i are sufficiently sparse (have many zero entries) the operations count compared to a direct computation is decreased and (3) is called an FFT. For example, DFT_4 can be factorized as

$$DFT_4 = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & -1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & \\ 1 & \\ i & \\ & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \\ 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & \\ 1 & \\ 1 & \\ 1 & \end{bmatrix}, \quad (4)$$

where omitted values are zero. This example also demonstrates why the matrix-vector multiplications in (3) are not performed using a generic sparse linear algebra library, but, since the M_i are known and fixed, by a specialized program.

Conversely, every FFT can be written as in (3) (with varying numbers of factors). The matrices M_i in FFTs are not only sparse but also structured, as a glimpse on (4) illustrates. This structure can be efficiently expressed using a formalism based on matrix algebra and also clearly expresses the parallelism inherent to an FFT.

Matrix formalism and parallelism. The $n \times n$ identity matrix is denoted with I_n , and the *butterfly matrix* is a DFT of size 2:

$$DFT_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \quad (5)$$

The *Kronecker product* of matrices A and B is defined as

$$A \otimes B = [a_{k,\ell} B], \quad \text{for } A = [a_{k,\ell}].$$

It replaces every entry $a_{k,\ell}$ of A by the matrix $a_{k,\ell} B$. Most important for FFTs are the cases where A or B is the identity matrix. As examples consider

$$I_4 \otimes DFT_2 = \begin{bmatrix} 1 & 1 & & & \\ 1 & -1 & & & \\ & & 1 & 1 & \\ & & 1 & -1 & \\ & & & 1 & 1 \\ & & & 1 & -1 \\ & & & & 1 & 1 \\ & & & & 1 & -1 \end{bmatrix},$$

$$DFT_2 \otimes I_4 = \begin{bmatrix} 1 & & & & 1 & & & \\ & 1 & & & & 1 & & \\ & & 1 & & & & 1 & \\ & & & 1 & & & & 1 \\ \hline 1 & & & & 1 & & & \\ & 1 & & & & -1 & & \\ & & 1 & & & & -1 & \\ & & & 1 & & & & -1 \end{bmatrix},$$

with the corresponding dataflows shown in Fig. 1. Note that the dataflows are from right to left to match the order of computation in (3). $I_4 \otimes DFT_2$ clearly expresses block parallelism: four butterflies computing on contiguous subvectors; whereas $DFT_2 \otimes I_4$ expresses vector parallelism: four butterflies operating on interleaved subvectors which is the same as one *vector butterfly* operating on *vectors* of length four as emphasized in Fig. 1(b). More precisely, consider the code for DFT_2 (i.e., $y = DFT_2 x$):

```
y[0] = x[0] + x[1];
y[1] = x[0] - x[1];
```

Then code for $\text{DFT}_2 \otimes I_4$ is obtained by replacing every scalar operation by a four-way vector operation:

```
y[0:3] = x[0:3] + x[4:7];
y[4:7] = x[0:3] - x[4:7];
```

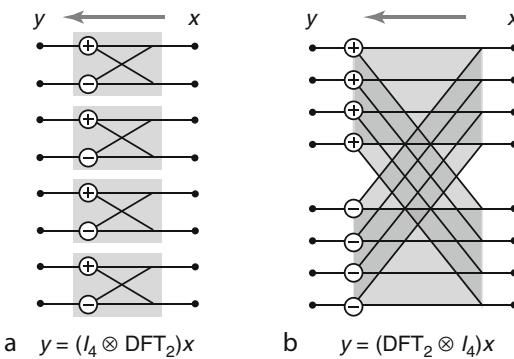
Here, $x[a:b]$ denotes (Matlab or FORTRAN style) the subvector of x starting at a and ending at b . These examples illustrate how the tensor product captures parallelism. To summarize:

block parallelism (n blocks): $I_n \otimes A$, (6)

vector parallelism (n -way): $A \otimes I_n$, (7)

where A is any matrix.

The *stride permutation* matrix L_m^{mn} permutes the elements of the input vector as $in + j \mapsto jm + i$, $0 \leq i < m$, $0 \leq j < n$. If the vector x is viewed as an $n \times m$



FFT (Fast Fourier Transform). Fig. 1 Dataflow (right to left) of a block parallel and its “dual” vector parallel construct (figure from [5])

matrix, stored in row-major order, then L_m^{mn} performs a transposition of this matrix. Further, if P is a permutation (matrix), then $A^P = P^{-1}AP$ is the *conjugation* of A with P .

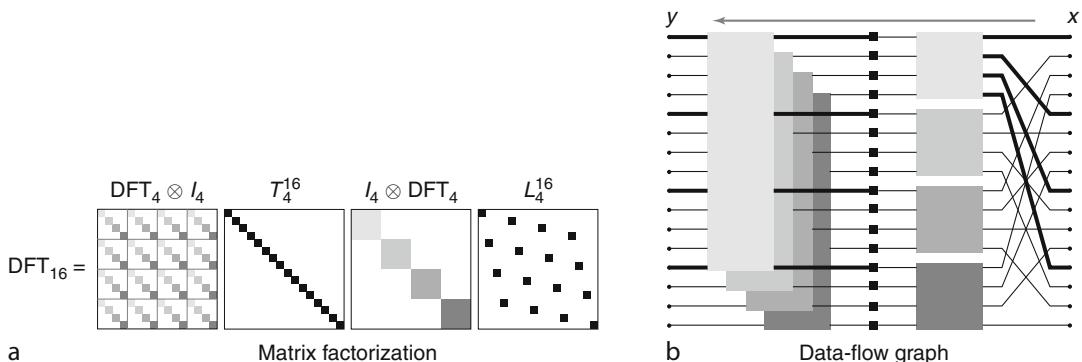
Cooley–Tukey FFT. The fundamental algorithm at the core of the most important parallel FFTs derived in the literature is the general-radix decimation-in-time Cooley–Tukey type FFT expressed as

$$\text{DFT}_n = (\text{DFT}_k \otimes I_m) T_m^n (I_k \otimes \text{DFT}_m) L_k^n, \quad n = km. \quad (8)$$

Here, k is called the radix and T_m^n is a diagonal matrix containing the *twiddle factors*. The algorithm factors the DFT into four factors as in (4), which shows the special case $n = 4 = 2 \times 2$. Two of the four factors in (8) contain smaller DFTs; hence the algorithm is divide-and-conquer and has to be applied recursively. At each step of the recursion the radix is a degree of freedom. For two-power sizes $n = 2^\ell$, (8) is sufficient to recurse up to $n = 2$, which is computed by definition (5).

Figure 2 shows the special case $16 = 4 \times 4$ as matrix factorization and as corresponding data-flow graph (again to be read from right to left). The smaller DFTs are represented as blocks with different shades of gray.

A straightforward implementation of (8) suggests four steps corresponding to the four factors, where two steps call smaller DFTs. However, to improve locality, the initial permutation L_k^n is usually not performed but interpreted as data access for the subsequent computation, and the twiddle diagonal T_m^n is fused with the



FFT (Fast Fourier Transform). Fig. 2 Cooley–Tukey FFT (8) for $16 = 4 \times 4$ as matrix factorization and as (complex) data-flow graph (from right to left). Some lines are bold to emphasize the strided access (figure from [5])

subsequent DFTs. This strategy is chosen, for example, in the library FFTW 2.x and the code can be sketched as follows:

```
void dft(int n, complex *y, complex *x) {
    int k = choose_factor(n);
    // t1 = (I_k tensor DFT_m)L(n,k)*x
    for(int i=0; i < k; ++i)
        dft_iostride(m, k, 1, t1 + m*i, x + i);
    // y = (DFT_k tensor I_m)diag(d(j))*t1
    for(int i=0; i < m; ++i)
        dft_scaled(k, m, precomp_d[i], y + i,
                   t1 + i);
}
// DFT variants needed
void dft_iostride(int n, int istride,
                  int ostride, complex *y, complex *x);
void dft_scaled(int n, int stride,
                complex *d, complex *y, complex *x);
```

The DFT variants needed for the smaller DFTs are implemented similarly based on (8). There are many additional issues in implementing (8) to run fast on a nonparallel platform. The focus here is on mapping (8) to parallel platforms for two-power sizes n .

Parallel FFTs: Basic Idea

The occurrence of tensor products in (8) shows that the algorithm has inherent block and vector parallelism as explained in (6) and (7). However, depending on the platform and for efficient mapping, the algorithm should exhibit one or both forms of parallelism throughout the computation to the extent possible. To achieve this, (8) can be formally manipulated using well-known matrix identities shown in Table 1.

The table makes clear that there is a virtually unlimited set of possible variants of (8), which also explains the large set of publications on FFTs. These variants hardly differ in operations count but in structure, which is crucial for parallelization. The remainder of this entry introduces the most important parallel FFTs derived in the literature. All these FFTs can be derived from (8) using Table 1. The presentation is divided into iterative and recursive FFTs. Each FFT is visualized for size $n = 16$ in a form similar to (1) (and again from right to left) to emphasize block and vector parallelism. In these visualizations, the twiddle factors are dropped since they do not affect the dataflow and hence pose no structural problem for parallelization.

FFT (Fast Fourier Transform). Table 1 Formula identities to manipulate FFTs. A is $n \times n$, and B and C are $m \times m$. A^T is the transpose of A

$(BC)^T$	$= C^T B^T$
$(A \otimes B)^T$	$= A^T \otimes B^T$
I_{mn}	$= I_m \otimes I_n$
$A \otimes B$	$= (A \otimes I_m)(I_n \otimes B)$
$I_n \otimes (BC)$	$= (I_n \otimes B)(I_n \otimes C)$
$(BC) \otimes I_n$	$= (B \otimes I_n)(C \otimes I_n)$
$A \otimes B$	$= L_n^{mn}(B \otimes A)L_m^{mn}$
$(L_m^{mn})^{-1}$	$= L_n^{mn}$
L_n^{kmn}	$= (L_n^{kn} \otimes I_m)(I_k \otimes L_n^{mn})$
L_{km}^{kmn}	$= (I_k \otimes L_m^{mn})(L_k^{kn} \otimes I_m)$
L_k^{kmn}	$= L_{km}^{kmn} L_{kn}^{kmn}$

Iterative FFTs

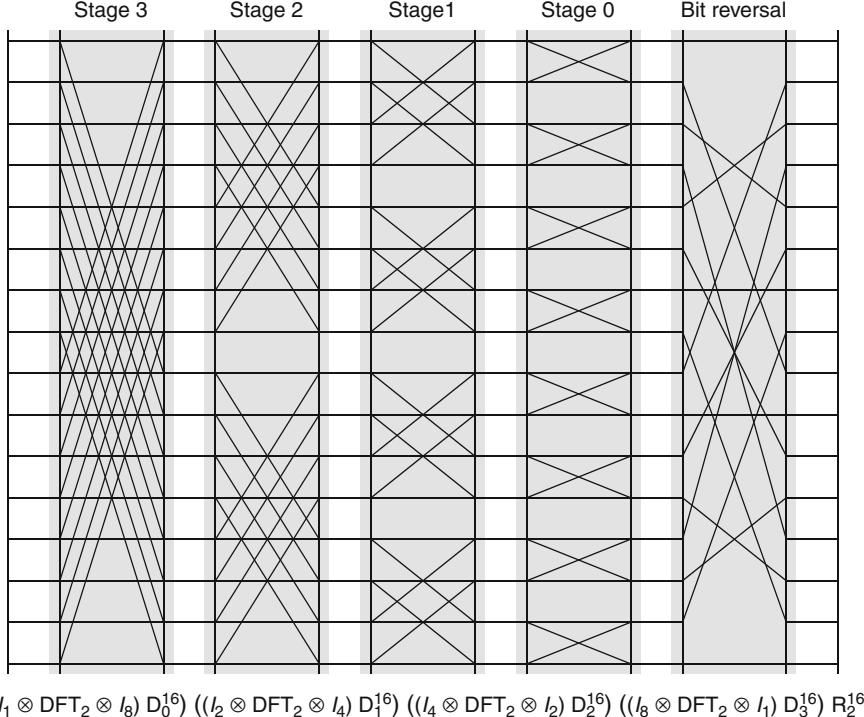
The historically first FFTs that were developed and adapted to parallel platforms are iterative FFTs. These algorithms implement the DFT as a sequence of nested loops (usually three). The simplest are *radix-r* forms (usually $r = 2, 4, 8$), which require an FFT size of $n = r^\ell$; more complicated mixed-radix radix variants always exist. They all factor DFT_n into a product of ℓ matrices, each of which consists of a tensor product and twiddle factors. Iterative algorithms are obtained from (8) by recursive expansion, flattening the nested parentheses, and other identities in Table 1.

The most important iterative FFTs are discussed next, starting with the standard version, which is not optimized for parallelism but included for completeness. Note that the exact form of the twiddle factors differs in these FFTs, even though they are denoted with the same symbol.

Cooley–Tukey iterative FFT. The radix- r *iterative decimation-in-time FFT*

$$DFT_{r^\ell} = \left(\prod_{i=0}^{\ell-1} (I_{r^i} \otimes DFT_r \otimes I_{r^{\ell-i-1}}) D_i^{r^\ell} \right) R_r^{r^\ell}, \quad (9)$$

is the prototypical FFT algorithm and shown in Fig. 3. $R_r^{r^\ell}$ is the radix- r digit reversal permutation and the diagonal $D_i^{r^\ell}$ contains the twiddle factors in the i th stage. The radix-2 version is implemented by Numerical Recipes using a triple loop corresponding to the two tensor products (inner two loops) and the product (outer loop).



FFT (Fast Fourier Transform). Fig. 3 Iterative FFT (9) for $n = 2^4$ and $r = 2$

Formal transposition of (9) yields the *iterative decimation-in-frequency FFT*:

$$\text{DFT}_{r^\ell} = R_r^{r^\ell} \prod_{i=0}^{\ell-1} D_i^{r^\ell} (I_{r^{\ell-i-1}} \otimes \text{DFT}_r \otimes I_{r^i}). \quad (10)$$

Both (9) and (10) contain the bit reversal permutation $R_r^{r^\ell}$. The parallel and vector structure of the occurring butterflies depends on the stage. Thus, even though every stage is data parallel, the algorithm is neither well suited for machines that require block parallelism nor vector parallelism. For this reason, very few parallel triple-loop implementations exist and compiler parallelization and vectorization tend not to succeed in producing any speedup when targeting the triple-loop algorithm.

Pease FFT. A variant of (9) is the *Pease FFT*

$$\text{DFT}_{r^\ell} = \left(\prod_{i=0}^{\ell-1} L_r^{r^\ell} (I_{r^{\ell-i-1}} \otimes \text{DFT}_r) D_i^{r^\ell} \right) R_r^{r^\ell}, \quad (11)$$

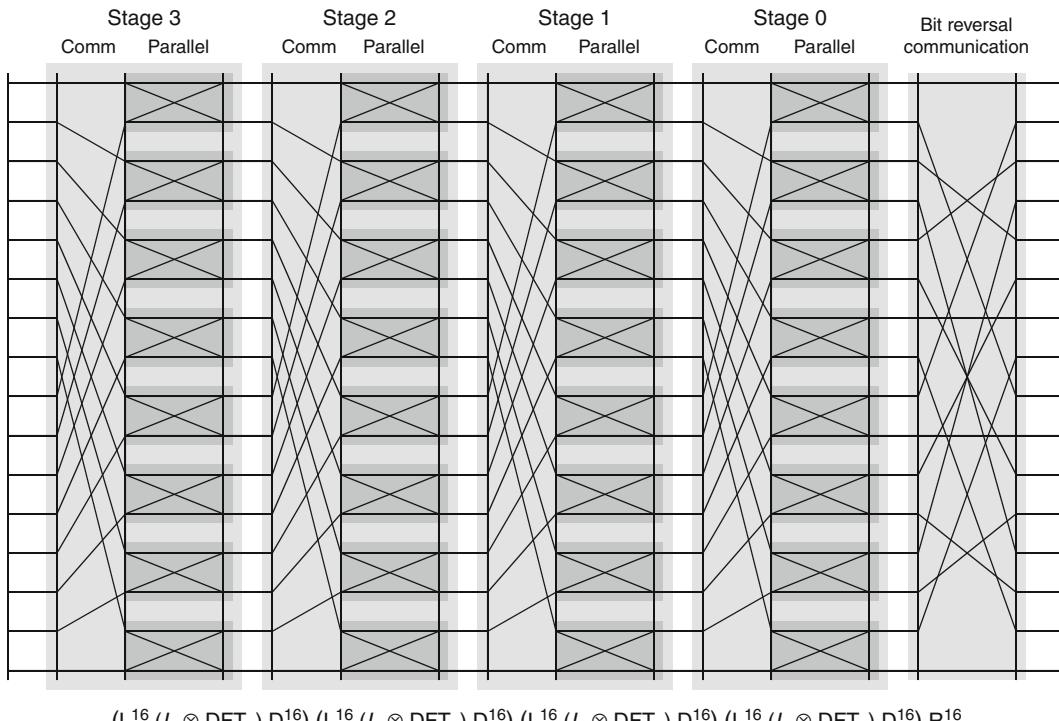
shown in Fig. 4 for $r = 2$. It has constant geometry, that is, the control flow is the same in each stage, and maximizes block parallelism by reducing the block sizes to r on which single butterflies are computed. However,

the Pease FFT also requires the digit reversal permutation. Each stage of the Pease algorithm consists of the twiddle diagonal and a parallel butterfly block, followed by the same data exchange across parallel blocks specified through a stride permutation. The Pease FFT was originally developed for parallel computers, and its regular structure makes it a good choice for field-programmable gate arrays (FPGAs) or ASICs. Formal transposition of (11) yields a variant with the bit-reversal in the end.

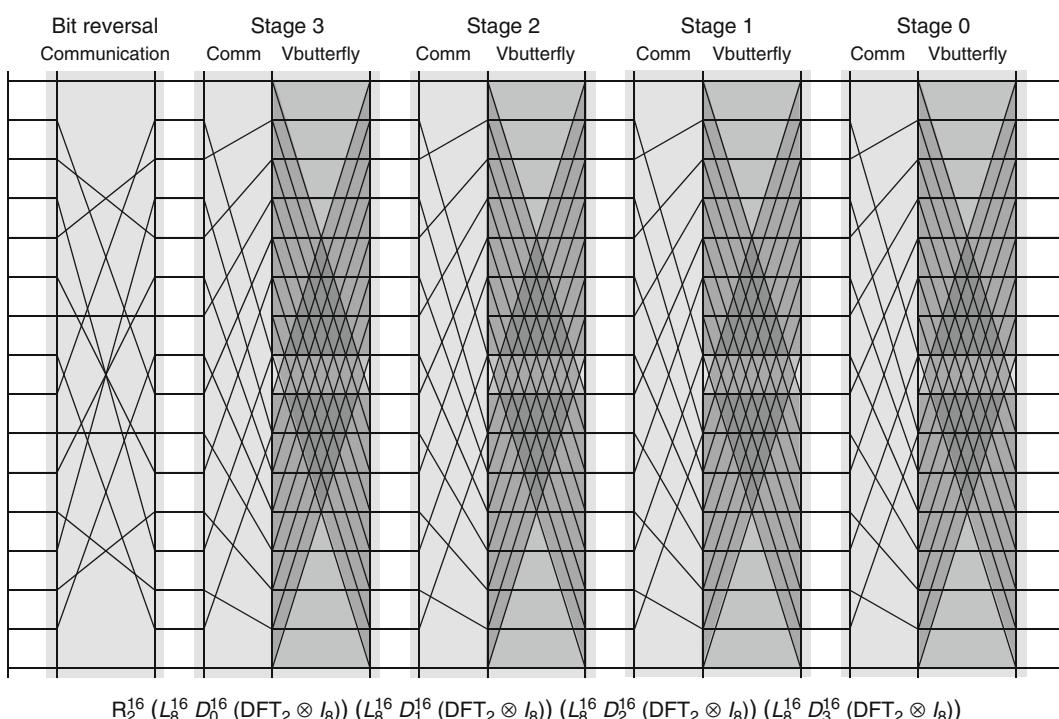
Korn–Lambiotte FFT. The *Korn–Lambiotte FFT* is given by

$$\text{DFT}_{r^\ell} = R_r^{r^\ell} \left(\prod_{i=0}^{\ell-1} L_{r^{\ell-i-1}}^{r^\ell} D_i^{r^\ell} (\text{DFT}_r \otimes I_{r^{\ell-i-1}}) \right), \quad (12)$$

and is the algorithm that is dual to the Pease FFT in the sense used in Fig. 1. Namely, it has also constant geometry, but maximizes vector parallelism as shown in Fig. 5 for $r = 2$. Each stage contains one vector butterfly operating on vectors of length n/r , and a twiddle diagonal. As last step it performs the digit reversal permutation. The Korn–Lambiotte FFT was developed for early vector computers. It is derived from the Pease



FFT (Fast Fourier Transform). Fig. 4 Pease FFT in (11) for $n = 2^4$ and $r = 2$



FFT (Fast Fourier Transform). Fig. 5 Korn-Lambiotte FFT in (12) for $n = 2^4$ and $r = 2$. Vbutterfly = vector butterfly

algorithm through formal transposition followed by the translation of the tensor product from a parallel into a vector form.

Stockham FFT. The Stockham FFT

$$\text{DFT}_{r^\ell} = \prod_{i=0}^{\ell-1} (\text{DFT}_r \otimes I_{r^{\ell-i}}) D_i^{r^\ell} (L_r^{r^{\ell-i}} \otimes I_{r^i}), \quad (13)$$

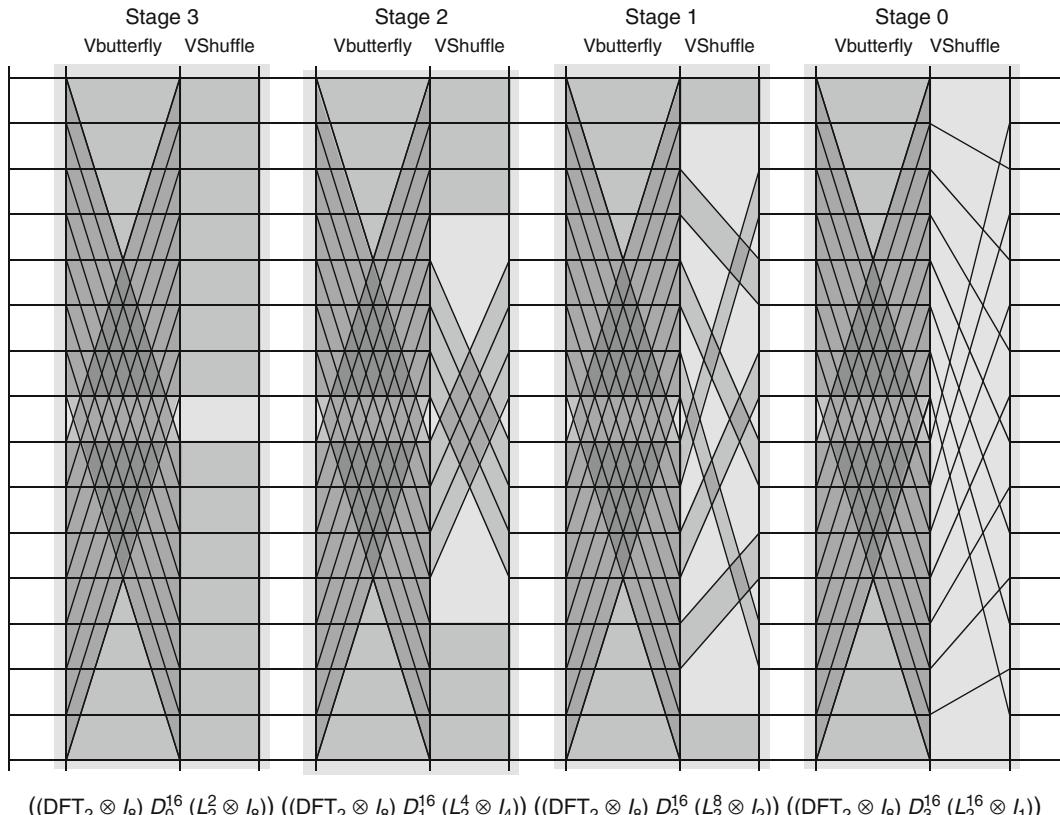
is *self-sorting*, that is, it does not have a digit reversal permutation. It is shown in Fig. 6 for $r = 2$. Like the Korn–Lambiotte FFT, it exhibits maximal vector parallelism but the permutations change across stages. Each of these permutations is a vector permutation, but the vector length increases by a factor of r in each stage (starting with 1). Thus, for most stages a sufficiently long vector length is achieved. The Stockham FFT was originally developed for vector computers. Its structure is also suitable for graphics processors (GPUs), and indeed most current GPU FFT libraries are based on the

Stockham FFT. The formal transposition of (13) is also called Stockham FFT.

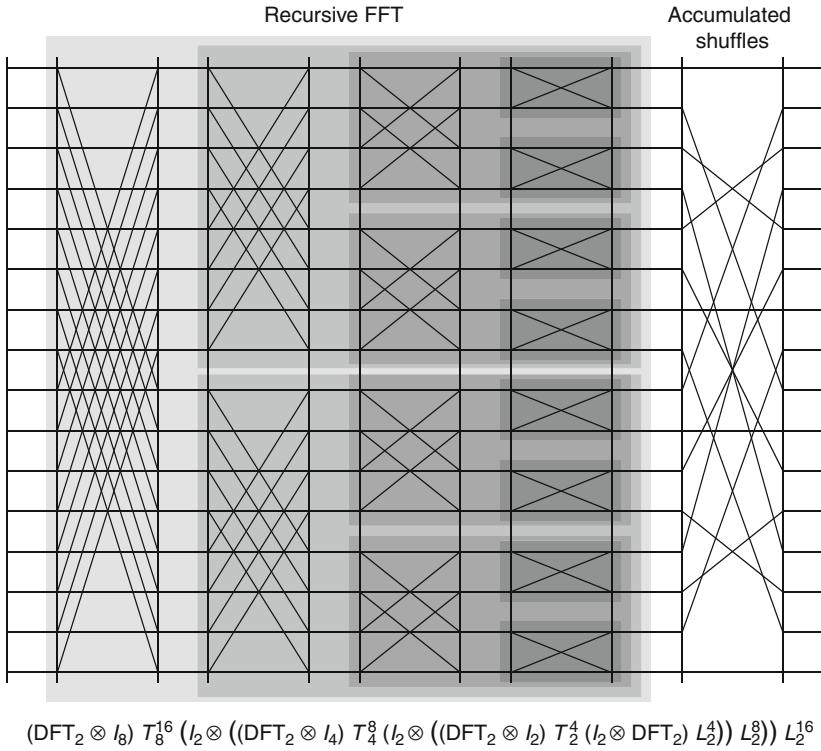
Recursive FFT Algorithms

The second class of Cooley–Tukey-based FFTs are recursive algorithms, which reduce a DFT of size $n = km$ into k DFTs of size m and m DFTs of size k . The advantage of recursive FFTs is better locality and hence better performance on computers with deep memory hierarchies. They also can be used as kernels for iterative algorithms. For parallelism, recursive algorithms are derived, for example, to maximize the block size for multicore platforms, or to obtain vector parallelism for a fixed vector length for platforms with SIMD vector extensions. The most important recursive algorithms are discussed next.

Recursive Cooley–Tukey FFT. The recursive, general-radix decimation-in-time Cooley–Tukey FFT was shown before in (8). Typically, k is chosen to be



FFT (Fast Fourier Transform). Fig. 6 Stockham FFT in (13) for $n = 2^4$ and $r = 2$. Vbutterfly = vector butterfly, VShuffle = vector shuffle



FFT (Fast Fourier Transform). Fig. 7 Recursive radix-2 decimation-in-time FFT for $n = 2^4$

small, with values up to 64. If (8) is applied to $n = r^\ell$ recursively with $k = r$ the algorithm is called radix- r decimation-in time FFT. As explained before, the initial permutation is usually not performed but propagated as data access into the smaller DFTs. For radix-2 the algorithm is shown in Fig. 7. Note that the dataflow is equal to Fig. 3, but the order of computation is different as emphasized by the shading.

Formal transposition of (8) yields the *recursive decimation-in-frequency FFT*

$$DFT_n = L_m^n (I_k \otimes DFT_m) T_m^n (DFT_k \otimes I_m), \quad n = km. \quad (14)$$

Recursive application of (8) and (14) eventually leads to prime sizes k and m , which are handled by a special prime-size FFT. For two-powers n the butterfly matrix DFT_2 terminates the recursion.

The implementation of (8) and (14) is more involved than the implementation of iterative algorithms, in particular in the mixed-radix case. The divide-and-conquer nature of (8) and (14) makes them good choices for machines with memory hierarchies, as at some

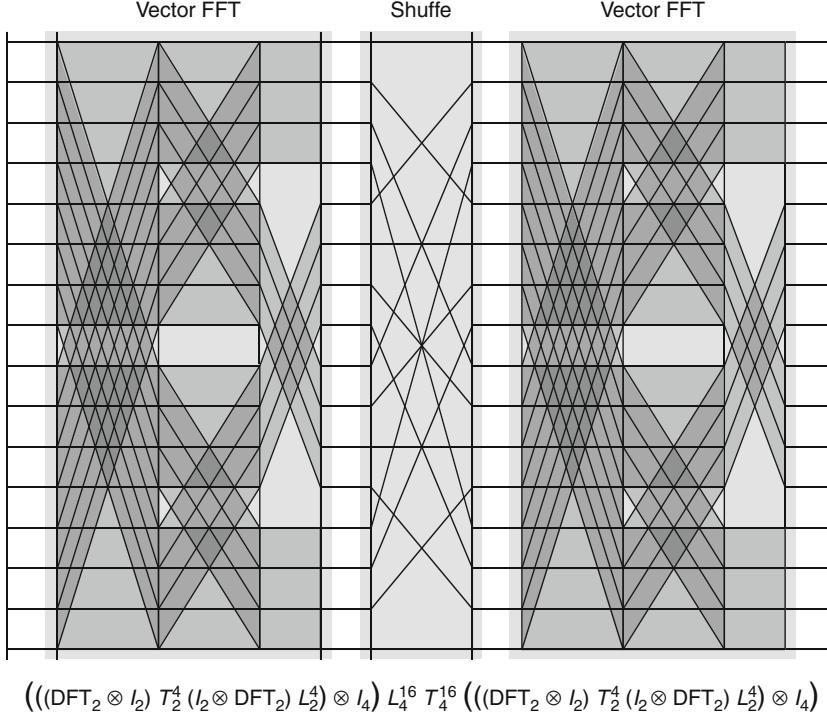
recursion level the working set will be small enough to fit into a certain cache level, a property sometimes called *cache oblivious*. Both (8) and (14) contain both vector and parallel blocks and stride permutations. Thus, despite their inherent data parallelism, they are not ideal for either parallel or vector implementations. The following variants address this problem.

Four-step FFT. The *four-step FFT* is given by

$$DFT_n = (DFT_k \otimes I_m) T_m^n L_k^n (DFT_m \otimes I_k), \quad n = km, \quad (15)$$

and shown in Fig. 8. It is built from two stages of vector FFTs, the twiddle diagonal and a transposition. Typically, $k, m \approx \sqrt{n}$ is chosen (also called “square root decomposition”). Then, (15) results in the longest possible vector operations except for the stride permutation in the middle.

The four-step FFT was originally developed for vector computers and the stride permutation (or transposition) was originally implemented explicitly while the smaller FFTs were expanded with some other FFT—typically iterative. The transposition can



FFT (Fast Fourier Transform). Fig. 8 Four-step FFT for $n = 2^4$ and $k = m = \sqrt{n} = 4$

be implemented efficiently using blocking techniques. Equation (15) can be a good choice on parallel machines that execute operations on long vectors well and on which the overhead of a transposition is not too high. Examples includes vector computers and machines with streaming memory access like GPUs.

Six-step FFT. The *six-step FFT* is given by

$$\text{DFT}_n = L_k^n (I_m \otimes \text{DFT}_k) L_m^n T_m^n (I_k \otimes \text{DFT}_m) L_k^n, \quad n = km, \quad (16)$$

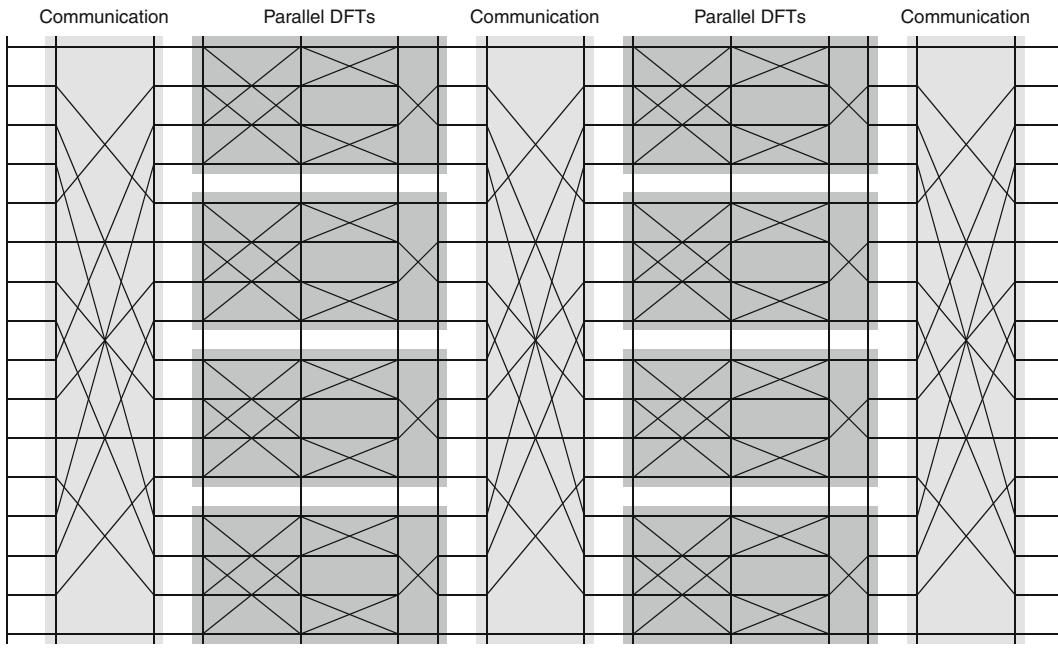
and shown in Fig. 9. It is built from two stages of parallel butterfly blocks, the twiddle diagonal, and three global transpositions (all-to-all data exchanges). (16) was originally developed for distributed memory machines and out-of-core computation. Typically, $k, m \approx \sqrt{n}$ is chosen to maximize parallelism. The transposition was originally implemented explicitly as all-to-all communication while the smaller FFTs were expanded with some other FFT algorithm—typically iterative. As in (15), the required matrix transposition can be blocked for more efficient data movement. (16) can be a good choice on parallel machines that have

multiple memory spaces and require explicit data movement, like message passing, off-loading to accelerators (GPUs and FPGAs), and out-of-core computation.

Multicore FFT. The *multicore FFT* for a platform with p cores and cache block size μ is given by

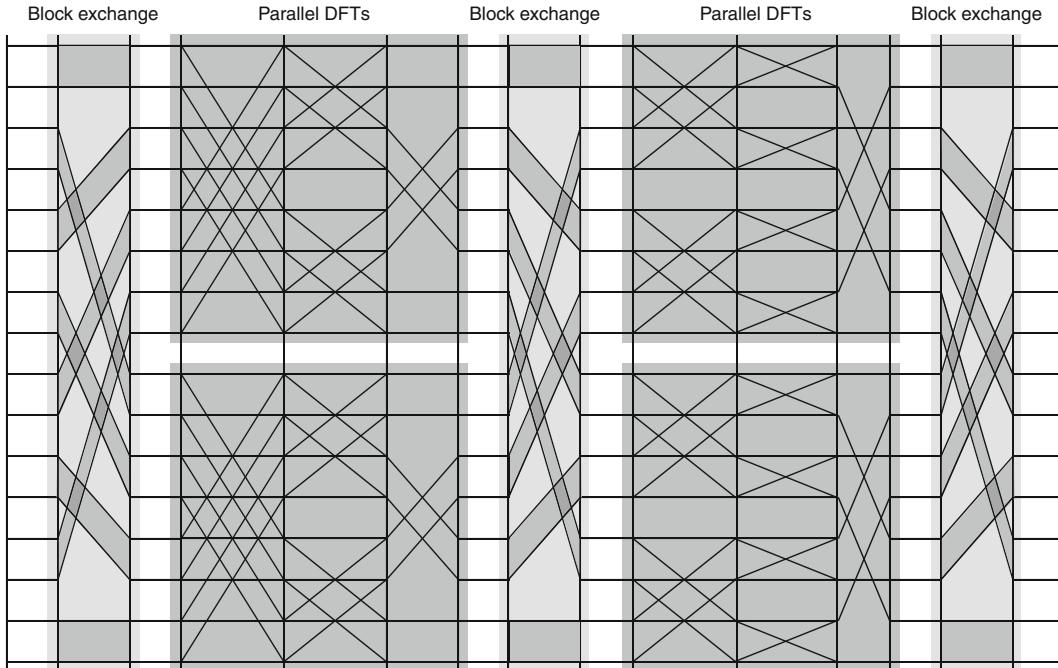
$$\begin{aligned} \text{DFT}_n = & \left(I_p \otimes (\text{DFT}_k \otimes I_{m/p}) \right)^{\left((L_p^{kp} \otimes I_{m/p\mu}) \otimes I_\mu \right)} T_m^n \\ & \times \left(I_p \otimes (I_{k/p} \otimes \text{DFT}_m) L_{k/p}^{n/p} \right) \\ & \times \left((L_p^{pm} \otimes I_{k/p\mu}) \otimes I_\mu \right), \quad n = km, \end{aligned} \quad (17)$$

and is a version of (8) that is optimized for homogeneous multicore CPUs with memory hierarchies. An example is shown in Fig. 10. (17) follows the recursive FFT (8) closely but ensures that all data exchanges between cores and all memory accesses are performed with cache block granularity. For a multicore with cache block size μ and p cores, (17) is built solely from permutations that permute entire cache lines and p -way parallel compute blocks. This property allows for parallelization of small problem sizes across a moderate



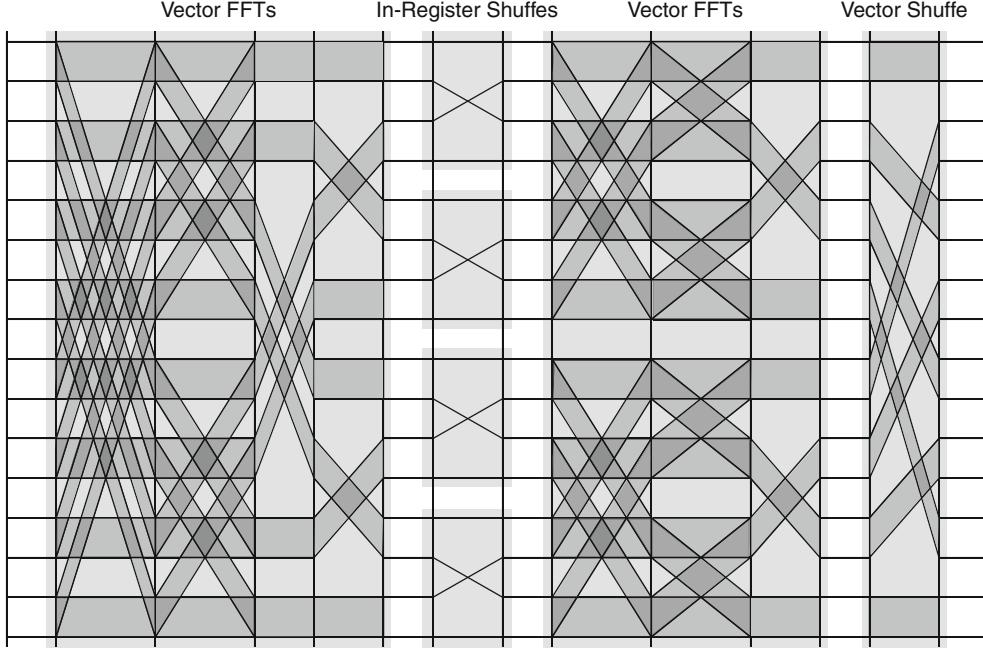
$$L_4^{16} \left(I_4 \otimes ((DFT_2 \otimes I_2) T_2^4 (I_2 \otimes DFT_2) L_2^4) \right) L_4^{16} T_4^{16} \left(I_4 \otimes ((DFT_2 \otimes I_2) T_2^4 (I_2 \otimes DFT_2) L_2^4) \right) L_4^{16}$$

FFT (Fast Fourier Transform). Fig. 9 Six-step FFT for $n = 2^4$ and $k = m = \sqrt{n} = 4$



$$(L_4^8 \otimes I_2) (I_2 \otimes ((DFT_2 \otimes I_2) T_2^4 (I_2 \otimes DFT_2) L_2^4) \otimes I_2) (L_2^8 \otimes I_2) T_4^{16} (I_2 \otimes (I_2 \otimes (DFT_2 \otimes I_2) T_2^4 (I_2 \otimes DFT_2)) R_2^8) (L_2^8 \otimes I_2)$$

FFT (Fast Fourier Transform). Fig. 10 Multicore FFT for $n = 2^4$, $k = m = 4$, $p = 2$ cores, and cache block size $\mu = 2$



$$\left(\left((\text{DFT}_2 \otimes I_2) T_2^4 (I_2 \otimes \text{DFT}_2) L_2^4 \right) \otimes I_2 \right) T_4^{16} \left(I_2 \otimes (L_2^4 \otimes I_2) (I_2 \otimes L_2^4) \left(\left((\text{DFT}_2 \otimes I_2) T_2^4 (I_2 \otimes \text{DFT}_2) L_2^4 \right) \otimes I_2 \right) (L_2^8 \otimes I_2) \right)$$

FFT (Fast Fourier Transform). Fig. 11 Short vector FFT in (18) for $n = 2^4$, $k = m = 4$, and (complex) vector length $v = 2$

number of cores. Implementation of (17) on a cache-based system relies on the cache coherency protocol to transmit cache lines of length μ between cores and requires a global barrier. Implementation on a scratchpad-based system requires explicit sending and receiving of the data packets, and depending on the communication interface additional synchronization may be required.

The smaller DFTs in (17) can be expanded, for example, with the short vector FFT (discussed next) to optimize for vector extensions.

SIMD short vector FFT. For CPUs with SIMD v -way vector extensions like SSE and AltiVec and a memory hierarchy, the *short vector FFT* is defined as

$$\begin{aligned} \text{DFT}_n = & \left((\text{DFT}_k \otimes I_{m/v}) \otimes I_v \right) T_m^n \left(I_{k/v} \otimes (L_v^m \otimes I_v) \right. \\ & \times \left. (I_{m/v} \otimes L_v^{v^2}) (\text{DFT}_m \otimes I_v) \right) \left(L_{k/v}^{n/v} \otimes I_v \right), \quad n = km, \end{aligned} \quad (18)$$

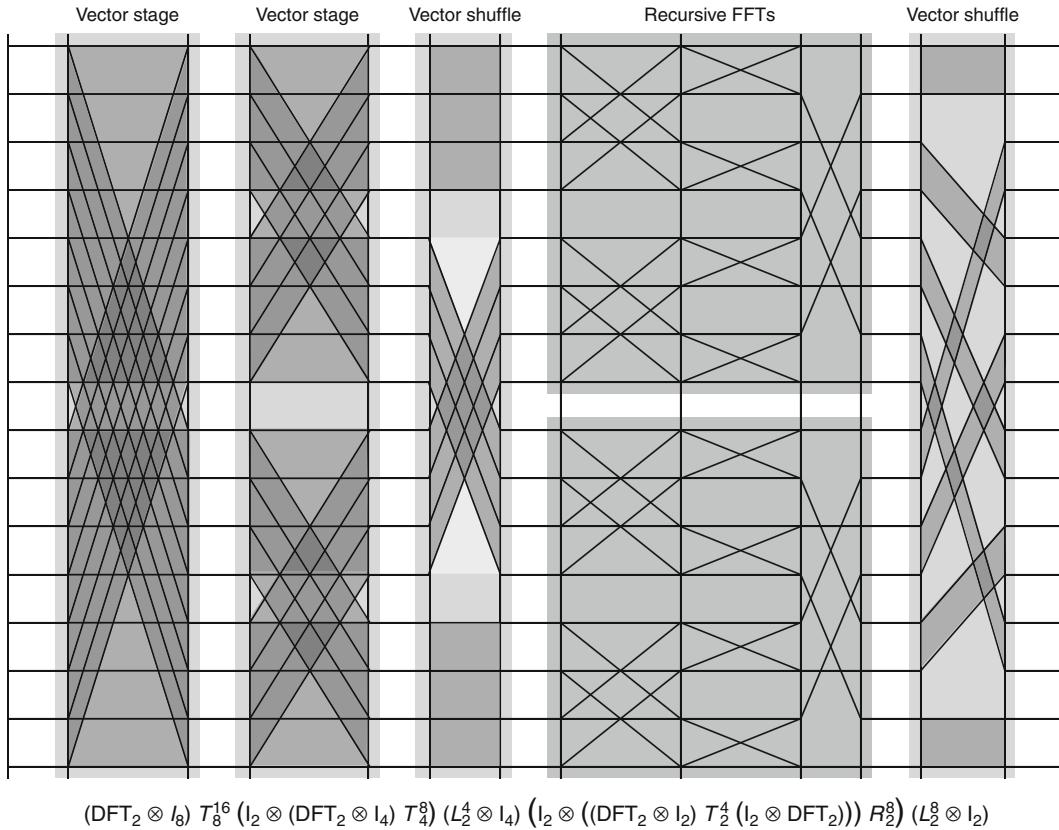
and can be implemented using solely vector arithmetic, aligned vector memory access, and a small number of vector shuffle operations. An example is shown in Fig. 11. All compute operations in (18) have complex v -way vector parallelism. The only operation that is not

v -way vectorized is the stride permutations $L_v^{v^2}$, which can be implemented efficiently using in-register shuffle instructions. (18) requires the support or implementation of complex vector arithmetic and packs v complex elements into a machine vector register of width $2v$. A variant that vectorizes the real rather than the complex dataflow exists.

Vector recursion. The *vector recursion* performs a locality optimization for deep memory hierarchies for the first stage $(I_k \otimes \text{DFT}_m)L_k^n$ of (8). Namely, in this stage DFT_m is further expanded using again (8) with $m = m_1 m_2$ and the resulting expression is manipulated to yield

$$\begin{aligned} (I_k \otimes \text{DFT}_m)L_k^n = & \left(I_k \otimes (\text{DFT}_{m_1} \otimes I_{m_2}) T_{m_2}^m \right) \\ & \times \left(L_k^{km_1} \otimes I_{m_2} \right) \left(I_{m_1} \otimes (I_k \otimes \text{DFT}_{m_2}) L_k^{km_2} \right) \\ & \times \left(L_{m_1}^m \otimes I_k \right). \end{aligned} \quad (19)$$

While the recursive FFT (8) ensures that the working set will eventually fit into any level of cache, large two-power FFTs induce large 2-power strides. For caches with lower associativity these strides result in a high number of conflict misses, which may impose



FFT (Fast Fourier Transform). Fig. 12 Vector recursive FFT for $n = 2^4$. The vector recursion is applied once and yields vector shuffles, two recursive FFTs, and two iterative vector stages

a severe performance penalty. For large enough two-power sizes, in the first stage of (8) every single load will result in a cache miss. The vector recursion alleviates this problem by replacing the stride permutation in (8) by stride permutations of vectors, at the expense of an extra pass through the working set. Since (19) matches $(I_k \otimes DFT_n) L_k^{kn}$, it is recursively applicable and will eventually produce child problems that fit into any cache level. The vector recursion produces algorithms that are a mix of iterative and recursive as shown in Fig. 12.

Other FFT Topics

So far the discussion has focused on one-dimensional complex two-power size FFTs. Some extensions are mentioned next.

General size recursive FFT algorithms. DFT algorithms fundamentally different from (8) include prime-factor (n is a product of coprime factors), Rader (n is

prime), and Bluestein or Winograd (any n). In practice, these are mostly used for small sizes < 32 , which then serve as building blocks for large composite sizes via (8). The exception is Bluestein's algorithm that is often used to compute large sizes with large prime factors or large prime numbers.

DFT variants and other FFTs. In practice, several variants of the DFT in (2) are needed including forward/inverse, interleaved/split complex format, for complex/real input data, in-place/out-of-place, and others. Fortunately, most of these variants are close to the standard DFT in (2), so fast code for the latter can be adapted. An exception is the DFT for real input data, which has its own class of FFTs.

Multidimensional FFT algorithms. The Kronecker product naturally arises in 2D and 3D DFTs, which respectively can be written as

$$DFT_{m \times n} = DFT_m \otimes DFT_n, \quad (20)$$

$$DFT_{k \times m \times n} = DFT_k \otimes DFT_m \otimes DFT_n. \quad (21)$$

For a 2D DFT, applying identities from Table 1 to (20) yields the row-column algorithm

$$\text{DFT}_{m \times n} = (\text{DFT}_m \otimes I_n)(I_m \otimes \text{DFT}_n). \quad (22)$$

The 2D vector-radix algorithm can also be derived with identities from Table 1 from (20):

$$\begin{aligned} \text{DFT}_{mn \times rs} &= (\text{DFT}_{m \times r} \otimes I_{ns})^{I_m \otimes L_r^m \otimes I_s} (T_n^{mn} \otimes T_s^{rs}) \\ &\times (I_{mr} \otimes \text{DFT}_{n \times s})^{I_m \otimes L_r^m \otimes I_s} (L_m^{mn} \otimes L_r^{rs}). \end{aligned} \quad (23)$$

Higher-dimensional versions are derived similarly, and the associativity of \otimes gives rise to more variants.

Related Entries

- [ATLAS \(Automatically Tuned Linear Algebra Software\)](#)
- [FFTW](#)
- [Spiral](#)

Bibliographic Notes and Further Reading

The original Cooley–Tukey FFT algorithm can be found in [2]. The Pease FFT in [13] is the first FFT derived and represented using the Kronecker product formalism. The other parallel FFTs were derived in [10] (Korn–Lambiotte FFT), [16] (Stockham FFT), [11] (four-step FFT), [1] (six-step FFT). The vector-radix FFT algorithm can be found in [8], the vector recursion in [7], the short vector FFT in [3], and the multicore FFT in [4]. A good overview on FFTs including the classical parallel variants is given in Van Loan’s book [18] and the book by Tolimieri, An and Lu [17]; both are based on the formalism used here. Also excellent is Nussbaumer FFT book [12]. An overview on real FFTs can be found in [20].

At the point of writing the most important fast DFT libraries are FFTW by Frigo and Johnson [6, 7], Intel’s MKL and IPP, and IBM’s ESSL and PESSL. FFTE is currently used in the HPC Challenge as *Global FFT* benchmark reference implementation. Most CPU, GPU, and FPGA vendors maintain DFT libraries. Some historic DFT libraries like FFTPACK are still widely used. Numerical Recipes [14] provides C code for an iterative radix-2 FFT implementation. BenchFFT provides up-to-date FFT benchmarks of about 60 single-node DFT libraries. The Spiral system is capable of generating parallel DFT libraries directly from the tensor product-based algorithm description [15, 19].

Bibliography

1. Bailey DH (1990) FFTs in external or hierarchical memory. *J. Supercomput* 4:23–35
2. Cooley JW, Tukey JW (1965) An algorithm for the machine calculation of complex Fourier series. *Math Comput* 19:297–301
3. Franchetti F, Püschel M (2003) Short vector code generation for the discrete Fourier transform. In: *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, pp 58–67, Washington, DC, USA
4. Franchetti F, Voronenko Y, Püschel M (2006) FFT program generation for shared memory: SMP and multicore. In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, New York, NY, USA
5. Franchetti F, Püschel M, Voronenko Y, Chellappa S, Moura JMF (2009) Discrete Fourier transform on multicore. *IEEE Signal Proc Mag*, special issue on “Signal Processing on Platforms with Multiple Cores” 26(6):90–102
6. Frigo M, Johnson SG (1998) FFTW: An adaptive software architecture for the FFT. In: *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol 3, pp 1381–1384, Seattle, WA
7. Frigo M, Johnson SG (2005) The design and implementation of FFTW3. *Proc IEEE* special issue on “Program Generation, Optimization, and Adaptation” 93(2):216–231
8. Harris DB, McClellan JH, Chan DSK, Schuessler HW (1977) Vector radix fast Fourier transform. In: *Proc Inter Conf Acoustics, Speech, and Signal Processing. Conference Proceedings (ICASSP ’77)*, pp 548–551, Los Alamitos, IEEE Comput. Soc. Press
9. Heidemann MT, Johnson DH, Burrus CS (1985) Gauss and the history of the fast fourier transform. *Arch Hist Exact Sci* 34:265–277
10. Korn DG, Lambiotte JJ, Jr. (1979) Computing the fast Fourier transform on a vector computer. *Math Comput* 33(7): 977–992
11. Norton A, Silberger AJ (1987) Parallelization and performance analysis of the Cooley-Tukey FFT algorithm for shared-memory architectures. *IEEE Trans Comput* 36(5):581–591
12. Nussbaumer HJ (1982) Fast Fourier transformation and convolution algorithms, 2nd ed. Springer, New York
13. Pease MC (1968) An adaptation of the fast Fourier transform for parallel processing. *J ACM* 15(2):252–264
14. Press WH, Flannery BP, Teukolsky SA, Vetterling WT (1992) Numerical recipes in C: the art of scientific computing, 2nd ed. Cambridge University Press, Cambridge
15. Püschel M, Moura JMF, Johnson J, Padua D, Veloso M, Singer BW, Xiong J, Franchetti F, Gačić A, Voronenko Y, Chen K, Johnson RW, Rizzolo N (1987) SPIRAL: Code generation for DSP transforms. *Proc IEEE* special issue on “Program Generation, Optimization, and Adaptation” 93(2):232–275
16. Schwarztrauber PN (1987) Multiprocessor FFTs. *Parallel Comput* 5:197–210
17. Tolimieri R, An M, Lu C (1997) Algorithms for discrete Fourier transforms and convolution, 2nd ed. Springer, New York
18. Van Loan C (1992) Computational framework of the fast Fourier transform. SIAM, Philadelphia, PA, USA

19. Voronenko Y, de Mesmay F, Püschel M (2009) Computer generation of general size linear transform libraries. In: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization, pp 102–113, Washington, DC, USA
20. Voronenko Y, Püschel M (2009) Algebraic signal processing theory: Cooley-Tukey type algorithms for real DFTs. *IEEE Trans Signal Proc* 57(1):205–222

Fast Algorithm for the Discrete Fourier Transform (DFT)

- [FFT \(Fast Fourier Transform\)](#)

FFTW

FFTW is a C library for computing the Discrete Fourier Transform (DFT). It was originally developed at MIT by Matteo Frigo and Steven G. Johnson. FFTW is an example of autotuning in two ways. First, the library is adaptive by allowing for different recursion strategies. The best one is chosen at the time of use by a feedback-driven search. Second, the recursion is terminated by small optimized kernels (called codelets) that are generated by a special purpose compiler. The library continues to be maintained by the developers and is widely used due to its excellent performance. The name FFTW stands for the somewhat whimsical “Fastest Fourier Transform in the West.”

Related Entries

- [Autotuning](#)
- [FFT \(Fast Fourier Transform\)](#)
- [Spiral](#)

Bibliography

1. Frigo M, Johnson SG (1998) FFTW: An adaptive software architecture for the FFT. In: Proceedings IEEE international conference acoustics, speech, and signal processing (ICASSP), 3, pp 1381–1384
2. Frigo M (1999) A fast Fourier transform compiler. In: Proceedings of the ACM SIGPLAN 1999 conference on programming language design and implementation (PLDI '99). ACM, New York, pp 169–180
3. Frigo M, Johnson SG (2005) The design and implementation of FFTW3. *Proceedings of the IEEE* 93(2):216–231

File Systems

ROBERT B. ROSS

Argonne National Laboratory, Argonne, IL, USA

Synonyms

[Cluster file systems](#)

Definition

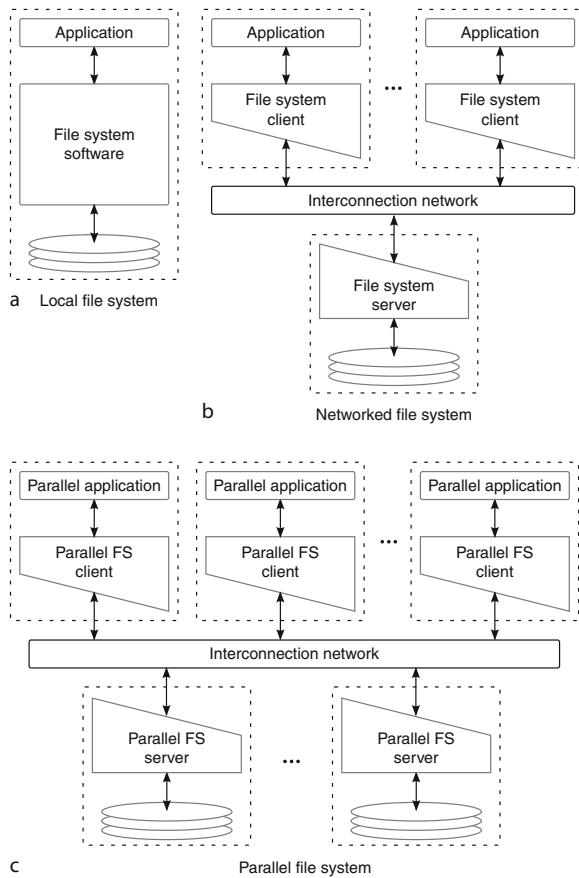
A parallel file system (PFS) is a system software component that organizes many disks, servers, and network links to provide a file system name space that is accessible from many clients; distributes data across many devices to enable high aggregate bandwidth; and coordinates changes to file system data so that clients' views of the data are kept coherent.

Discussion

Introduction

File systems are the traditional mechanism for reading and writing data on persistent storage. The current standard interface for file systems, part of the Portable Operating System Interface (POSIX) standard, was first defined in 1988 [14]. As networking of computers became common, sharing storage between multiple computers became desirable. In 1989 [15], the Network File System (NFS) standard was created, allowing a file system stored locally on one computer to be accessed by other computers. With the advent of NFS, *shared file name spaces* were possible – many computers could see and access the same files at the same time.

Figures 1a and b show a high-level view of a local and network file system, respectively. In the local file system case, a single software component manages local storage resources (typically one or more disks) and allows processes running on that computer to store and retrieve data organized as files and directories. In the network file system case, the software allowing access to storage is split from the software managing the storage. The file system client presents the traditional view of files and directories to processes on a computer, while the file system server software manages the storage and coordinates access from multiple clients.



File Systems. Fig. 1 Local, network, and parallel file systems

As distributed-memory parallel computers became popular, the utility of a shared file name space became obvious as a way for parallel programs to store and retrieve data without concern for the actual location of files and file data. However, accessing data stored on a single server quickly became a bottleneck, driving the implementation of parallel file systems – file systems that opened up many concurrent paths between computers and persistent storage.

Figure 1c depicts a parallel file system. With multiple network links, servers, and local storage units, high degrees of concurrent access are possible. This enables very high aggregate I/O rates for the parallel file system as a whole when it is accessed by many processes on different nodes in the system, especially when those accesses are relatively evenly distributed across the storage servers. A wide variety of network configurations

and technologies are used in PFS deployments, from storage area networks and InfiniBand to Gigabit Ethernet and TCP/IP.

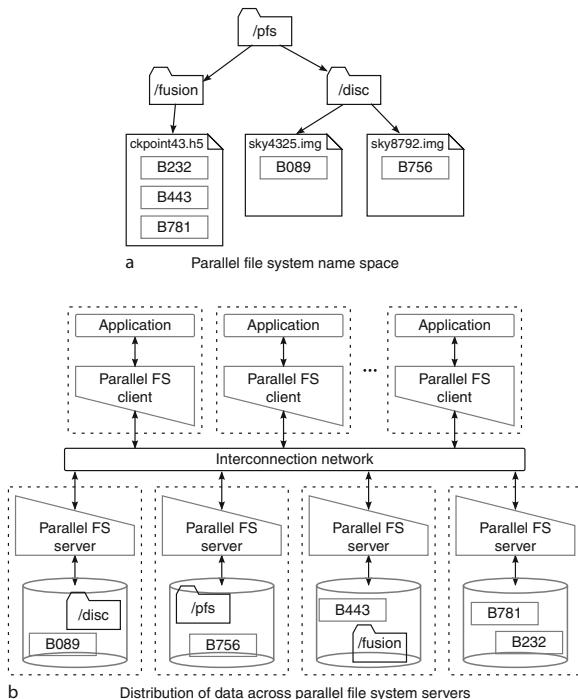
This discussion focuses on parallel file system deployments where processes accessing the file system are on compute nodes distinct from the nodes that manage storage of data (storage nodes). This model is the most common in high-performance computing (HPC) deployments, because (a) there are typically many more compute nodes than needed to provide storage services, (b) placing disks in all compute nodes has an impact on their reliability and power consumption, (c) servicing I/O operations for some other process on a node executing an HPC application can perturb performance of that application, and (d) keeping storage nodes distinct enables higher-reliability components to be used and helps collocate storage resources in the machine room.

That said, Beowulf-style commodity clusters and clusters used for Data Intensive Scalable Computing (DISC) applications are typically configured with local drives, and organizing those drives with a parallel file system can be an effective way to provide a low-cost, high-performance, cluster-wide storage resource.

Data Distribution

A number of approaches are used for managing data distribution in parallel file systems. Figures 2a and b show a simple example file system name space and a distribution of that name space onto multiple storage nodes, respectively. In the name space, two HPC applications have generated three files. A fusion code has written a large checkpoint, while a DISC application is analyzing a pair of relatively small images from observations of the night time sky. The checkpoint has been split into a set of blocks distributed across multiple storage nodes, while the data for each of the smaller images resides on a single server. In addition to distributing the file data across storage nodes, in this example the file system *metadata* has also been distributed, as depicted by directory data residing on different storage nodes. File system metadata is the data that represents the directory tree as well as information such as the owners and permissions of files.

File data is typically divided into fixed-size units called *stripe units* and distributed in a round-robin manner over a fixed set of servers. This approach is similar



File Systems. Fig. 2 Data distribution in a parallel file system (PFS)

to the way in which data is distributed over disks in a disk array. How PFSes keep track of where data is stored will be covered in the section on Metadata Storage and Access.

In a configuration such as the one shown here, the file system deployment is said to be *symmetric*, meaning that metadata and data are both distributed and stored on the same storage nodes. Not all parallel file systems operate this way, and those that keep metadata on separate servers are considered *asymmetric*. There are advantages to both approaches. The symmetric approach provides a high degree of concurrency for metadata operations, which can be important for applications that generate and process many files. On the other hand, storing metadata on a smaller set of nodes leads to greater locality of access for metadata and simplifies fault tolerance for the name space. Both approaches are commonly used, and some parallel file systems, such as GPFS [1] and PVFS [2], can be deployed with either configuration depending on workload expectations.

Access Patterns

The designs of parallel file systems have been heavily influenced by the characteristics and access patterns of HPC applications. These applications execute at very large scale, with current systems allowing over 128,000 application processes to execute at once. Potentially all of those processes might initiate an I/O operation at any time, including all at once. Additionally, HPC applications are primarily scientific simulations [CROSSREF](#). These codes operate on complex data models, such as regular or irregular grids, and these data models are reflected in the data that they store. The end result is that HPC access patterns are often highly concurrent and complex, including noncontiguity of access [3]. This often arises from access to subarrays of large global arrays.

A primary use case for parallel file systems in HPC environments is for *checkpointing*. Checkpointing is the writing of application state to persistent storage for the purpose of enabling an application to restart at some point in the event of a system failure. This state is typically referred to as a “checkpoint.” If a node on which the application is running crashes, then the application can be restarted using the checkpoint to avoid having to recalculate early results. Typically an application will compute for some amount of time, and then all the processes will pause and write their checkpoint so that the checkpoint represents the same point in time across all the processes.

Users are often surprised that their serial workloads are not faster when run on a parallel file system. The cost of coordination in the PFS in conjunction with different caching policies and the latency of communicating with file servers can result in performance no different from, or even worse than, that of local storage. Serial workloads are simply not the primary focus of PFS designs.

Consistency

In the context of PFSes, *consistency* is about the guarantees that the file system makes with respect to concurrent access by processes, particularly processes on different compute nodes. The stronger these guarantees, the more coordination is generally necessary, inside the PFS, to ensure the guarantees.

For example, most parallel file systems ensure that data written by one process is visible to all other processes immediately following the completion of the write operation. If no other copies of file data are kept in the system, then this is easy to enforce: the server owning the data simply always returns data from the most recent completed write.

Some parallel file systems make a stronger guarantee, that of *atomicity* of writes. This means that other processes accessing the PFS will always see either all, or none, of a write operation. This guarantee requires coordination, because if a write spans multiple servers, then the servers need to agree on which data to return in reads if reading is allowed while a write is in progress. Alternatively, the PFS could force readers to wait when a write is ongoing. Many parallel file systems do guarantee atomicity of writes, and this approach of delaying reading when writes are ongoing is commonly used to provide this guarantee. These PFSes include a distributed locking component. Distributed locking systems in parallel file systems are typically single-writer, multi-reader, meaning that they allow many client nodes to simultaneously read a resource but permit only a single client node access when writing is in progress. Scalable parallel file systems perform locking on ranges of bytes in files, usually rounded to some block size. This allows concurrent writes to different ranges from different compute nodes.

The use of distributed locking in parallel file systems has two significant impacts on designs that rely on them. First, the granularity of locking places a limit on concurrency and leads to *false sharing* during writes. False sharing occurs when two processes on different compute nodes attempt to write to different bytes of a file that happen to map to a single lock. While the two processes are not really performing shared access to the same bytes, the locking system behaves as if they were, serializing their writes. Given the nature of HPC access patterns, false sharing is common. Second, availability is impacted. If a compute node fails while holding a lock, access to the locked resource is halted until the lock can be reclaimed. Determining the difference between a busy compute node and a failed one can be difficult at large scale, especially in the presence of lightweight, non-preemptive kernels. For these reasons, some PFS designs eschew the use of distributed

locking, instead slightly relaxing consistency guarantees to a level that may be supported without locking infrastructure [4].

Parallel File System Design

Five major mechanisms shape a PFS design:

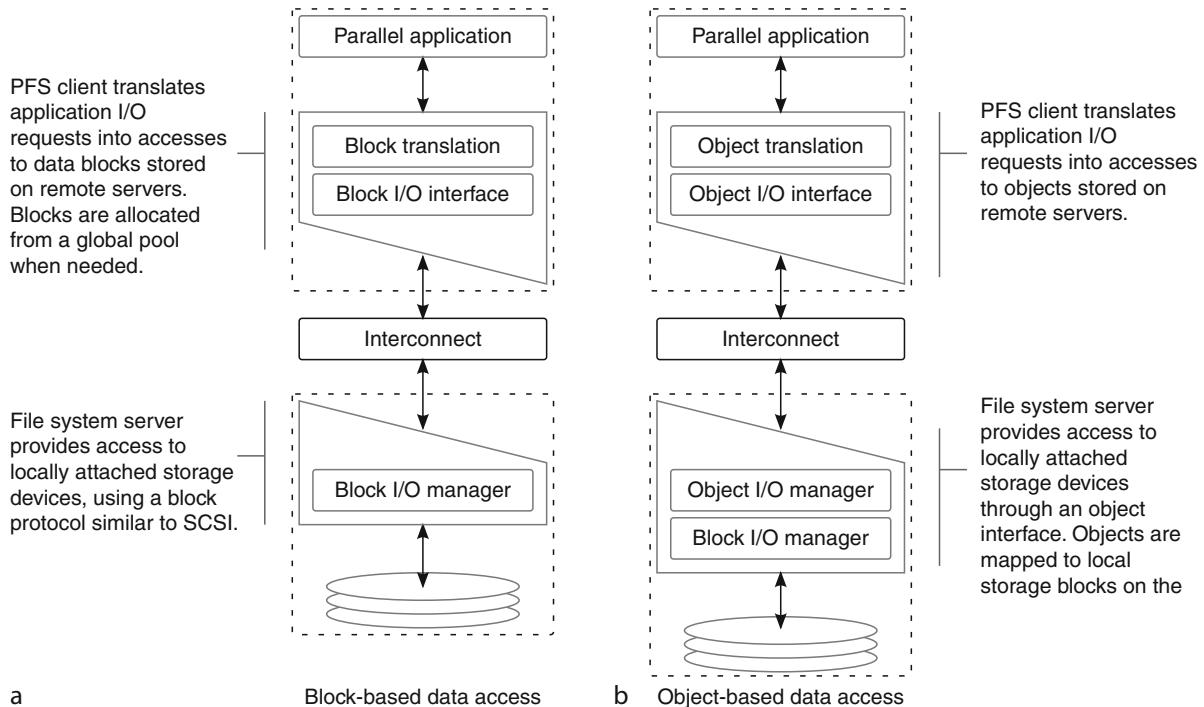
- Underlying storage abstraction
- Metadata storage and access
- Caching
- Tolerance of failures
- Enforcement of consistency semantics

Consistency semantics have been discussed. The other four topics are covered next.

Underlying Storage Abstraction

Some early parallel file systems relied on storage area networks (SANs) to provide shared access to physical drives. In that approach, data was accessed directly by block locations on drives or logical volumes. Eventually designers realized that interposing storage servers had some advantages, but this block access model was retained. IBM's General Parallel File System (GPFS) uses this model, with software on the server side providing access to storage over an interconnection network such as Gigabit Ethernet. Figure 3a shows the division of functionality in such a parallel file system. On the client side, PFS software translates application I/O operations directly into block accesses on specific file servers and obtains permission to access those blocks. File servers receive block access requests and perform them on locally accessible storage.

An important development that shaped more recent PFS designs was the concept of *object storage devices*. Object storage devices (OSDs) are storage devices (e.g., disks, disk arrays) that allow users to store data in logical containers called objects. By abstracting away block locations, file systems built using this abstraction are simpler to build, and blocks can be managed locally, at the device itself. Recently developed parallel file systems, such as Lustre [5], PVFS [2], and PanFS [6], all use object-based access. Figure 3b shows how functionality is divided in such a design. Application accesses are translated into accesses to objects stored on file



File Systems. Fig. 3 The two most common approaches to data storage in parallel file systems are block-based (*left*) and object-based (*right*)

servers. File servers translate these accesses into locations on locally accessible storage and manage allocation of space when needed.

A less obvious advantage of the object-based approach is the possibility of *algorithmic distribution* of file data. This relates to file system metadata.

Metadata Storage and Access

There are three major components of PFS metadata: directory contents, file and directory attributes, and file data locations. Directory contents are simply the listing of files and directories residing in a single directory. Typically the contents for a single directory are stored on a single server. This can become a bottleneck for workloads that generate a lot of files in a single directory, so some parallel file systems will split the contents of a single directory across multiple servers (e.g., GPFS [1]).

File and directory attributes are pieces of information such as the owner, group, modification time, and permissions on a file or directory. Some of this information changes infrequently (e.g., owner), while

other information can change rapidly (e.g., modification time). Some parallel file systems keep all these attributes on a single server, while others distribute the attributes for different files to different servers to allow concurrent access.

File data locations are the metadata necessary to map a logical portion of a file to where it is stored in the file system. For a block-based PFS, this information generally consists of a list of blocks or (server, block) pairs, similar to the traditional inode structure in a local file system. For an object-based PFS, a list of objects or (server, object) pairs is kept along with some parameters that define how data is mapped from the logical file into the objects. In a typical round-robin distribution, a stripe unit parameter defines how much data is placed in each object for a single stripe, and this pattern is repeated as necessary over the set of objects. This algorithmic approach to data distribution has the advantages that file data location metadata does not grow over time and does not need to be modified as the file grows.

Updates to metadata are usually handled in one of two ways. First, if the PFS already implements a distributed locking system, that system might be used to synchronize access to metadata by clients. Clients can then read, modify, and write metadata themselves in order to make updates. This approach has the advantage of allowing caching of metadata as well. Second, the PFS might implement a set of atomic operations that allow clients to update metadata without locks. For example, a special “insert directory entry” operation can be used by a client to add a new file. The server receiving this message is then responsible for ensuring atomicity of the operation, blocking read operations, or returning previous state until it completes its modifications. This approach reduces network traffic for modifications, but it does not lend itself to caching of modifiable data on the compute node side.

Caching

Caching of file system data contributes greatly to the observed performance of local file systems, in part because access patterns on local file systems (e.g., editing files, compiling code, surfing the Internet) often focus on a relatively small subset of data with multiple accesses to the same data in a short period of time. Implementing caching of file system data in the local file system context is also fairly straightforward, because the file system code sees all accesses and can ensure that the cache is kept consistent with what is on storage without any communication.

Caching in the context of a parallel file system is more complex when consistency semantics are taken into account. Keeping multiple copies of file system data and metadata consistent across multiple compute nodes requires the same sort of infrastructure as needed to enforce serialization of changes to file data and to ensure notification of changes. Typically parallel file systems that allow caching of data on compute nodes rely on a distributed locking system to ensure that all cached copies are the same, or *coherent*. Thus the caches are organized much the same as a software distributed memory system.

The benefits of caching in parallel file systems are highly dependent on the applications using the system. Checkpointing operations do not typically benefit from caching, because the amount of data written usually exceeds data cache sizes, and the data is not accessed

again soon after. Likewise, large analysis applications that read datasets in parallel often exhibit access patterns that confuse typical cache read-ahead algorithms. Nevertheless, if the parallel file system is also used for home directory storage, or if applications access many small files, then caching can be beneficial.

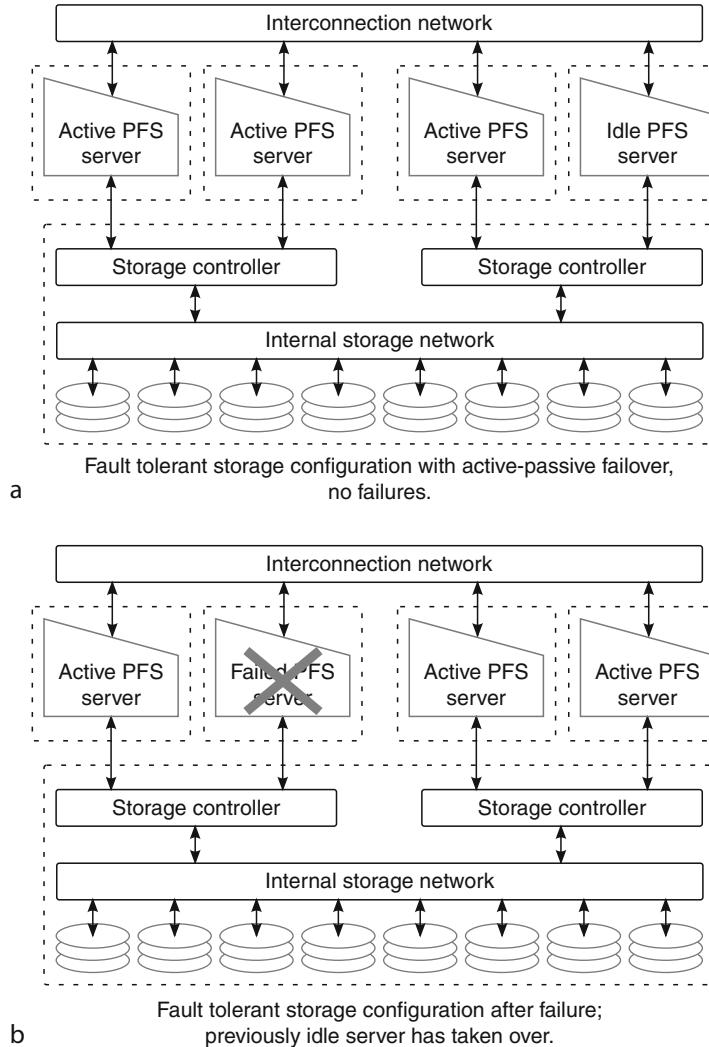
Fault Tolerance

Leading HPC systems use parallel file systems with hundreds of servers and thousands of drives. With that many components, failures are inevitable. In order to ensure that data is not lost, and that the file system remains available in the event of component failures, parallel file systems rely on a collection of fault tolerance techniques.

Nearly all parallel file systems rely on redundant array of independent disk (RAID) techniques to cope with drive failures, usually employed only on drives directly accessible by servers (i.e., not between multiple servers). These techniques use parity or erasure codes to allow data to be reconstructed in the event of a failure using the data that remains. Typically, RAID is performed on blocks of storage, but some systems perform RAID on an object-by-object basis [6].

Server failures are often handled through the use of *failover* techniques. Figures 4a and 4b depict a set of PFS servers configured with shared storage to allow for failover. In Fig. 4a, no failures have occurred. The system is running in an active-passive configuration, meaning that an additional server is sitting idle in case a failure occurs. When a server does fail (Fig. 4b), the idle server is brought into active service to take over where the previous server failed. The active-passive configuration allows for a certain number of failures to occur without degrading service. Alternatively, in an active-active configuration all servers are active all the time, with some server “doubling up” and handling the workload of a failed server when a failure occurs. This can lead to a significant degradation of performance.

While not depicted in the figures, this configuration can also tolerate storage controller failures, because all drives are accessible through either of the storage controllers. If a storage controller were to fail, the servers on the remaining functional storage controller would take over the responsibilities of the servers that could



File Systems. Fig. 4 Typical PFS configuration using shared backend storage to enable server failover, before (a), and after (b) a server failure. Compute nodes have been omitted from the figure

no longer see storage, maintaining accessibility but with obvious performance penalties.

Designs exist that do not share access to storage on the back-end [6, 7]. As opposed to the approach described above, these designs apply RAID techniques across multiple servers. This approach eliminates the need for expensive enterprise storage on the back-end, but it requires additional coordination between clients and servers in order to correctly update erasure code blocks. In the presence of concurrent writers to the same file region (e.g., writing to different portions of the same file stripe), performance can be significantly reduced.

Parallel File System Interfaces

Most applications access parallel file systems through the same interface provided for local file systems, and the industry standard for file access is the POSIX standard. As mentioned previously, the POSIX standard was developed primarily with local file systems and single-user applications in mind, and as a result it is not an ideal interface for parallel I/O. One reason is its use of the open model for translating a file name into a reference for use in I/O. This model forces name translation to take place on every process, which can be expensive at scale.

Additionally, the POSIX `read` and `write` I/O calls allow for description only of contiguous I/O regions. Thus, complex I/O patterns cannot be described in a single call, leading instead to a long sequence of small I/O operations. Moreover, the POSIX consistency semantics require atomicity of writes and immediate visibility of changes on all nodes. These semantics are stronger than are needed for many HPC applications, and require communication to enforce. An effort is underway to define a set of extensions to the POSIX standard for HPC I/O. This effort includes a set of new function calls that address the three issues outlined here. Implementations of these calls have not yet begun to appear in operating systems, but a few custom implementations do exist in specific parallel file systems, and early results are encouraging.

The MPI-IO interface standard is the only current alternative to POSIX for low-level file system access.

Related Entries

- ▶ [Checkpointing](#)
- ▶ [Clusters](#)
- ▶ [Cray XT3 and Cray XT Series of Supercomputers](#)
- ▶ [Cray XT4 and Seastar 3-D Torus Interconnect](#)
- ▶ [Exascale Computing](#)
- ▶ [Fault Tolerance](#)
- ▶ [HDF5](#)
- ▶ [IBM Blue Gene Supercomputer](#)
- ▶ [MPI-IO](#)
- ▶ [NetCDF I/O Library, Parallel](#)
- ▶ [Roadrunner Project, Los Alamos](#)
- ▶ [Synchronization](#)
- ▶ [Software Distributed Shared Memory](#)

Bibliographic Notes and Further Reading

Early Parallel File Systems

The Vesta parallel file system [8, 9] was a file system built at the user level (i.e., not in the kernel) and tested on IBM SP systems. AIX Journaled File System volumes were used for storage by file system servers. Vesta is notable for using an algorithmic distribution of data, exposing a two-dimensional view of files, and providing numerous collective I/O modes and control over consistency semantics.

The Frangipani [10] parallel file system and Petal [11] virtual block device were originally developed at Digital Equipment Corporation and are well-documented examples of one way of organizing a block-based parallel file system. The Petal system combines a collection of block devices into a single, large, replicated pool that the Frangipani system uses as its underlying storage.

Current Parallel File Systems

Currently the largest HPC systems tend to use one of four parallel file systems: IBM's General Parallel File System (GPFS); Sun's Lustre; Panasas's PanFS; and the Parallel Virtual File System (PVFS), a community-built parallel file system whose development is led by Argonne National Laboratory and Clemson University.

GPFS [1] grew out of the Tiger Shark multimedia file system. It uses a block-based approach, and can operate either with clients physically attached to a storage area network or, more typically, with clients accessing storage attached to file system servers via a shared disk model. A lock-based approach is used for coordinating access to blocks of data, and through its locking system GPFS is able to cache data on the client side and provide full POSIX semantics.

Lustre [5] is an asymmetric parallel file system that uses software-based object storage servers (OSSes) to provide data storage. It is the file system chosen by Cray for use on the Cray XT3/4/5 systems. File data is striped across objects stored on multiple OSSes for performance. A locking subsystem is used to coordinate file access by clients and allows for coherent, client-side caching. Failure tolerance is provided by using shared storage and traditional server failover.

PanFS [6] is an asymmetric parallel file system that uses a clustered metadata system in conjunction with lightweight data storage blades that provide data access via the ANSI T-10 object storage device (OSD) protocol [12, 13]. It is the file system used on the IBM Roadrunner system at Los Alamos National Laboratory. PanFS is notable in that it does not rely on shared storage on the back-end; instead clients compute and update parity information directly on data storage blades. Parity is calculated on a perfile basis, and objects in which file data is stored are spread across all storage blades to enable declustering of rebuilds in the event of a failure.

The PVFS [2] design team chose to specifically target HPC workloads with their design. It is one of the two parallel file systems deployed on the IBM Blue Gene/P system at Argonne National Laboratory; GPFS is the other. Coordinated caching and locking were not used, eliminating false sharing and locking overheads, and API extensions are available that allow complex non-contiguous accesses to be described with single I/O operations. Data is distributed algorithmically across a user-specifiable number of servers, and an object-based model is used that allows servers to locally allocate storage space. Failure tolerance is provided via shared storage between file system servers.

Acknowledgments

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory (“Argonne”). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.

Bibliography

1. Schmuck F, Haskin R (2002) GPFS: a shared-disk file system for large computing clusters. In: First USENIX Conference on File and Storage Technologies (FAST'02), Monterey, January 2002, pp 28–30
2. Lang S, Carns P, Latham R, Ross R, Harms K, Allcock W (2009) I/O performance challenges at leadership scale. In: Proceedings of Supercomputing. Portland, November 2009
3. Nieuwejaar N, Kotz D, Purakayastha A, Ellis CS, Best M (1996) File-access characteristics of parallel scientific workloads. *IEEE T Parall Distr*, 7(10):1075–1089 [Online]. Available: <http://www.computer.org/tpds/td1996/11075abs.htm>
4. Carns PH, Ligon III WB, Ross RB, Thakur R (2000) PVFS: a parallel file system for linux clusters. In: Proceedings of the 4th Annual Linux Showcase and Conference. Atlanta: USENIX Association, October 2000, pp 317–327. [Online]. Available: <http://www.mcs.anl.gov/thakur/papers/pvfs.ps>
5. Braam PJ (2003) The lustre storage architecture. Cluster File Systems, Tech. Rep., 2003 [Online]. Available: <http://lustre.org/docs/lustre.pdf>
6. Welch B, Unangst M, Abbasi Z, Gibson G, Mueller B, Small J, Zelenka J, Zhou B (2008) Scalable performance of the Panasas parallel file system. In: Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST). San Jose, 2008
7. Weil SA, Brandt SA, Miller EL, Long DDE, Maltzahn C (2006) Ceph: a scalable, high-performance distributed file system. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation. Berkeley, 2006, pp 307–320
8. Corbett PF, Feitelson DG (1996) The Vesta parallel file system. *ACM Trans Comput Syst* 14(3):225–264
9. Corbett PF, Feitelson DG, Prost J-P, Baylor SJ (1993) Parallel access to files in the Vesta file system. In: Proceedings of Supercomputing '93. IEEE Computer Society Press, Portland, 1993, pp 472–481
10. Thekkath C, Mann T, Lee E (1997) Frangipani: a scalable distributed file system. In: Proceedings of the Sixteenth ACM Symposium on Operating System Principles (SOSP). Saint-Malo, October 1997
11. Lee EK, Thekkath CA (1996) Petal: DISTRIBUTED virtual disks. In: Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems. Cambridge, October 1996, pp 84–92
12. ANSI INCITS T10 Committee, Project t10/1355-d working draft: Information technology – SCSI object-based storage device commands (OSD). July 2004
13. Weber RO (2009) SCSI object-based storage device Commands-2 (OSD-2), revision 05a. INCITS Technical Committee T10/1729-D working draft, work in progress, Tech. Rep., January 2009
14. Posix, IEEE (2004) (ISO/IEC) [IEEE/ANSI Std 1003.1, 2004 Edition] Information Technology — Operating System Interface (POSIX®) — Part I: System Application: Program Interface (API) [C Language]. IEEE, New York, NY, USA
15. Nowicki, B (1989) NFS: Network File System Protocol Specification, RFC 1094, Sun Microsystems, Inc. March, 1989, <http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc1094.html>

Fill-Reducing Orderings

► [METIS and ParMETIS](#)

First-Principles Molecular Dynamics

► [Car-Parrinello Method](#)

Fixed-Size Speedup

► [Amdahl's Law](#)

FLAME

► [libflame](#)

Floating Point Systems FPS-120B and Derivatives

CHRIS HSIUNG
Hewlett Packard, Palo Alto, CA, USA

Synonyms

[SIMD \(single instruction, multiple data\) machines](#)

Definition

FPS AP-120B was the first commercially successful attached array processor product designed and manufactured by Floating Point Systems Inc., in 1976. It is designed to be attached to a host computer, such as the DEC PDP-11, as a number crunching accelerator. The key innovation behind this family of array processors is the horizontal micro-coding of the wide instruction set architecture with pipelined execution units to achieve high parallelism. Floating Point Systems Inc., later introduced several derivative products to 64-bit arithmetic and floating point calculations, by employing higher speed silicon technology, VLSI, and multiple floating point coprocessors.

Discussion

Introduction

Floating Point Systems (FPS) Inc. was founded in 1970 by former Tektronix engineer C. Norm Winningstad in Beaverton, Oregon, to manufacture low-cost, high performance attached floating point accelerators, mainly, for minicomputers, targeting signal processing applications.

FPS AP-120B was the first attached processor product marketed under the company's own brand. It was co-designed by George O'Leary and Alan Charlesworth, first delivered in 1976. Its pipelined architecture gave it a peak performance of 12 MFLOPS (Million Floating-point Operations Per Second), with 38-bit floating point calculations. By 1985, roughly 4,400 machines had been delivered [1]. Since FPS-120B

was designed for mini-computer host machines, such as the DEC PDP-11, a larger memory version, FPS-190L, was also introduced for mainframe hosts such as the IBM-370 series.

In order to expand to more general purpose scientific and engineering applications, FPS announced the FPS-164 system in 1981. The word length was extended from 38-bit to 64-bit. The addressing capability was expanded from 16-bit to 24-bit, and the memory size grew to 7.25 MW (Million 64-bit Words). Even though the peak speed went down a bit, from 12 MFLOPS to 11 MFLOPS, the size of the problem and the precision of the calculations have both increased. The silicon technology remained the same, TTL.

In 1984, FPS announced the first 64-bit machine at higher speed with enhanced architecture, which is the FPS-164/MAX. This was the first VLSI design which incorporated matrix accelerator (MAX) boards. This is a SIMD (Single Instruction stream and Multiple Data stream) design that can incorporate up-to 15 MAX boards, each board having computation power equivalent to that of two FPS-164 processors. With a total of 31 FPS-164 CPU's, the peak computational power reached 341 MFLOPS.

In 1995, FPS announced the first ECL version, FPS-264, with a peak speed of 38 MFLOPS. In the meantime, FPS-164 also went through a makeover with the new VLSI technology, which can now be attached to both the MicroVAX or the Sun workstations.

Another derivative of the FPS-120B system is a MIMD (Multiple Instruction and Multiple Data streams) version called FPS-5000, announced in 1983. It incorporated a shared-memory bus architecture among a control processor, three arithmetic coprocessors and an I/O processor, connected to a host. The control processor is the traditional AP-120B, and the arithmetic coprocessors, XP32, were a new design, but by employing the same concept as AP-120B. Since this MIMD architecture approach digressed from the array concept, we will not go into any length in this discussion.

One word of clarification regarding AP. In the high performance computing circle, AP stands for Array Processors, even though, unlike the ICL DAP family, there was not necessarily any "array" of processors presented in the architecture. So, realistically, AP can be viewed as Attached Processors, or, we can interpret "array processor" as the attached processor designed

especially for efficient array operations. For this family of systems, the common grounds are the horizontal micro-coding concept (each instruction contains multiple fields of micro-instructions, and are decoded in parallel) and the pipelined architecture.

FPS AP-120B

Hardware

The FPS AP-120B was a micro-coded, pipelined, array processor design, attached to host computers such as the DEC PDP-11/34. Data transfer between host and AP was done via MDA (Direct Memory Access). Its cycle time was 167 ns (6 MHz), with a peak speed (multiple-add) of 12 MFLOPS. For a detailed architecture block diagram, please consult with Hockney and Jesshope's book, Fig. 2.39, p. 210, in [1].

The whole idea of micro-coding was based on the concept of the (synchronous) parallel execution of multiple execution units. A single 64-bit instruction word was divided into arithmetic field (address calculation), floating point fields, and two data pad register fields, all under the control of the decoding and execution unit. The address and logic unit (ALU) was 16-bit wide. There were 16 of 16-bit S-registers for the ALU. The floating point units (FMUL and FADD) were 38-bit wide. There were two 32 data pad registers (DX and DY), each was 38-bit wide. They were both for receiving data from memory and used as operands in floating point calculations. The FADD was a two-stage adder, and FMUL was a three-stage multiplier. In order to keep the pipelines moving, they need to be explicitly “pushed.” For example, in order to push the result of a FADD operation into its destination register, we need to add another FADD operation in the next cycle, either with a valid floating add of another pair of operands (FADD DX, DY), or with another FADD operation (FADD) without any operands. The same principle is true with FMUL.

There were three kinds of memory units, the program memory, the main (data) memory, and the table memory. The program memory was $4K \times 64$ -bit. The table memory was $64K \times 38$ -bit. It contained approximations for sine/cosine and other special functions, and was used to calculate intrinsic function values. The main memory was interleaved into two banks, odd and even. For fast memory option (333 ns access time), each bank was capable of producing one result every other cycle

(two cycle latency to data pad register). Hence, in the case of fast memory, sequential accesses could be done at each cycle with no delay. For slow memory, access time 500 ns, each bank can only deliver one result every three cycles.

The floating point format of the 38-bit data was a bit unconventional in its word length. The exponent field had 10-bits in Radix-2, and the mantissa had 28-bits in 2's complement format. Thus, the exponent had a dynamic range of 10 to the power ± 153 , and the mantissa had 8-digit of accuracy. Hence, it provided much better range and accuracy compared to the IBM 32-bit format. Extra guard bits were also kept in the calculation in order to improve precision. Format conversion between host and array was done on the fly during the data move.

Data moving into, or out of, AP-120B were performed by I/O-port (IOP) or a general programmable I/O port (GPIO) by DMA operations, by stealing cycles if necessary. There were two data width, 16-bit and 38-bit. The bandwidth of moving into AP memory was 1.5 MW/s, and that of moving out was 1.3 MW/s. Special features to support real-time audio and video processing were also provided.

Software and Performance

In order to achieve optimal performance on pipelined and horizontally micro-coded machines, it was best to write compact loops, based on skillful arrangement of the critical path. If done right, it was truly a thing of beauty.

In carrying out matrix-vector type of operations, for example, the main loop body sometimes could be written in an one-line (single instruction) loop structure. Within that one line, there could be a floating multiple-add pair (might be unrelated ops, but typically “chained” together), indexing, memory ops, and conditional test and branch. Sometimes, an extra cycle could even be saved by storing/accessing some temp data in the table memory, since table memory had an independent data path. Of course, once the loop body was set, the programmer needed to wrap the initialization part and the winding-down code around the loop in order to complete the whole execution. The whole philosophy was very much like coding for vector machines, except that there were multiple instruction fields in the horizontally micro-coded instruction set.

There was no hardware support for divide in the machine. The loop body for a vector divide was three-cycles, for example. A scalar divide was much longer.

Initially, in order to use the array processor, users had to use library procedures from the main program running on the host machine. Users use library procedures to transfer data to-and-from the attached machine, and then library procedures to do the computation. Library procedures mainly were used to execute vector (array) operations including intrinsic functions, basic arithmetic operations, signal processing library, image processing library (including 2D FFT, convolution, filtering), and advanced math library (inc. sparse matrix solution and eigenvalues, Runge-Kutta integration).

The next step was the availability of cross Fortran compiler on the host machine, to compile Fortran IV code into object code for the AP, and executed on the array processor side. With this offering, the AP became much more versatile.

But, since AP-120B did not provide a real time clock, most timing analyses were done on the host side, through APSIM, a timing simulator. Of course, with the use of a simulator, which was 1,000 times slower, it could only be useful for shorter running codes.

FPS-164

The FPS-164, announced in 1981, was a 64-bit enhancement from the 38-bit AP-120B. It incorporated some VLSI components in its design. The cycle time was at 182 ns, a bit slower than its predecessor. It made for 11 MFLOPS peak speed. Other than the double precision design, there are other improvements as well. First, the instruction memory (program memory) and the data memory were merged into one large main memory. Instruction cache was loaded automatically from the instruction memory as needed. Hence this instruction cache (size: 1,024 64-bit) replaced the program memory of the AP-120B. There was also a stack memory, consisting of 256 32-bit entries used to store subroutine return addresses, to expedite procedure calls. The table memory, also called auxiliary memory, has primarily become random access memory, used to store intermediate data from registers (e.g., register overflow area). However,

the first 8K or memory space was read-only, used to store commonly used constants and table values.

The main memory consisted of 12 memory modules. Each module had two boards (hence two banks). The memory size ranged from the initial 1.5 MW (with 16K-bit dynamic NMOS) to 7.25 MW later (with 64K-bit dynamic NMOS). All memory accesses were done in a pipelined fashion, both for main memory and table memory. Memory mapping and protections were also available.

The FPS-164 did provide programmable real-time clock and a CPU timer, which enabled accurate timing of program executions.

FPS-164/MAX

The FPS-164/MAX was introduced in 1984. The “MAX” here stood for matrix algebra accelerator. The machine consisted of one FPS-164 processor, and up-to 15 MAX boards. Each board was capable of two FPS-164 CPU. Besides, four vector registers were added, each with 2,048 elements, at each CPU. In total, the computing power consisted of 31 164-CPUs, and had a peak performance of 341 MFLOPS.

The MAX boards could execute just a handful of vector operations (SDOT, vector multiple, vector add, scalar vector multiple add, etc.). The logic was implemented with CMOS VLSI technology. The vector registers were memory mapped with the top 1 MW of the 16 MW main memory. That means, vector operations were executed simply by reading from, or writing to, appropriate memory locations.

Of course, in order to take advantage of the extra computing power, the arrays had to be very long. Hence, this type of SIMD architecture was more of a special purpose machine.

FPS-264

The FPS-264, announced in 1985, was an (custom air-cooled Fairchild 100K) ECL implementation of the (Schottky TTL technology) FPS-164. It increased the clock cycle time from 182 to 53 ns, a speed up of more than three times. Other improvements in the instruction cache and data memory (static NMOS) boasted a total improvement of 4x to 5x over the FPS-164 generation.

Bibliographic Notes and Further Reading

For interested readers, the best textbook that covers the complete family of FPS products can be found in [1]. For a complete history of all parallel computers, please find it in [2].

In 1985, FPS brought to market its long anticipated T-Series product, a hypercube, distributed memory, message passing multicomputer, with Occam software. Because of its arcane software environment, and non-general-purpose nature, it was not very successful in the marketplace. Since then, it changed direction, and brought FPS-500 product family to the market. These developments are beyond the scope of this entry. Interested readers can find more details from: "FPS Computing: A History of Firsts," by Howard Thraikill, former President and CEO of FPS, in [3].

Bibliography

1. Hockney RW, Jesshope CR (1988) Parallel computers 2: architecture, programming, and algorithms. IOP, Bristol
2. Wilson GV The history of the development of parallel computing. <http://ei.cs.vt.edu/~history/Parallel.html>
3. Ames KR, Brenner A (1994) Frontiers of supercomputing II: a national reassessment. University of California Press, Berkeley

Flow Control

JOSÉ FLICH
Technical University of Valencia, Valencia, Spain

Synonyms

Backpressure

Definition

Flow control is a synchronization protocol for transmitting and receiving units of information. It determines the advance of information between a sender and a receiver, enabling and disabling the transmission of information. Since messages are usually buffered at intermediate switches, flow control also determines how resources in a network are allocated to messages traversing the network.

Discussion

Flow control is defined, in its broad sense, as a synchronization protocol that dictates the advance of information from a sender to a receiver. Flow control determines how resources in a network are allocated to packets traversing the network. First, basic definitions and types of flow control are introduced. Then, differentiation between flow control and switching is highlighted. Finally, basic flow control mechanisms are described and briefly compared.

Messages, Packets, Flits, and Phits

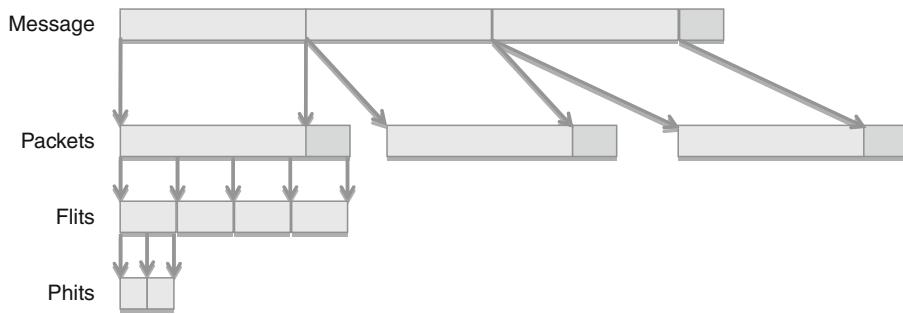
Usually, the sender needs to transmit information to the receiver. To do this, the sender transmits blocks of information, referred to as messages, through a path. A header including the routing information is prepended to the message. Depending on the switching technique used (see below and the "switching" entry), the message may be further divided into packets. A packet is made up of a header (with the proper information to route the packet through the network) and a payload.

Every message or packet is then further split into *flits*. The *flit* (flow control bit) term is defined as the minimum amount of information that is flow controlled. *Flits* can then be divided into smaller data units called *phits* (physical units). A *phit* is usually the amount of information that is forwarded in a cycle through a network link. Depending on the link width a certain number of *phits* will be required to transmit a *flit*. *Phits* are not flow controlled, they only exist because of the link width. Figure 1 shows the breakdown of a message into packets, *flits*, and *phits*.

Not all the communication systems use these four types of data units. Indeed, in some networks, packets are not needed and the entire message is transmitted (see *Wormhole Switching* in *Switching Techniques* entry). Also, in some networks, the *flit* size equals the *phit* size, and thus, there is no need to split a *flit* into smaller *phits*. In addition, as will be highlighted later, the *flit* size can span from few bits to the entire packet size.

Types of Flow Control

The sender and receiver can be connected in different ways, for instance they can be directly connected through a link, or they can be connected through a



Flow Control. Fig. 1 Messages, packets, flits, and phits

network (made of routers, switches, and links) where several routers/switches are visited, and links traversed, prior to reaching the receiver. In both cases, a flow control protocol is defined to set up the proper transmission of information. Therefore, different levels of flow control can be used at the same time: end-to-end flow control and link-level flow control.

Bufferless Flow Control Versus Buffered Flow Control

Usually, flow control is used to prevent the overflow of buffering resources when transmitting information. When the queue at the receiver gets full then the flow control must stop the transmission of information at the sender. Once there is room at the queue the flow control resumes the transmission of information. Therefore, flow control is closely related to buffer management.

However, there are cases where buffers (or queues) are not used along the transmission path of messages. In that case a bufferless flow control is defined. This is the simplest form of flow control. In that case information is simply forwarded, and potentially the information gets discarded or misrouted (due to contention along the path from the sender to the receiver). A bufferless flow control is also used in the so-called lossy networks where packets may be dropped (discarded) at receiver when the buffer fills up. In such networks, the sender will need to retransmit the packet (via time-out or a NACK control packet sent back from the receiver from upper communication layers, e.g., transport layer). However, in buffered networks, some flow control mechanism is usually used to prevent packet dropping, and such networks are known as lossless (or flow controlled) networks. This is an important distinction of types of networks with huge implications. Indeed,

the type of network (lossy or lossless) constrains the solutions for packet routing, congestion management, and deadlock issues. Also, network performance will be significantly different. Typically, storage/system-area networks (SAN) used in parallel systems and on-chip networks are lossless and local area networks (LAN) and wide area networks (WAN) are lossy.

Flow Control Versus Switching

Flow control is also closely related to switching. Switching determines when and how the information advances through a switch or router. Nowadays, the most frequently used switching mechanisms are *wormhole* and *virtual cut-through*. One of the key differences between *wormhole* switching and *virtual cut-through* switching lies in the size of buffers and messages. In *wormhole* switching buffers at routers are usually smaller than message sizes, having slots for only a few *flits* (and messages can be tens of *flits* in size or even much larger). However, in *virtual cut-through* switching, buffers must be larger than packets so as to store an entire packet in a router. Indeed, in *virtual cut-through*, whenever a packet blocks in the network the packet needs to be stored completely at the blocking router. On the contrary, in *wormhole* switching the message is spread along its paths across several routers. This fact greatly affects the flow control mechanism and, indeed, affects the *flit* size of the flow control protocol. In *wormhole* switching, the *flit* size is a small part of the message, whereas in *virtual cut-through*, the *flit* size is the size of the packet. Because original messages from the sender could be larger than buffers, packetization (splitting a message into bounded packets) is required in *virtual cut-through*.

As information may travel across multiple routers, the switching inside a router is also seen as a means of flow control. Indeed, some authors do not distinguish between flow control and switching [1], and they describe most used flow control mechanisms as *worm-hole* flow control and *virtual cut-through* flow control. However, a basic distinction exists depending on where the flow control is applied. If applied inside a router or switch then switching mechanism is used and if applied over a link connecting two nodes then flow control is used. Keep in mind also the end-to-end flow control as a third possibility. **Figure 2** shows a pair of connecting devices (source and destination nodes) connected through a series of switches and links. The figure illustrates the different levels of flow control (end-to-end and link-level) and where switching is performed (inside the switches). Taking into account this basic differentiation between flow control and switching, the following sections describe the most frequently used link-level flow control mechanisms. They are described linked to the suitable switching mechanisms they usually work with.

Another key property that distinguishes flow control from switching is that flow control is used as a means to provide backpressure not to overflow receiving buffers. Instead, switching is related to decide when and how information is switched from input ports of a switch/router to output ports of the switch/router.

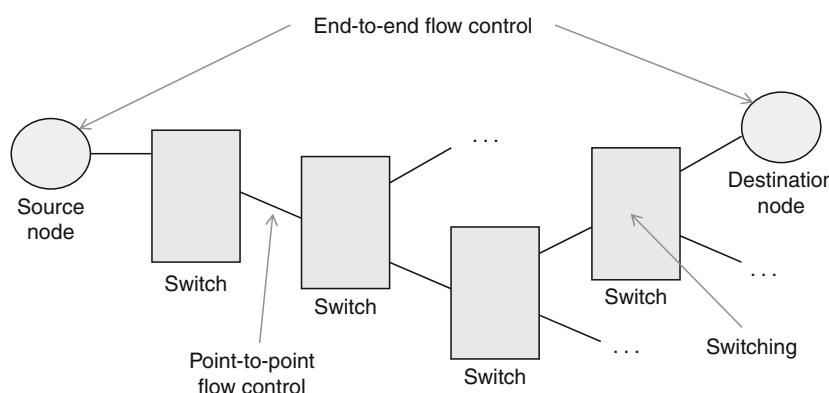
Impact of Flow Control and Buffer Size

Flow control is a key component in a network. An optimized flow control protocol will keep a fraction

of the buffers used most of the times and will minimize the time *flits* are seated waiting at queues, thus achieving high performance in the network. However, a suboptimal flow control protocol may keep resources underutilized and *flits* blocked most of the time waiting at buffers, thus ending up in a network with low performance (low throughput and high latencies).

In a flow control protocol, the buffer resources at the receiver end must be efficiently used to keep maximum performance. To do this, the buffer depth must be sized according to the latency of the path between the transmitter and the receiver (the time it takes for a flit to travel from the transmitter to the receiver). As both receiver and transmitter will exchange flow control information, the buffer must be sized, at minimum, to allow for the round-trip time of the link. This value is the path latency multiplied by two (two way transmission) plus the time it takes to generate and process the flow control command at both sides. Notice that this is needed in any type of flow-controlled transmission, either link-level or end-to-end.

Flow control is also highly related to congestion management. Congestion management deals with high levels of blocking in the network among different messages. To solve such a situation, a congestion management technique may reduce the injection of traffic from a router or from an end node, thus resembling flow control methods (as they dictate when flows advance). However, the reader must be aware that in a congestion management technique the goal is to reduce the blocking experienced in the network, whereas in a flow control method the goal is not to overrun a buffer.



Flow Control. Fig. 2 Different levels of flow control

Link-Level Flow Control

The most commonly used link-level flow control protocols are Xon/Xoff, credit-based, and Ack/Nack. However, there are other solutions that, although basic and probably low in performance, help to better understand the concept. This is the case for a handshake protocol. Next, each of these flow control protocols is described.

Handshake protocol. The most straightforward method to implement flow control among two devices is by setting a handshake protocol. As an example, a sender may assert a request signal to the receiver thus notifying it has a *flit* to send. The receiver (depending on its availability for storing new flits) may assert a grant signal to let the transmitter know that there is slot at the queue. Then, the transmission of the *flit* may begin. For the next *flit* the handshake protocol needs to be repeated (the handshake works at the *flit* granularity). Although such protocol is correct and ensures that no queue will overrun, the problem is the low performance achieved in the transmission. Indeed, the delay incurred in the control signal exchange is not overlapped with the transmission of *flits*, and therefore, there are bubbles introduced along the transmission path. Another potential problem is the excessive exchange of control information (request, grant signals) between the transmitter and the receiver for every transmitted *flit*. The following link-level flow control methods reduce such overhead.

Xon/Xoff flow control. Xon/Xoff flow control is also known as Stop&Go. It is mostly used in networks with *wormhole* switching. Basically, the receiver notifies the sender either to stop or to resume sending *flits* once high and low buffer occupancy levels are reached, respectively. To do this, two queue thresholds are set at the receiving side. When the queue fills and passes the Xoff threshold, then the receiver side sends an *off* flow control signal to the transmitter to stop transmission (in order to prevent buffer overrun). At the transmitter side, when the *off* control signal is received the transmission is stopped and the sender enters the *off* mode. Transmission will be resumed only when notified by the receiver. Indeed, when the queue drains and reaches the Xon threshold, then the receiver sends an *on* flow control signal to the transmitter. The transmitter enters again the *on* mode and resumes transmission.

Xon and Xoff thresholds are critical from the performance point of view. Indeed, Xoff threshold must be set not to overrun the receiving queue, and the Xon threshold must be set not to introduce bubbles along the transmission path (once forwarding of flits is resumed). Therefore, the round-trip time must be taken into account when setting the thresholds. In addition, Xoff and Xon thresholds must be away enough so as not to send many *off* and *on* control signals unnecessarily. It is common to set the Xoff threshold at 2/3 and the Xon threshold at 1/3 of buffer capacity. Control signals may be sent back to the sender either by using special control signals (thus not using the link data path) or using special control packets through the link data path. With appropriate thresholds, Xon/Xoff significantly reduces the amount of flow control information exchange. However, it requires large buffer memories to avoid buffer overruns.

However, there are other approaches to set up the thresholds that may be beneficial. This is the case for the Prizma switch [6] where both thresholds are set to the same value. This is efficient if the link length is short enough (round trip time is short) as is the case for the Prizma switch. The benefit in this case is the use of a single control signal that when set means a Go, and when reset means a Stop.

Credit-based flow control. Credit-based flow control is usually linked to networks using *virtual cut-through* switching. Indeed, the flit size is the packet size and, therefore, packet size is limited to a maximum (buffer size needs to be larger or equal than packet size). In credit-based flow control the sender has a counter of credits. The counter is initially set to the number of *flits*/slots available at the receiver side. Typically, a credit means a slot for a *flit* is granted at the receiver side. There are, however, systems where the credit means a chunk of bytes not necessarily the size of the packet (e.g., in InfiniBand network the credit means a slot for 64 bytes).

The sender transmits a *flit* whenever it has credits available. If the credit counter is different from zero, then a *flit* can be transmitted, and the counter is decremented by one. If the counter reaches zero then the transmitter cannot send more *flits* to the receiver.

The receiver keeps track of receptions and transmissions. Whenever a *flit* is consumed the receiver sends a new credit upstream. This is notified by a flow

control signal (again this info can travel decoupled from the transmission link or as a control packet through the transmission link). Alternatively, and to reduce the overhead in transmitting control information, the receiver may collect credits and send them in a batch. However, in that case, performance may suffer since forwarding of *flits* may be delayed.

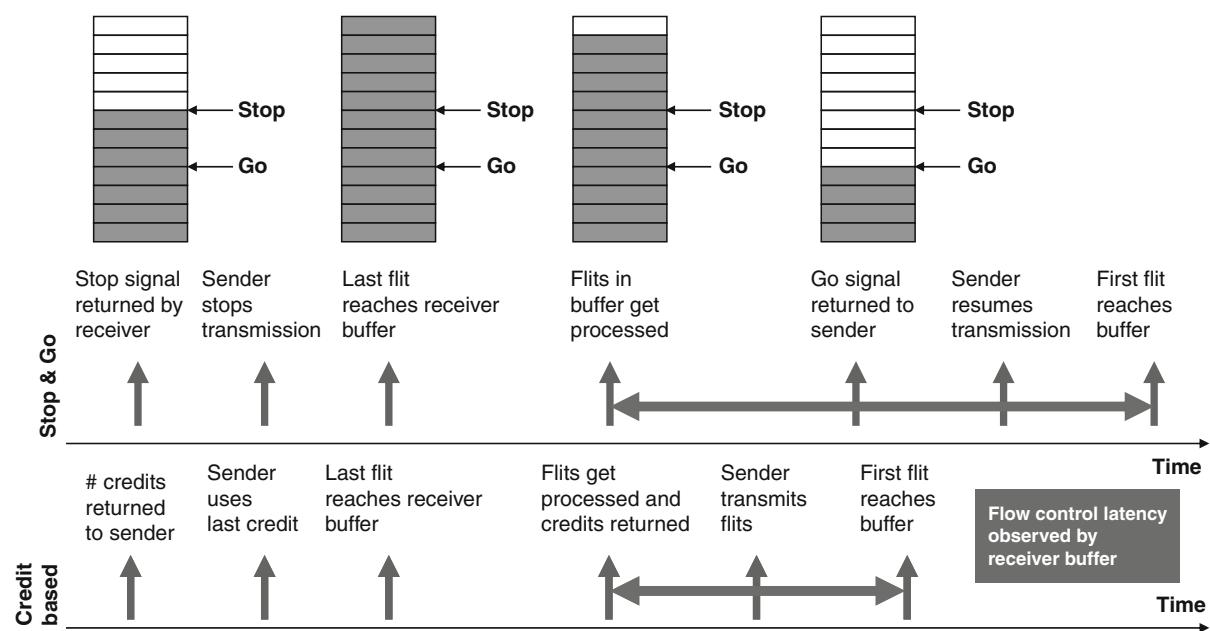
Credit-based flow control requires less than half the buffer size required by Xon/Xoff flow control, thus being more effective regarding resource requirements. In addition, the link reaction time is shorter in credit-based flow control. When the sender runs out of credits then it stops sending *flits*. When a single credit is returned from the receiver then the transmission is resumed. In Xon/Xoff the go threshold needs to be reached before sending the go signal, thus incurring higher transmission latencies. **Figure 3** shows a comparison between the Stop&Go flow control protocol (Xon/Xoff) and the credit-based flow control. As can be seen, the credit-based flow control protocol is able to resume forwarding flits sooner than Stop&Go.

Ack/Nack flow control. Flow control is also linked to fault-tolerance in message transmissions. Indeed, the

Ack/Nack flow control allows the receiver to notify the sender that the last *flit* sent was correct (received without error) or erroneous (due to a failure or even because there was no buffer available). There is no state information at the transmitter side. Indeed, the transmitter is optimistic and keeps sending *flits* to the receiver. In this sense, Ack/Nack protocol is also known as optimistic flow control.

This flow control method, although it reduces the latency (the sender just transmits the *flit* when it is available) has a poor bandwidth utilization (sending flits that are dropped). In addition, the transmitter needs logic to deal with timeouts in order to trigger retransmissions (in the absence of either an ACK or NACK signal). Also, it needs to keep a copy of the *flit* until it receives an ACK signal. Thus, the protocol incurs in an inefficient buffer utilization.

Xon/Xoff and credit-based flow control methods do not support failures. Indeed, these protocols only care about buffer flooding. When using such protocols and in the face of a failure, the system needs to rely on a higher-level flow control protocol, most of the time being an end-to-end flow control protocol.



Flow Control. Fig. 3 Flow control comparison (Xon/Xoff vs credits)

Other Specific Flow Control Protocols

Depending on the type of network used, other flow control protocols may be used trying to add new functionalities, like higher performance, lower *flit* latencies, fault coverage, etc. One example is T-ERR [5] flow control protocol, proposed for on-chip networks. T-ERR increases the operating frequency of a link with respect to a conventional design. In such situations, timing errors become likely on the link. These faults are handled by a repeater architecture leveraging upon a second clock to resample input data, so as to detect any inconsistency. Therefore, T-ERR obtains higher link throughput at the expense of introducing timing errors.

Flow Control and Virtual Channels

An important issue related to flow control is how to deal with virtual channels (► [Switch Architecture](#)). Flow control can be instantiated for each virtual channel so that individual virtual channels do not get overflowed. However, this leads to the assumption that buffer resources are statically assigned to each virtual channel, leading to low memory utilization. The problem with this approach comes from the fact that every virtual channel must be deep enough to cover the round-trip time of the link and avoid introducing bubbles in the transmission line. To reduce such overhead, it is common to design a single (and shared) landing pad at the input port of the switch. The landing pad is flow controlled. Flits arriving to the landing pad are then moved to the appropriate virtual channel.

More interesting approaches rely on the dynamic assignment of memory resources to virtual channels. Thus virtual channels may grow in depth depending on the traffic demands. In order to cope with such dynamicity, it is common to combine two flow control mechanisms. One example is the use of the Stop&Go flow control protocol per virtual channel and the credit-based flow control protocol for the entire memory at the input port of the switch. Thus, whenever a switch wants to forward a flit to the next switch, it needs to check whether there are global credits available at the input port and whether the receiving virtual channel is in go state. Notice that with the proper setting of thresholds a particular

virtual channel will not consume the entire memory resources, thus leaving room for other virtual channels.

End-to-End Flow Control

Link-level flow control protocols can, in principle, be applied in end-to-end scenarios. Indeed, nothing impedes them to work. This is the case for the Ack-/Nack protocol implemented at end nodes, in order to guarantee the reception of messages at destination. However, since the round trip time is longer, large buffers are required at both sides to transmit multiple messages while waiting for ACK signals. Usually this is not a major problem, since end nodes implement much larger buffer resources than routers and switches.

There are, however, other protocols designed at that level. Indeed, these flow control protocols are known also as message protocols. Two examples are the eager protocol and the rendezvous protocol.

The eager protocol just assumes the receiver will have enough allocated space at its buffers to store the transmitted message. In case the receiver was not expecting the message then it has to allocate more buffering and copy the message. During that time, the network may fill up. Thus, the eager protocol may impact performance.

In contrast, the rendezvous protocol does not send a message unless the receiver is aware of it. This is of particular interest when the amount of information to send is large. When using the rendezvous protocol, the transmitter first sends a single short message requesting buffer storage at the reception so as to allocate a large buffer. The receiver, after allocating the buffer, sends an acknowledgment to the sender. Upon reception, the sender injects all the messages, which include a sequence number in their header. The receiver just stores every incoming message at the correct buffer slot, depending on the sequence number the message has. Upon reception of all the messages the data info is at the receiver and ordered.

The use of the eager or the rendezvous protocol may also affect the way programs are coded. This is the case for MPI programming where different commands can be implemented with different end-to-end flow control protocols. Also, out of order delivery of

messages can be avoided with the rendezvous protocol. As there are applications that do not work in networks with unordered delivery of messages (e.g., using adaptive routing), then the choice of the rendezvous protocol may be a good choice for exploiting the benefits of adaptive routing without suffering out-of-order delivery.

Related Entries

- ▶ Congestion Management
- ▶ Switching Techniques

Bibliographic Notes and Further Reading

There are several books that cover the field of interconnection networks. High-performance interconnection networks (used in Massively Parallel Systems and/or clusters of computers) are covered in the books of Duato [2] and Dally [1]. Both offer a good text for flow control and in some cases complementary views. In particular, Dally's book couples flow control with switching providing a common text for both terms. In Duato's book the term is clearly differentiated and is more focused on switching (rather than flow control).

For more specific and emerging scenarios, like on-chip networks, the book by De Micheli and Benini [3] provides alternative flow control protocols (link-level flow control in Chap. 4 and end-to-end flow control protocols in Chap. 5). For computer networks (LANs and WANs) the reader should focus on books like Tanenbaum [4] (Chap. 6 for the transport layer and Chap. 3 for the data link layer).

Bibliography

1. Dally W, Towles B (2004) Principles and practices of interconnection networks. Morgan Kaufmann, San Francisco, CA
2. Duato J, Yalamanchili S, Ni N (2002) Interconnection networks: an engineering approach. Morgan Kaufmann, San Francisco, CA
3. de Micheli G, Benini L (2006) Networks on chips: technology and tools. Morgan Kaufmann, San Francisco, CA
4. Tanenbaum AS (2003) Computer networks. Prentice-Hall, Upper Saddle River, NJ
5. Tamhankar R, Murali S, Micheli G (2005) Performance driven reliable link for networks-on-chip. In: Proceedings of the Asian Pacific Conference on Design Automation, Shahghai
6. Denzel WE, Engbersen APJ, Iliadis I (1995) A flexible shared-buffer switch for ATM at Gb/s rates. Comput Netw ISDN Syst 27(4)

Flynn's Taxonomy

MICHAEL FLYNN

Stanford University, Stanford, CA, USA

Synonyms

SIMD (single instruction, multiple data) machines

Definition

Flynn's taxonomy is a categorization of forms of parallel computer architectures. From the viewpoint of the assembly language programmer, parallel computers are classified by the concurrency in processing sequences (or streams), data, and instructions. This results in four classes SISD (single instruction, single data), SIMD (single instruction, multiple data), MISD (multiple instruction, single data), and MIMD (multiple instruction, multiple data).

Discussion

Introduction

Developed in 1966 [1] and slightly expanded in 1972 [2], this is a methodology to classify general forms of parallel operation available within a processor. It was proposed as an approach to clarify the types of parallelism either supported in the hardware by a processing system or available in an application. The classification is based on the view of either the machine or the application by the machine language programmer. It implicitly assumes that the instruction set accurately represents a machine's micro-architecture.

At the instruction level, parallelism means that multiple operations can be executed concurrently within a program. At the loop level, consecutive loop iterations are ideal candidates for parallel execution provided that there is no data dependency between subsequent loop iterations. Parallelism available at the procedure level depends on the algorithms used in the program. Finally, multiple independent programs can obviously execute in parallel. Different instruction set (or instructions set architectures or simply, computer architectures) have been built to exploit this parallelism. In general, an instruction set architecture consists of instructions each

of which specifies one or more interconnected processor elements that operate concurrently, solving a single overall problem.

The Basic Taxonomy

The taxonomy uses the stream concept to categorize the instruction set architecture. A stream is simply a sequence of objects or actions. There are both instruction streams and data streams, and there are four simple combinations that describe the most familiar parallel architectures [3]:

1. SISD – single instruction, single data stream; this is the traditional uniprocessor (Fig. 1) which includes pipelined, superscalar, and VLIW processors.
2. SIMD – single instruction, multiple data stream, which includes array processors and vector processors (Fig. 2 shows an array processor).
3. MISD – multiple instruction, single data stream, which includes systolic arrays, GPUs, and dataflow machines (Fig. 3).
4. MIMD – multiple instruction, multiple data stream, which includes traditional multiprocessors (multi-core and multi-threaded) as well as the newer work of networks of workstations (Fig. 4).

As the stream description of instruction set architecture is the programmer's view of the machine, there are limitations to the stream categorization. Although it serves as useful shorthand, it ignores many subtleties of

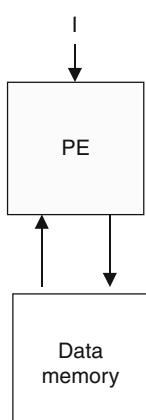
an architecture or an implementation. As pointed out, the original work in 1966 even an SISD processor can be highly parallel in its execution of operations. This parallelism need not be visible to the programmer at the assembly language level, but generally becomes visible at execution time through improved performance and, possibly, programming constraints.

SISD: Single Instruction, Single Data Stream

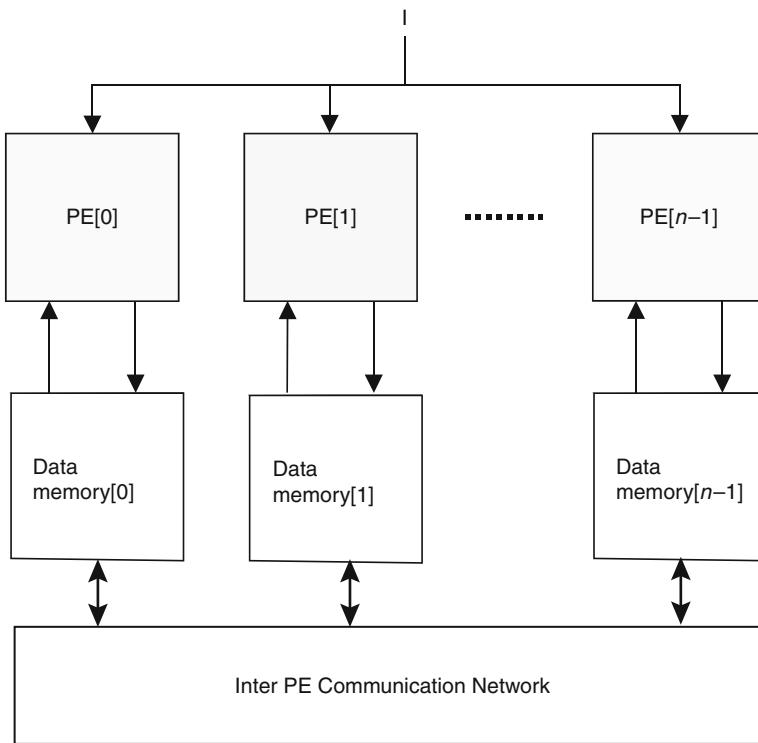
The SISD class of processor architecture includes most commonly available workstation type computers. Although a programmer may not realize the inherent parallelism within these processors, a good deal of concurrency can be present. Obviously, pipelining is a powerful concurrency technique and was recognized in the 1966 original work. Indeed, in that work, the idea of simultaneously decoding multiple instructions was mentioned but was regarded as infeasible for the technology of the time. (That paper referred to the issuance of more than one instruction at a time as a bottleneck and later publications sometimes refer to this problem as *Flynn's bottleneck*). Of course by now almost all machines use pipelining and many machines use one or another form of multiple instruction issue. These multiple issue techniques aggressively exploit parallelism in executing code whether it is declared statically or determined dynamically from an analysis of the code stream. During execution, a SISD processor executes one or more operations per clock cycle from the instruction stream. Strictly speaking, the instruction stream consists of a sequence of instruction words. An instruction word is a container that represents the smallest execution packet visible to the programmer and executed by the processor.

One or more operations are contained within an instruction word. In some cases, the distinction between instruction word and operations is crucial to distinguish between processor behaviors. When there is only one operation in an instruction word, it is simply referred to as an instruction.

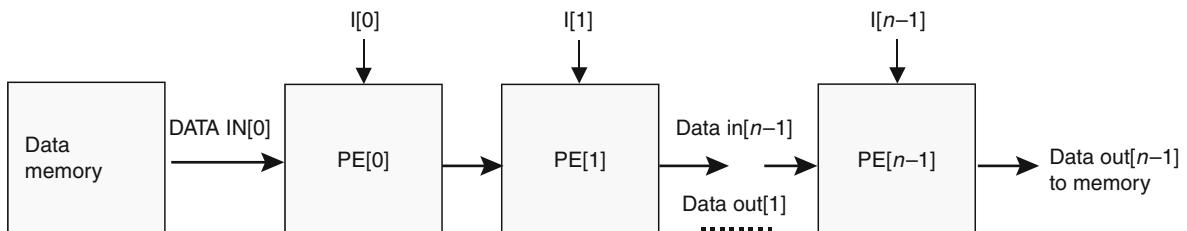
Scalar and superscalar processors execute one or more instructions per cycle where each instruction contains a single operation. VLIW processors, on the other hand, execute a single instruction word per cycle where this instruction word contains multiple operations.



Flynn's Taxonomy. Fig. 1 SISD – single instruction operating on a single data unit



Flynn's Taxonomy. Fig. 2 SIMD – array processor type using PEs with local memory



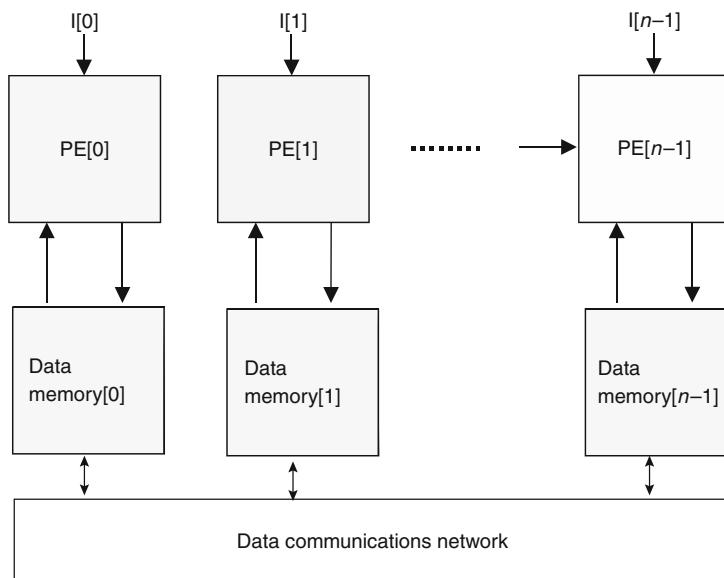
Flynn's Taxonomy. Fig. 3 MISD – a streaming processor example

The amount and type of parallelism achievable in a SISD processor has four primary determinates:

1. The number of operations that can be executed concurrently on a sustainable basis.
2. The scheduling of operations for execution. This can be done statically at compile time, dynamically at execution, or possibly both.
3. The order that operations are issued and retired relative to the original program order – these operations can be in order or out of order.
4. The manner in which exceptions are handled by the processor – precise, imprecise, or a combination.

On the occurrence of an exception (or interrupt), a precise exception handler will (usually) complete processing of the current instruction and immediately process the exception. An imprecise handler may have already completed instructions that followed an instruction that caused the exception and will simply signal that the exception has occurred and it is left to the operating system to manage the recovery.

Most SISD processors implement precise exceptions, although a few high-performance architectures allow imprecise floating-point exceptions.



Flynn's Taxonomy. Fig. 4 MIMD – ►multiple instruction

Scalar (Including Pipelined) Processors

A scalar processor is a simple (usually pipelined) processor that processes a maximum of one instruction per cycle and executes a maximum of one operation per cycle. These processors process instructions sequentially from the instruction stream. The next instruction is not processed until the execution for the current instruction is complete, and its results have been stored. The semantics of the instruction determines that a sequence of actions that must be performed to produce the desired result (instruction fetch, decode, data or register access, operation execution, and result storage). These actions can be overlapped but the result must appear in the specified serial order. This sequential execution behavior describes the sequential execution model that requires each instruction executed to completion in sequence. In the sequential execution model, execution is instruction-precise if the following conditions are met:

1. All instructions (or operations) preceding the current instruction (or operation) have been executed and all results have been committed.
2. All instructions (or operations) after the current instruction (or operation) are unexecuted or no results have been committed.
3. The current instruction (or operation) is in an arbitrary state of execution and may or may not have completed or had its results committed.

For scalar and superscalar processors with only a single operation per instruction, instruction-precise and operation-precise executions are equivalent. The traditional definition of sequential execution requires instruction-precise execution behavior at all times, mimicking the execution of a nonpipelined sequential processor.

Most scalar processors directly implement the sequential execution model. The next instruction is not processed until all execution for the current instruction is complete and its results have been committed.

Pipelining is based on concurrently performing different phases (instruction fetch, decode, execution, etc.) of processing an instruction, with a maximum of one instruction being decoded each cycle. Ideally, these phases are independent between different operations and can be overlapped; when this is not true, the processor stalls the process to enforce the dependency. Thus, multiple operations can be processed simultaneously with each operation at a different phase of its processing.

For a simple pipelined machine, only one operation occurs in each phase at any given time. Thus, one operation is being fetched, one operation is being decoded, one operation is accessing operands, one operation is in execution, and one operation is storing results. The most rigid form of a pipeline, sometimes called the



static pipeline, requires the processor to go through all stages or phases of the pipeline whether required by a particular instruction or not. Dynamic pipeline allows the bypassing of one or more of the stages of the pipeline depending on the requirements of the instruction.

There are at least two basic types of dynamic pipeline processors:

1. Dynamic pipelines that require instructions to be decoded in sequence and results to be executed and stored in sequence. This type of dynamic pipeline provides a small performance over a static pipeline. In-order execution requires the actual change of state to occur in order specified in the instruction sequence.
2. Dynamic pipelines that require instructions to be decoded in order, but the execution stage of operation need not be in order. In these organizations, the address generation stage of the load and store instructions must be completed before any subsequent ALU instruction does a write back. The reason is that the address generation may cause a page fault and affect the execution sequence of following instructions. This type of pipeline can result in imprecise exceptions mentioned earlier without the use of special order-preserving hardware.

Superscalar Processors

Even the most sophisticated scalar dynamic pipelined processor is limited to decoding a single operation per cycle. Superscalar processors decode multiple instructions in a cycle and use multiple functional units and a dynamic scheduler to process multiple instructions per cycle. These processors can achieve execution rates of several instructions per cycle (usually limited to 2 or 3, but more is possible in favorable application). A significant advantage of a superscalar processor is that processing multiple instructions per cycle is done transparently to the user and provide binary code compatibility. Compared to a dynamic pipelined processor, a superscalar processor adds a scheduling instruction window that analyzes multiple instructions from the instruction stream each cycle. Although processed in parallel, these instructions are treated in the same manner as in a pipelined processor. Before an instruction is

issued for execution, dependencies between the instruction and its prior instructions must be checked by hardware. Advanced superscalar processors usually include order-preserving hardware to insure precise exception handling, simplifying the programmer's model. Because of the complexity of the dynamic scheduling logic, high-performance superscalar processors are usually limited to processing four to eight instructions per cycle.



VLIW Processors

VLIW processors also decode multiple operations per cycle and use multiple functional units. A VLIW processor executes operations from statically scheduled instruction words that contain multiple independent operations. In contrast to superscalar which uses hardware-supported dynamic analyses of the instruction stream to determine which operations can be executed in parallel, VLIW processors rely on static analyses by the compiler. VLIW processors are thus less complex than superscalar processors, and have, at least, the potential for higher performance.

For applications that can be effectively scheduled statically, VLIW implementations offer high performance. Unfortunately, not all applications can be so effectively scheduled as execution does not proceed exactly along the path defined by the compiler code scheduler.

Two classes of execution variations can arise and affect the scheduled execution behavior:

1. Delayed results from operations whose latency differs from the assumed latency scheduled by the compiler.
2. Exceptions or interrupts, which change the execution path to a completely different and unanticipated code schedule.

While stalling the processor can control delayed result, this can result in significant performance penalties. The most common execution delay is a data cache miss. Most VLIW processors avoid all situations that can result in a delay by avoiding data caches and by assuming worst-case latencies for operations. However, when there is insufficient parallelism to hide the worst-case operation latency, the instruction schedule can have many incompletely filled or empty operation slots in the instructions, resulting in degraded performance.

SIMD: Single Instruction, Multiple Data Stream

The SIMD class of processor architecture includes both array and vector processors. The SIMD processor is created around the use of certain regular data structures, such as vectors and matrices. From the reference point of an assembly-level programmer, programming SIMD architecture appears to be very similar to programming a simple SISD processor except that some operations perform computations on these aggregate data structures. As these regular structures are widely used in scientific programming, the SIMD processor has been very successful in these environments.

Array processor and the vector processor differ both in their implementations and in their data organizations.

An array processor consists of interconnected processor elements with each having its own local memory space. A vector processor consists of a single processor that references a single global memory space and has special functional units that operate specifically on vectors.

Array Processors

The array processor consists of multiple processor elements connected via one or more networks, possibly including local and global inter element communications and control communications. Processor elements operate in lockstep in response to a single broadcast instruction from a control processor. Each processor element has its own private memory, and data is distributed across the elements in a regular fashion that is dependent on both the actual structure of the data and also on the computations to be performed on the data. Direct access to global memory or another processor element's local memory is expensive, so intermediate values are propagated through the array through local interprocessor connections, which requires that the data be distributed carefully so that the routing required to propagate these values is simple and regular. It is sometimes easier to duplicate data values and computations than it is to affect a complex or irregular routing of data between processor elements.

As instructions are broadcast, there is no means local to a processor element of altering the flow of the instruction stream; however, individual processor elements can conditionally disable instructions based on local status information – these processor elements are

idle when the specified condition occurs. Often implementation consist of an array processor coupled to a SISD general-purpose control processor that executes scalar operations and issues array operations that are broadcast to all processor elements in the array. The control processor performs the scalar sections of the application, interfaces with the outside world, and controls the flow of execution; the array processor performs the array sections of the application as directed by the control processor.

The programmer's reference point for an array processor is typically the high-level language level; the programmer is concerned with describing the relationships between the data and the computations but is not directly concerned with the details of scalar and array instruction scheduling or the details of the inter processor distribution of data within the processor. In fact, in many cases, the programmer is not even concerned with size of the array processor. In general, the programmer specifies the size and any specific distribution information for the data and the compiler maps the implied virtual processor array onto the physical processor elements that are available and generates code to perform the required computations.

Vector Processors

A vector processor is a single processor that resembles a traditional SISD processor except that some of the function units (and registers) operate on vectors – sequences of data values that are processed as a single entity. These functional units are deeply pipelined and have a high clock rate; although the vector pipelines have a long or longer latency than a normal scalar function unit, their high clock rate and the rapid delivery of the input vector data elements results in a significant throughput that cannot be matched by scalar function units. Early vector processors processed vectors directly from memory. The primary advantage of this approach was that the vectors could be of arbitrary lengths and were not limited by processor registers or resources; however, the high startup cost, limited memory system bandwidth, and memory system contention were significant limitations.

Modern vector processors require that vectors be explicitly loaded into special vector registers and stored back into memory from these registers, the same course that modern scalar processors have taken for similar reasons. The vector register file consists of several sets of

vector registers (with perhaps 64 registers in each set). Vector processors have several features that enable them to achieve high performance. One feature is the ability to concurrently load and store values between the vector registers and main memory while performing computations on values in the vector register file. This feature is important because the limited length of vector registers requires that long vectors be processed in segments. Not being able to overlap memory accesses and computations would pose a significant performance bottleneck. Just like SISD processors, vector processors support a form of result bypassing – in this case called chaining – that allows a follow-on computation to commence as soon as the first value is available from the preceding computation. Thus, instead of waiting for the entire vector to be processed, the follow-on computation can be largely overlapped with the preceding computation that it is dependent on. Sequential computations can be efficiently compounded and behave as if they were a single operation with a total latency equal to the latency of the first operation with the pipeline and chaining latencies of the remaining operations but none of the start-up overhead that would be incurred without chaining.

As with the array processor, the programmer's reference point for a vector machine is the high-level language. In most cases, the programmer sees a traditional SISD machine; however, as vector machines excel on vectorizable loops, the programmer can often improve the performance of the application by carefully coding the application, in some cases explicitly writing the code to perform register memory overlap and chaining, and by providing hints to the compiler that help to locate the vectorizable sections of the code. As languages are defined (such as Fortran 90 or High-Performance Fortran) that make vectors a fundamental data type, the programmer is exposed less to the details of the machine and to its SIMD nature.

MISD: Multiple Instruction, Single Data Stream

The MISD architecture is a pipelined ensemble where the function of the individual stages are determined by the programmer before execution is begun; then data is streamed through the pipeline, forwarding results from one (stage) function unit to the next. On the micro-architecture level this is exactly what the vector processor does. However, in the vector pipeline

the programmer initially organizes the data into suitable data structure, then streams the data elements into stages where the operations are simply fragments of an assembly-level operation, as distinct from being a complete operation. Surprisingly, some of the earliest attempts in the 1940s could be seen as the MISD computers (they were actually programmable calculators; as they were not “stored program machines” in the sense that the term is now used). They used plug boards for programs, where data in the form of a punched card was introduced into the first stage of a multistage processor. A sequential series of actions was taken where the intermediate results were forwarded from stage to stage until, at the final stage, a result would be punched into a new card.

While the MISD term is not commonly used, there are interesting uses of the MISD organization. A frequently used synonym for MISD is “streaming processor.” Some of these processors are commonly available. One example is the GPU (graphics processing unit); early and simpler GPUs provided limited programmer flexibility in selecting the function or operation of a particular stage (ray tracing, shading etc.). More modern GPGPUs (general purpose GPU) are more truly MISD with a more complete operational set at the individual stage in the pipeline. Indeed, many modern GPGPU incorporate both MISD and MIMD as these systems offer multiple MISD cores.

Other MISD style architectures include dataflow machines. In these machines, the source program is converted into a data flow graph, each node of which is a required operation. Data then is streamed across the implemented graph. Each path through the graph is a MISD implementation. If the graph is relatively static for particular data sequences, then the path is strictly a MISD if multiple paths are invoked during program execution then each path is MISD and there are MIMD instances in the implementation. Such dataflow machines have been realized by FPGA implementations.

MIMD: Multiple Instruction, Multiple Data Stream

The MIMD class of parallel architecture consists of multiple processors (of any type) and some form of interconnection. From the programmer's point of view each processor executes independently but to cooperatively execute to solve a single problem although some

form of synchronization is required to pass information and data between processors. Although no requirement exists that all processor elements be identical, most MIMD configurations are homogeneous with all processor elements identical. There have been heterogeneous MIMD configurations that use different kinds of processor elements to perform different kinds of tasks, but these configurations are usually for special-purpose applications.

From a hardware perspective, there are two types of homogeneous MIMD: multi-threaded and multi-core processors; frequently implementations use both.

Multi-Threaded Processors

In multi-threaded MIMD, a single base processor is extended to include multiple sets of program and data registers. In this configuration, separate threads or programs (instruction streams) occupy each register set. As resources are available, the threads continue execution. Since the threads are independent so is their resource usage. So ideally, the multiple threads make better use of resources such as floating point units and memory ports resulting in a higher number of instructions executed per cycle. Multi-threading is a quite useful complement to large superscalar implementations as these have relative extensive set of expensive floating point and memory resources. Critical threads (tasks) can be given priority to insure short execution while less critical tasks avail of the underused resources. As a practical matter, these designs must insure a minimum service for every task to avoid a type of task lockout. Otherwise, the implementation is straightforward since all tasks can share data caches and memory without any interconnection network.

Multi-Core and Multiple Multi-Core Processor Systems

At the other end of the MIMD spectrum are multiple multi-core processors. These implementations must communicate results through an interconnection network and coordinate task control. Their implementation is significantly more complex than the simpler MIMD or even more complex SIMD array processor.

The interconnection network in the multiprocessor passes data between processor elements and synchronizes the independent execution streams between processor elements. When the memory of the

processor is distributed across all processors and only the local processor element has access to it, all data sharing is performed explicitly using messages and all synchronization is handled within the message system.

When communications between processor elements are performed through a shared memory address space – either global or distributed between processor elements (called distributed shared memory to distinguish it from distributed memory) – there are two significant problems that arise. The first is maintaining memory consistency: the programmer-visible ordering effects on memory references, both within a processor element and between different processor elements. This problem is usually solved through a combination of hardware and software techniques. The second is cache coherency – the programmer invisible mechanism to ensure that all processor elements see the same value for a given memory location. This problem is usually solved exclusively through hardware techniques.

The primary characteristic of a large MIMD multiprocessor system is the nature of the memory address space. If each processor element has its own address space (distributed memory), the only means of communication between processor elements is through message passing. If the address space is shared (shared memory), communication is through the memory system.

The implementation of a distributed memory machine is easier than the implementation of a shared memory machine, when memory consistency and cache coherency are taken into account, while programming a distributed memory processor can be more difficult.

In recent times, large-scale (more than 1,000 cores) multiprocessors have used the distributed memory model based on implementation requirements.

Related Entries

- ▶ [Cray Vector Computers](#)
- ▶ [Data Flow Computer Architecture](#)
- ▶ [EPIC Processors](#)
- ▶ [Fujitsu Vector Computers](#)
- ▶ [IBM Blue Gene Supercomputer](#)
- ▶ [Illiac IV](#)
- ▶ [Superscalar Processors](#)
- ▶ [VLIW Processors](#)

Bibliographic Notes and Further Reading

Parallel processors have been implemented in one form or another almost as soon as the first uniprocessor implementation. Most early implementations were straightforward MIMD multiprocessors. With the advent of SIMD array processors, such as Illiac IV [4], the need for a distinction became clear. Because the taxonomy is simple, numerous extensions have been proposed; some fairly general such as those of Handler [5], Kuck [6], and El-Rewini [7]. Other extensions were of more specialized nature such as Göhringer [8], Chemij [9].

Additional comments on the taxonomy and elaborations can be found in the standard texts on parallel architecture such as Hwang [10], Hockney [11], and El-Rewini [7]. Xavier [12] and Quinn [13] are helpful texts which view the taxonomy from the programmer's view point.

Bibliography

1. Flynn MJ (1966) Very high speed computers. Proc IEEE 54:1901–1909
2. Flynn MJ (1972) Some computer organizations and their effectiveness. IEEE Trans Comput 21(9):948–960
3. Flynn MJ, Rudd RW (1996) Parallel architectures. ACM Comput Surv 28(1):67–70
4. Barnes GH, Brown RM, Kato M, Kuck D, Slotnick DL, Stokes RQ (1968) The ILLIAC IV computer. IEEE Trans Comput C-17: 746–757
5. Handler W (1982) Innovative computer architecture – how to increase parallelism but not complexity. In: Evans DJ (ed) Parallel processing systems, an advanced course. Cambridge University Press, Cambridge, pp 1–41, 0-521-24366-1
6. Kuck D (1980) The Structure of Computers and Computation vol. 1, J Wiley, New York
7. El-Rewini H, Abd-El-Barr M (2005) Advanced computer architecture and parallel processing. Wiley, New York
8. Göhringer D et al (2009) A taxonomy of reconfigurable single-/multiprocessor systems-on-chip. Int J Reconfigurable Comput 2009:11
9. Chemij W (1994) Parallel computer taxonomy, MPhil, Aberystwyth University
10. Hwang K, Briggs FA (1988) Computer architecture and parallel processing. McGraw Hill, London, pp 32–40, 0-07-031556-6
11. Hockney RW, Jesshope CR (1988) Parallel computers 2. Adam Hilger/IOP Publishing, Bristol, 0-85274-812-4
12. Xavier C, Iyengar SS (1998) Introduction to parallel algorithms, Wiley, New York
13. Quinn MJ (1987) Designing efficient algorithms for parallel computers. McGraw Hill, London, 0-07-051071-7

Forall Loops

► Loops, Parallel

FORGE

JOHN M. LEVESQUE¹, GENE WAGENBRETH²

¹Cray Inc., Knoxville, TN, USA

²University of Southern California, Topanga, CA, USA

Synonyms

Interactive parallelization; Parallelization; Vectorization; Whole program analysis

Definition

FORGE is an interactive program analysis package built on top of a database representation of an application. FORGE was the first commercial package that addressed the issue of performing interprocedural analysis for Program Consistency, Shared and Distributed Parallelization, and Vectorization. FORGE was developed by a team of computer scientists employed by Pacific Sierra Research and later Applied Parallel Research.

Discussion

Introduction

When FORGE was marketed in 1995, whole program parallelization was not quick, easy, or convenient. Vectorization was a mature user and compiler art. Many users could write vectorizable loops that a compiler could translate to efficient vector instructions. Distributed and shared memory multiprocessors could be treated as vector processors and parallelized in the same manner. This is frequently very inefficient. Parallelization introduces much more overhead than vectorization. To overcome overhead parallelization requires a much higher granularity than vectorization. Parallelization of outer loops containing nested subroutine calls is usually the best approach. Achieving this high granularity was very difficult for a user to do by hand, and practically impossible for a compiler to accomplish automatically. FORGE was created to bridge

this gap, allowing the user to interactively initiate compiler analysis and view the results. The user controlled the parallelization process using high-level program knowledge. FORGE performed the time-consuming, tedious analysis that computers are better at than programmers.

At the time, a group at Rice University, under the direction of Ken Kennedy, were developing a “Whole Program Analysis” software called Parascope. At the same time, a few of the developers of the VAST vectorizing pre-compiler began a project to develop FORGE, based on a database representation of the input application. In addition to the source code, runtime statistics could be entered into the database to identify important features derived during the execution of the application. Upon this database representation of the program an interactive, windows-based environment was built that allowed the user to interactively:

1. Peruse the database to find inconsistencies in the program
 - a. COMMON block analysis
 - b. Tracing of variable through the call chain showing aliases from subroutine calls, equivalences, and COMMON blocks
 - c. Identification of variables that are used and not set
 - d. Identification of variables that are set and not used
 - e. Queries to identify variables that satisfied certain user-defined templates
2. Interactively Vectorize Application
 - a. Perform loop restructuring
 - i. Loop splitting
 - ii. Routine inlining – pull routine into a loop
 - iii. Routine outlining – push loop into a routine
 - iv. Loop inversion
3. Interactive Shared Memory Parallelization
 - a. Array scoping
 - b. Storage conflicts due to COMMON blocks
 - c. Identification of Reduction function variables
 - d. Identification of Ordered regions due to data dependencies
4. Interactive Distributed Memory Parallelization
 - a. User-directed loop parallelization
 - b. User-directed array distribution

- c. Automatic loop parallelization based on array distribution
- d. Automatic array distribution based on parallel loop access
- e. Iteration of c. and d. for whole program parallelization
- f. Generate HPF
5. Interactive Program Restructuring
 - a. Loop Inversion
 - b. Loop splitting
 - c. Array index reordering
 - d. Routine inlining
 - e. Routine outlining
 - f. TASK COMMON generation
6. Interactive CACHE View tool
 - a. Given a hardware’s Level 1 and Level 2 Cache sizes and runtime statistics
 - i. Instrument code
 - ii. Identify Cache alignment issues
 - iii. Identify Cache overflows
 - iv. Propose Blocking strategies

General Structure

Forge was written in the C programming language. The parser was constructed using the compiler-compiler tools LEX and YACC. The windowing system used was originally a custom interface written for SUN, APOLLO, and DEC workstations. Later a MOTIF based X-Windows system was used.

The Database

To perform the aforementioned functions, FORGE needed significant information concerning the input application. At the end of the development of FORGE in 1998, the database could input the application source code (Only Fortran 77) and runtime statistics.

From the source code, the following data was extracted:

1. For each routine
 - a. Variable information
 - i. Aliasing from COMMON, EQUIVALENCE, SUBROUTINE arguments
 - ii. Read and Write references
 - iii. Data type and size
 - iv. Array dimensions
 - v. Array indexing information

- b. Control structures within the routine – IF's and DO's
- 2. Call chain information
 - a. Static Call chain – all possible calling sequences
 - b. Dynamic Call chain – derived from the runtime statistics
- 3. Variable Aliasing through call chain, both static and dynamic

Mode	Status
1 LES30	
24 I ALLOC	
33 I I EJECT	<See 9>
34 I I EJECT	<See 9>
35 I GRID	
36 I I GRID/DO <1> K	
37 I I I GRID/DO <2> J	
38 I I I GRID/DO <3> I	
39 I I GRID/DO <4> K	
40 I I I GRID/DO <5> J	
41 I I I I GRID/DO <6> I	
42 I I I GRID/DO <7> K	
43 I I I I GRID/DO <8> J	
44 I I I I GRID/DO <9> I	
45 I I I GRID/DO <10> K	
46 I I I I GRID/DO <11> J	
47 I I I I GRID/DO <12> I	
48 I I GRID/DO <13> K	
49 I I I I GRID/DO <14> J	
50 I I I I GRID/DO <15> I	
51 I I I I GRID/DO <16> I	
52 I I GRID/DO <17> K	
53 I I I I GRID/DO <18> I	
54 I I I I GRID/DO <19> J	
55 I I I I GRID/DO <20> J	
56 I I GRID/DO <21> J	
57 I I I I GRID/DO <22> I	
58 I I I I GRID/DO <23> K	
59 I I I I GRID/DO <24> K	
60 I SETIV	
61 I I SETIV/DO <1> K	
62 I I I SETIV/DO <2> J	
63 I I I I SETIV/DO <3> I	
64 I I SETIV/DO <4> LL	
65 I I I SETIV/DO <5> K	

.Variables... | .Show... | .Show Cells... | .Expand

Done

- 4. Basic Block (control flow) information for the entire application
 - a. All IF constructs controlling the execution of the basic block
 - b. All reaching definitions for variables used in the basic block
 - c. All uses of variable set in the basic block
- 5. Array section analysis
 - a. Information on every looping structure
 - i. Section of array that is accessed by each DO loop
- 6. Integer value information (FACTS) for array index and data dependency analysis

Database Design

The FORGE database was designed and implemented by the developers. An off-the-shelf database was not used. In design of the database, the generation of the database was performed incrementally with the ability to automatically update the database when the source code was modified. The intent was to spend the time building the database so the interactive analysis within the FORGE environment could be performed quickly.

FORGE stored a program as a Package. Each package contained a list of the source files in a program, along with other information such as directories to search, compiler options to use, and target hardware.

Brute force whole program analysis using the hardware available at the time was very time consuming for convenient interactive program analysis. For that reason, a database was created containing only data needed for parallelization of a program. Each program unit (subroutine, function, main program) was analyzed and stored separately in the database, on disk. Last modified dates were saved for each file. A checksum was saved for each program unit in a file. When the modification date for a file indicated that the database was out of date, the file was rescanned and the checksums for each routine compared to the checksums saved with the database. The database was regenerated only for modified routines. Building the initial database for a large program could take an hour. After the initial creation, updating the database for source file changes usually took a few seconds.

When the user selected a package for analysis, tables were built in memory combining the databases for called and calling routines. This process typically took

a few seconds. The efficiency of the incremental update of database and the handling of called routines in memory are the key features that made interactive whole program analysis feasible.

The database was designed to quickly supply information to the Vectorizer and Parallelizer when performing array section and scalar dependency analysis.

All of the analysis features used by the parallelization analysis were made available to the user for program browsing. This only required the implementation of menus, since all the analysis was already implemented. The browser ended up as one of the most popular and powerful features of FORGE.

Users could develop templates to define routines that were called from the application and whose source was not available. These templates simply supplied the information about variables that were either passed into the routine through the argument list or through COMMON blocks.

Runtime Statistics

FORGE had an option to instrument the application to gather information about important routines, DO loops, IF constructs, and Cache utilization. The instrumented program could then be run with an impor-

tant dataset and an ASCII output would be generated that could be fed into the database to assist in performing analysis. There was an option to ignore static information in favor of the dynamic information when performing the aforementioned analysis. The following runtime statistics were gathered.

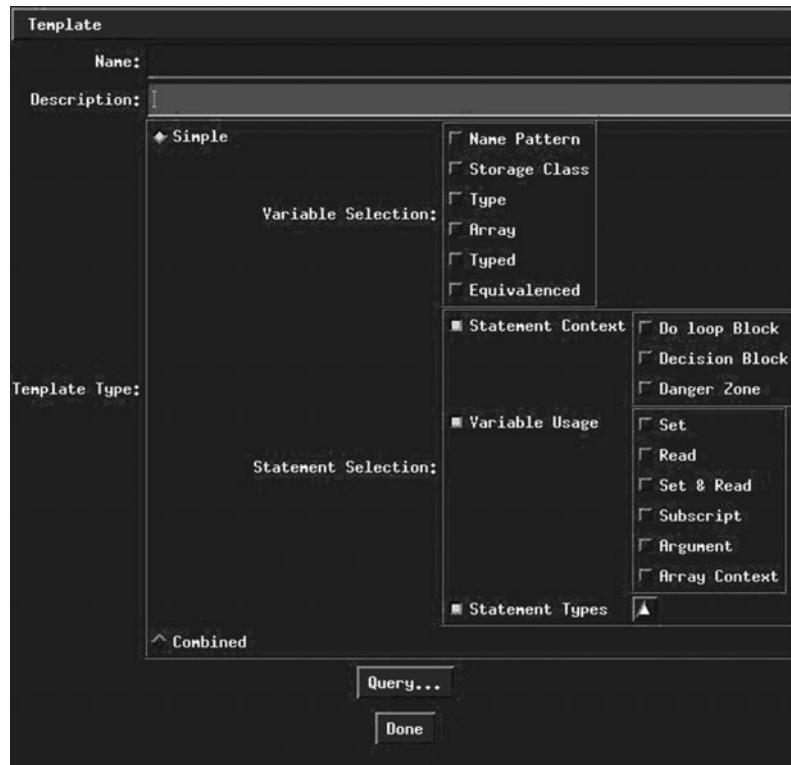
- CPU time of each routine, inclusive and exclusive
- Lengths of DO loops
- CPU time of each DO loop, inclusive and exclusive
- Frequency of IF constructs (optional)
- Array addresses (optional)

Interactive Tools available to the User

Query Database

A popular tool named Query allowed the user to find all variables in a program that matched criteria defined in templates. Templates specified criteria for:

- Variable name
- Variable type
- Variable storage
- Statement context
- Statement type





Many common templates were supplied or the user could create their own templates.

- Show all variables read into the program
- Show all variables written out
- Show all variables used within particular DO loop
- Show all variables set under a condition and used unconditionally

Cache Vu Tool

Cache Vu tool, another tools within the FORGE system, was extremely powerful and informative. The runtime statistics to support this feature was very large and had to be collected in a controlled manner. FORGE was able to visually show the user which arrays were overlapping in the cache and being evicted from cache due to cache associativity.

COMMON Block Grid

A grid was displayed with routines down the side and COMMON blocks across the top. When a COMMON block was selected, variables within that COMMON block were shown across the top showing how each variable was referenced in each routine.

USE-DEF and DEF-USE

This was extremely useful in debugging an application. Plans to attach FORGE to a debugger never materialized.

- User could click on a variable in the source and drop down a menu to select, show reaching definition, or show subsequent uses. The uses and/or sets were shown within the control structures controlling the execution of the line.

Tracing Variables Through the Call Chain

All occurrences of a variable were shown interspersed within the call chain, including IF constructs. Aliasing was shown where occurs due to EQUIVALENCE, COMMON block, or SUBROUTINE call.

Shared Memory Parallelizer

If runtime statistics were available, the window would present the user with the highest level DO loop that is a potential candidate for parallelization. If the DO loop

contained I/O or unknown routine, the user would be alerted to that. The user would then say:

1. Parallelize Automatically
2. Parallelize Interactively

The automatic mode would complete without interaction and show the user if there were any dependency regions, storage conflicts, or other inhibitors. The Interactive mode would converse with the user, showing each inhibitor as encountered. The users could then instruct the parallelizer to ignore dependencies, if appropriate, and continue.

Finally, the user could output the parallelized code in OpenMP, Cray Micro-tasking, or as a parallelized routine to be run with FORGE's shared memory runtime library. The latter was accomplished by outlining the DO loop into a subroutine call that got called in parallel. Shared variables were passed as arguments and private variables made local to the outlined routine.

Distributed Memory Parallelizer

The user could select a set of arrays to decompose, decompose the arrays through a menu – only single index decomposition was supported. FORGE would then identify every DO loop that accessed the decomposed dimension and the user could have FORGE distribute the loop iterations using the owner execute rule automatically. During the loop parallelization, FORGE searched for additional arrays to decompose and loops to parallelize. This cycle of array decomposition and loop distribution continued until no new transformations were found.

Parallelization was saved in internal format between sessions. The user could choose to save the parallelization in a new copy of the source program with HPF directives.

Executable parallel code could be generated either with the interactive parallelizer or a batch translator named XHPF. The translated code contained calls to a runtime library. Storage for distributed arrays was allocated at runtime as determined by the size of the array and the number of distributed CPUs used. Ghost cells were allocated as needed. Parallel loop bounds were computed at runtime. Internode communication was performed pre and post loop as required. Communication was inserted in serial code as required. One node was used to perform all IO operations.

```

--> partition CPK <SHRUNKBLOCK>
--> partition CONCO <SHRUNKBLOCK>
--> Spread the loop on CNUK<-2:103, -2:103, -2^1>
11:      DO K=-2,KCHRX+3
12:          DO J=-2,JCHRX+3
13:              DO I=-2,ICHRX+3
14:                  CNU(I,J,K) = CNUK
15:                  CEN(I,J,K) = 1.00*00/PRT
16:                  CEP(I,J,K) = CEPSK
17:              END DO
18:          END DO
19:      END DO
20:
--> Spread the loop on Q<-2:103, -2:103, -2:103, 1, 1^1>
--> Set of  Q<-2:103, -2:103, -2:103, 1, 1^1>
--> Set of  Q<-2:103, -2:103, -2:103, 2:ND, 1^1>
21:      DO LL = 1,2
22:          DO K = -2, KCHRX+3
23:              DO J = -2, JCHRX+3
24:                  DO I = -2, ICHRX+3
25:                      Q(I,J,K,LL) = 1.00*00
26:                      DQ(I,J,K,1) = 0.00*00
27:                  END DO
28:              END DO
29:          END DO
30:      DO LM = 2,ND
31:          DO K = -2, KCHRX+3
32:              DO J = -2, JCHRX+3
33:                  DO I = -2, ICHRX+3
34:                      Q(I,J,K,LM,LL) = 0.00*00
35:                      DQ(I,J,K,LM) = 0.00*00
36:                  END DO
37:              END DO
38:          END DO

```

Mode Status

1 LESS0
60 SETIV

2 partitioned arrays

```

61  ||| partition Q <*,*,*,*,SHRUNKBLOCK>
62  ||| partition HF <*,*,SHRUNKBLOCK,>
63
64  ||| SETIV/00 (1) K           Spread ON CNUK<-2:103,-2:103,-2^1>
65  ||| SETIV/00 (2) J           Within a spread loop
66  ||| SETIV/00 (3) I           Within a spread loop
67  ||| SETIV/00 (4) LL          Spread ON Q<-2:103,-2:103,-2:103,1,1^1>
68  ||| SETIV/00 (5) K           Within a spread loop
69  ||| SETIV/00 (6) J           Within a spread loop
70  ||| SETIV/00 (7) I           Within a spread loop
71  ||| SETIV/00 (8) LM          Within a spread loop
72  ||| SETIV/00 (9) K           Within a spread loop
73  ||| SETIV/00 (10) J          Within a spread loop
74  ||| SETIV/00 (11) I          Within a spread loop
75  ||| SETIV/00 (12) K          Spread ON HK<-2:103,-2:103,-2^1>
76  ||| SETIV/00 (13) J          Within a spread loop
77  ||| SETIV/00 (14) I          Within a spread loop
78  ||| SETIV/00 (15) NN          Within a spread loop
79  ||| SETIV/00 (16) K          Spread ON URNSC<-2:103,-2^1>
80  ||| SETIV/00 (17) J          Within a spread loop
81  ||| SETIV/00 (18) L           Within a spread loop
82  ||| SETIV/00 (19) NS          Within a spread loop
83  ||| SETIV/00 (20) NN          Spread ON RGK<1^1>
84  ||| SETIV/00 (21) NN          Spread ON CONCO<1^1>

```

Cancel analysis Expand Anchor

7 directives are created.
9 directives are created.

Program Restructurer

The parallelized code was available to the user for all the program transformations executed by FORGE. When the source code was modified, it would be presented to the user and the user could accept the new version and the database would be updated accordingly.

Parallel Program Generation

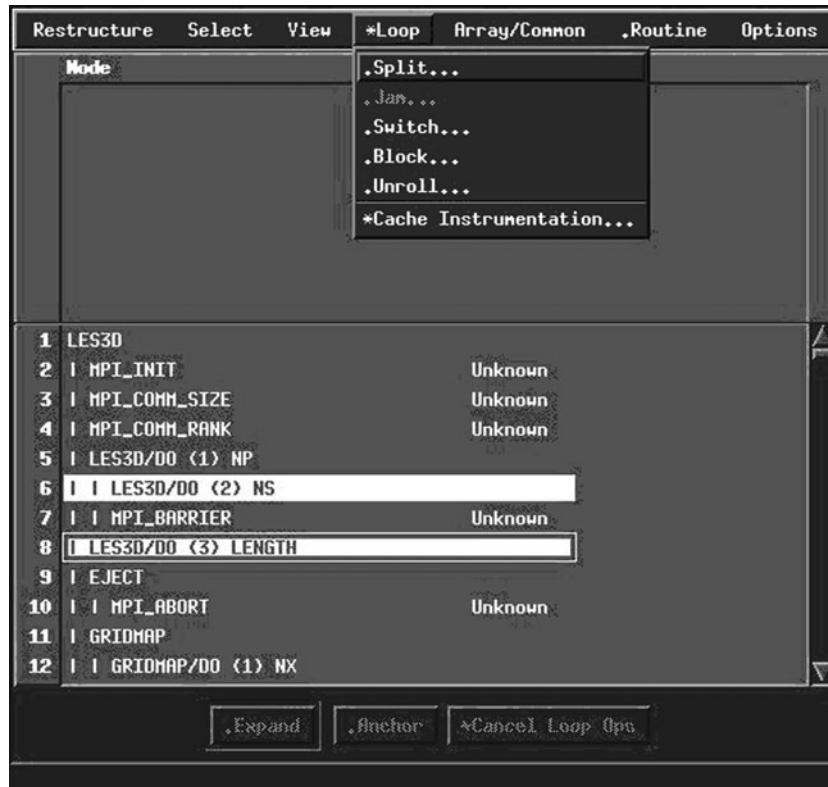
FORGE used the result of analysis to generate parallel programs. It performed source-to-source translation. The input could contain compiler parallelization directives. Syntax defined by most major vendors of the time (IBM, NEC, FUJITSU, DEC, INTEL, NCUBE) were supported, as well as HPF and OMP standards. FORGE could take any of these as input, combine them with interactive or automatic analysis, and generate directives. FORGE could also generate a parallel program containing calls to a FORGE runtime library. Versions of the library existed for shared and distributed memory parallelization for the major vendors. The user would compile the FORGE output with the vendor supplied F77 compiler, link with the appropriate FORGE library, and run the parallel application.

Instrumented versions of the FORGE parallel libraries were provided. When linked with the instrumented libraries and run, timing data was created. The timing data could be viewed interactive with FORGE, superimposed on a dynamic callchain. Efficiencies and “hotspots” were identifiable and correctable.

Batch tools, SPF – shared memory parallelizer, and XHPF (translate HPF) performed the same translation functions as FORGE. A user could modify program directives with a conventional editor and run these tools as precompilers as part of the normal compilation stack.

Future Directions

There were three major deficiencies in Forge. First, Forge handled only a subset of Fortran 90. Second, Forge was developed with Unix Xwindows interface. A convenient native MS Windows interface was needed. Last, the display of parallel loop inhibitors due to loop carried dependencies often contained many false



dependencies. The minimization of the false dependencies and a convenient method for the user to view and ignore dependencies were problems for Forge, as well as all other parallelization tools then and now. Work continued in these areas through the year 2001 when Applied Parallel Research went out of business.

Bibliography

1. Balasundaram V, Kennedy K, Kremer U, McKinley K, Subhlok J (1989) The paroscope editor: an interactive parallel programming tool. In: Conference on high performance networking and computing proceedings of the 1989 ACM/IEEE conference on Supercomputing, ACM, New York
2. Kushner EJ (1992) Automatic parallelization of grid-based applications for the iPSC/360. In: Lecture notes in computer science, vol 634, pp 637–645
3. Gernt M (1994) Automatic parallelization of a crystal growth simulation program for distributed-memory system. In: Lecture notes in computer science, vol 796, pp 281–286
4. Applied Parallel Research. FORGE 90 Baseline system user's guide
5. Applied Parallel Research. FORGE high performance Fortran xHPF user's guide
6. Applied Parallel Research. FORGE 90 Distributed memory parallelizer user's guide
7. Song ZW, Roose D, Yu CS, Berlamont J (1995) Parallelization of software for coastal hydraulic simulations for distributed memory parallel computers using FORGE 90. In: Power H (ed) Applications of high-performance computing in engineering IV, pp 203–210
8. Frumkin M, Hribar M, Jin H, Waheed A, Yan J (1998) A comparison of automatic parallelization tools/compilers on the SGI Origin2000. In: Proceedings of the 1998 ACM/IEEE SC98 Conference (SC'98), Orlando. IEEE Computer Society, Washington
9. Bergmark D (1995) Optimization and parallelization of a commodity trade model for the IBM SP1/2, using parallel programming tools. In: International Conference on Supercomputing, Proceedings of the 9th international conference on Supercomputing, Barcelona
10. Kennedy K, Koelbel C, Zima H (2007) The rise and fall of high performance Fortran: an historical object lesson - History of Programming Languages. In: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, San Diego. ACM, New York, pp 7-1-7-22
11. Saini S (1995) NAS Experiences of Porting CM Fortran Codes to HPF on IBM SP2 and SGI Power Challenge. NASA Ames Research Center, Moffett Field

Formal Methods-Based Tools for Race, Deadlock, and Other Errors

PETER MÜLLER
ETH Zurich, Zurich, Switzerland

Definition

Formal methods-based tools for parallel programs use mathematical concepts such as formal semantics, formal specifications, or logics to examine the state space of a program. They aim at proving the absence of common concurrency errors such as races, deadlocks, livelocks, or atomicity violations for all possible executions of a program. Common techniques applied by formal methods-based tools include deductive verification, model checking, and static program analysis.

Discussion

Introduction

Concurrent programs are notoriously difficult to test. Their behavior depends not only on the input but also on the scheduling of threads or processes. Many concurrency errors occur only for very specific schedulings, which makes them extremely hard to find and reproduce.

An alternative to testing is to apply formal methods. Formal methods use mathematical concepts to examine all possible executions of a program. Therefore, they are able to guarantee certain properties for all inputs and all possible schedulings of a program. Most formal methods-based tools for concurrent programs focus on races and deadlocks, the two most prominent concurrency errors. Some of them also detect other errors such as livelocks, atomicity violations, undesired non-determinism, and violations of specifications provided by the user.

This entry covers tools based on three kinds of formal methods. (1) Tools for deductive verification use mathematical logic to construct a correctness proof for the input program. (2) Tools for state space exploration (model checkers) check properties of a program by exploring all its executions. (3) Tools for static program analysis approximate the possible dynamic behaviors of a program and then check properties of these approximations. This entry summarizes the

foundations and applications of these approaches, discusses their strengths and weaknesses, and names a few representative tools for each approach.

The evaluation of the three approaches focuses on the following criteria:

Soundness. A verification or analysis tool is sound if it produces only results that are valid with respect to the semantics of the programming language. In other words, a sound tool does not miss any errors. Soundness is the key property that distinguishes formal methods from testing. However, some formal methods-based tools sacrifice soundness in favor of reducing false positives or improving efficiency.

Completeness. A verification or analysis tool is complete if it does not produce false positives, that is, each error detected by the tool can actually occur according to the semantics of the programming language. Minimizing the number of false positives is crucial for the practical applicability of a tool because each reported error needs to be subjected to further investigation, which usually involves human inspection.

Modularity. A verification or analysis tool is modular if it can analyze the modules of a software system independently of each other and deduce the correctness of the whole system from the correctness of its components. Modularity allows a tool to analyze program libraries and improves scalability.

Automation. A verification or analysis tool has a high degree of automation if it requires little or no user interaction. Typical forms of user interaction include providing the specification to be checked by the tool, assisting the tool through program annotations, and constructing proofs manually. A high degree of automation is necessary to make the application of formal methods-based tools economically feasible.

Efficiency. A verification or analysis tool is efficient if it can analyze large programs in a relatively short amount of time and space. In general, the number of states and executions of a program grows exponentially in the size of the program (for instance, the number of variables, branches, and threads). Therefore, tools must be highly efficient in order to handle practical programs.

Deductive Verification

Overview

Tools for deductive verification construct a formal proof that demonstrates that a given program satisfies its specification. Common ways of constructing the correctness proof are (1) to use a suitable calculus such as the weakest precondition calculus to compute verification conditions – logical formulas whose validity entails the correctness of the program – and then prove their validity in a theorem prover, or (2) to use symbolic execution to compute symbolic states for each path through the program and then use a theorem prover to check that the symbolic states satisfy the specification. Both approaches employ a formal semantics of the programming language. The specification to be verified typically includes implicit properties (such as deadlock freedom) as well as properties explicitly specified by the user (for instance through pre- and postconditions).

Deductive verifiers for concurrent programs apply various verification methodologies, which are illustrated by the Java example below. Monitor invariants (see entry ▶ [Monitors, Axiomatic Verification of](#)) allow one to express properties of the monitor variables (here, the fields in class `Account`) that hold when the monitor is not currently locked by any thread. For instance, one could express that the transaction count is a nonnegative number. Rely-guarantee reasoning [14] allows one to reason about the possible interference of other threads. For instance, one could prove that no thread ever decreases the transaction count. Deadlock freedom can be proved by establishing a partial order on locks and proving that each thread acquires locks in ascending order (see entry on ▶ [Deadlocks](#)). For instance, one could specify that the order on the locks associated with `Account` objects is given by the less-than order on the accounts' numbers. The check that locks are acquired in ascending order would fail for the example because the locks of `this` and `to` are acquired regardless of the order. Therefore, the concurrent execution of `x.transfer(y, a)` and `y.transfer(x, b)` might deadlock.

```
class Account {
    int number;
    int balance;
    int transactionCount;
```

```
void transfer(Account to, int amount) {
    if(this == to) return;
    acquire this;
    acquire to;
    balance = balance - amount;
    to.balance = to.balance + amount;
    transactionCount = transactionCount + 1;
    to.transactionCount = to.transactionCount + 1;
    release to;
    release this;
}

// constructors and other methods omitted
```

The absence of data races can be proved using a permission system that maintains an access permission for each shared variable. The permission gets created when the variable comes into existence; it can be transferred among threads and between threads and monitors, but not duplicated. Each shared variable access gives rise to a proof obligation that the current thread possesses the permission for that variable. Since access permissions cannot be duplicated, this scheme prevents data races. In the example above, one could store the permissions for `balance` and `transactionCount` in the monitor of their object. By acquiring the monitors for `this` and `to`, the current thread obtains those permissions and, therefore, is allowed to access the fields. When the monitors are released, the permissions are transferred back to the monitors. Such a permission system is for instance applied by concurrent separation logic [19].

An alternative technique for proving the absence of data races is through object ownership [12]. Ownership schemes associate zero or one owner with each object. An owner can be another object or a thread. A thread becomes the owner of an object by creating it or by locking its monitor. Each access to an object's field gives rise to a proof obligation that the object is (directly or transitively) owned by the current thread. Since an object has at most one owner, this scheme prevents data races.

Besides preventing data races, both permissions and ownership guarantee that the value of a variable does not change while a thread holds its permission or owns the object it belongs to. Therefore, they also give an atomicity guarantee, which allows verifiers to apply sequential reasoning techniques in order to prove additional program properties.

Discussion

Program verifiers are sound unless there is a flaw in the underlying logic or the implementation of the verifier.

In principle, program verification is complete if the underlying logic is complete with respect to the semantics of the programming language. However, program verifiers typically enforce some kind of verification methodology as described above. These methodologies simplify the verification of certain programs, but fail for others. For instance, a tool that requires lock ordering for deadlock prevention might not be able to handle programs that prevent deadlocks by other strategies. Another source of incompleteness are the limitations of theorem provers. Since the validity of verification conditions is in general undecidable, program verifiers based on automatic provers such as SMT solvers report spurious errors when the prover cannot confirm the validity of a verification condition.

A number of program verifiers support modular reasoning. For instance, a permission system removes the necessity to inspect the whole program in order to detect data races. Nevertheless, modular verification is still challenging for certain program properties such as global invariants and liveness properties.

Deductive program verifiers are typically not fully automatic. Users have to provide the specification the program is verified against. Even if the property to be proved is the absence of a simple error such as data races, users typically have to annotate the code with auxiliary assertions such as loop invariants. Moreover, interactive verifiers require users to guide the proof search.

The efficiency of automatic verifiers (those verifiers where the only user interaction is by writing annotations) depends on the number and complexity of the generated verification conditions as well as on the performance of the underlying automatic theorem prover. The former aspect is positively influenced by applying a modular verification methodology, the latter benefits from the enormous progress in the area of automatic theorem proving.

In summary, since program verifiers are sound and since modular verifiers scale well, they are well suited for software projects with the highest quality requirements, especially safety-critical software. However, the overhead of writing specifications as well as limitations of existing verification methodologies and automatic

theorem provers have so far prevented program verification from being applied routinely in mainstream software development.

Examples

Most verifiers for concurrent programs are in the stage of research prototypes. Microsoft's VCC is an automatic verifier for concurrent, low-level C code. It has been used to verify the virtualization layer Hyper-V. VCC applies a verification methodology based on ownership [7]. Permission systems are used by verifiers based on separation logic such as VeriFast for C programs [13], SmallfootRG [4], and Heap-Hop [24], as well as by Chalice [15]. Both Heap-Hop and Chalice support shared variable concurrency and message passing. Lock-free data structures have been verified using the interactive verifier KIV [22].

State Space Exploration

Overview

Software model checkers consider all possible states that may occur during the execution of a program. They check properties of individual states or sequences of states (traces) through an exhaustive search for a counterexample. This search is either performed by enumerating all possible states of a program execution (explicit state model checking) or by representing states through formulas in propositional logic (symbolic model checking). In both approaches, programs are regarded as transition systems whose transitions are described by a formal semantics of the programming language. The specifications to be checked are given as formulas in propositional logic for individual states (assertions) or as formulas in a temporal logic (typically LTL or CTL) for traces. The use of temporal logic allows model checkers in particular to check liveness properties such as “every request will eventually be handled.” Liveness properties are important for the correctness of all parallel programs, but especially for those based on message passing (such as MPI programs) because they often apply sophisticated protocols. Other errors such as deadlocks can be detected without giving an explicit specification. For instance for the account example above, a model checker detects that for some input values and thread interleavings, method transfer does not reach a valid terminal

state. The inputs and interleavings are then reported as a counterexample that can be inspected and replayed by the user.

The main limitation of software model checking is the so-called state space explosion problem. The state space of a program execution grows exponentially in the number of variables and the number of threads. Therefore, even small programs have a very large state space and require elaborate reduction techniques to be efficiently checkable. Partial order reduction reduces the number of relevant interleavings by identifying operations that are independent of actions of other threads (for instance, local variable access).

Abstraction is used to simplify the transition system before checking it. It is common that many aspects of a program execution are not relevant for a given property to be checked. For instance, the values of amount and balance in the `transfer` example are not relevant for the existence of a deadlock. Therefore, they can be abstracted away, which reduces the state space significantly. Yet another approach to reduce the state space is to limit the search to program executions with a small number of data objects, threads, or thread preemptions. Such limits compromise soundness, but practical experience shows that many concurrency errors may be detected with a small number of threads and preemptions. For instance, detecting the deadlock in method `transfer` requires only two `Account` objects, two threads, and one preemption.

Discussion

For finite transition systems (i.e., programs with a finite number of states such as programs with an upper bound on the number of threads, the number of dynamically allocated data objects, and the recursion depth, in particular, terminating programs), model checking is sound since all possible states and traces can be checked. However, some model checkers explore only a subset of the possible states of a program execution based on heuristics that are likely to detect errors; this approach trades efficiency for soundness.

Model checking for finite transition systems is in principle complete, although abstraction may lead to spurious errors. Counterexample-guided abstraction refinement [6] is a technique to recover completeness by using the counterexample traces of spurious errors to

iteratively refine the abstraction. In contrast to deductive verification, model checking is not limited by the expressiveness of a verification methodology. Model checkers are thus capable of analyzing arbitrary programs, including for instance optimistic concurrency, benevolent data races, and interlocked operations. In practice, software model checking can be inconclusive because the state space is too large, causing the model checker to abort.

Although there is work on compositional model checking, most model checkers are not modular because it is difficult to summarize temporal properties at module boundaries.

Model checking, including abstraction, is fully automatic. Users need to provide only the property to be checked. When the checking is not feasible with the available resources, users also need to manually simplify the program or provide upper limits on the number of objects, threads, etc.

Due to the large state space of programs (in particular of parallel programs and programs that manipulate large data structures) and the non-modularity of the approach, model checking is typically too slow to be applied routinely during software development (in contrast to type checking and similar static analyses). However, for control-intensive programs with rather small data structures such as device drivers or embedded software, model checking is an efficient and very effective verification technique.

Examples

Even though most software model checkers focus on sequential programs, there are a number of tools available that check concurrency errors. JavaPathFinder [25] is an explicit state model checker for concurrent Java programs that has been applied in industry and academia. It detects data races and deadlocks as well as violations of user-written assertions and LTL properties.

MAGIC [5] checks whether a labeled transition system is a safe abstraction of a C procedure. The C procedure may invoke other procedures which are themselves specified in terms of state machines, which enables compositional checking. MAGIC uses counterexample-guided abstraction refinement to reduce the state space of the system.

BLAST [11] finds data races in C programs. It combines counterexample-guided abstraction refinement and assume-guarantee reasoning to infer a suitable model for thread contexts, which results in a low rate of false positives.

The Zing tool [2] checks for data races, deadlocks, API usage rules, and other safety properties. Its distinctive features are the use of procedure summaries to make the model checking compositional and the use of Lipton's theory of reduction [16] to prune irrelevant thread interleavings.

CHESS [17] is a tool for systematically enumerating thread interleavings in order to find errors such as data races, deadlocks, and livelocks in Win32 DLLs and managed.NET code. The analysis is complete, that is, every error found by CHESS is possible in an execution of the program. CHESS uses preemption bounding, that is, limits the number of thread preemptions per program execution to reduce the state space. Modulo this preemption bound, CHESS is sound; all remaining errors require more preemptions than the bound. Soundness is achieved by capturing all sources of non-determinism, including memory model effects.

Model checkers for message-passing concurrency include for instance MPI-Spin [20], which check Promela models of MPI programs, and ISP [26], which operates directly on the MPI/C source code. Both tools detect deadlocks and assertion violations; MPI-Spin also checks liveness properties.

Static Program Analysis

Overview

Static analyzers compute static approximations of the possible behaviors of a program and then check properties of these approximations. Common forms of static analysis include, among others, data flow analysis and type systems. Both techniques are typically described formally as sets of inference rules that allow the analyzer to infer properties of execution states, for instance, the set of possible values of a variable. When annotations are given by the programmer, these rules are also used to check the annotations. Both data flow analysis and type systems have been applied extensively to detect concurrency errors.

Static data flow analyses typically apply the lockset algorithm to detect races. They compute the set of locks each thread holds when accessing any given variable. If for each variable the locksets of all threads are not disjoint then there is mutual exclusion on the variable accesses (see entry on Race Detection Techniques for details). In the example above, the static analysis would render the field accesses safe if it can show that for all accesses to `x.balance` and `x.transactionCount`, the executing thread holds at least the lock associated with the object `x`.

Static analyses detect atomicity violations using Lipton's theory [16]. Via a classification of operations into right and left movers, an analysis can show that a given code block is atomic, that is, any execution of the block is equivalent to an execution without interference from other threads. For this purpose, the analysis must show that variable accesses are both left and right movers, which amounts to showing the absence of data races. In the example, the conditional statement does not access any shared variable and is, thus, a both-mover. The `acquire` statements are right movers, the `release` statements are left movers, and the field accesses are both-movers provided that there are no data races. Consequently, the method is atomic.

Deadlocks can be detected by computing a static lock-order graph and checking this graph to be acyclic. A deadlock analysis would complain about the `transfer` example if it cannot rule out that two threads execute `transfer` concurrently with reversed arguments.

Most data flow analyses infer the information needed to check for concurrency errors. This also involves the computation of a whole range of additional program properties, which are required for these checks. For instance, the analyses require alias information in order to decide whether two expressions refer to the same object at run time. A may-alias analysis is for instance used for sound deadlock checking (to check whether two threads *potentially* compete for the same lock), whereas a must-alias analysis is required for sound race checking (to check whether two locksets *definitely* overlap). Another common auxiliary analysis is thread-escape analysis which distinguishes thread-local variables from shared variables; this is for instance useful to suppress warnings when accesses to thread-local variables are not guarded by any lock.



Discussion

Static analyses are in principle sound. However, many existing tools sacrifice soundness in favor of efficiency. For instance, several data race analyses omit a must-alias analysis, which is necessarily flow-sensitive, and, therefore, cannot check soundly whether two locksets overlap.

Static analyses are by nature incomplete since they over-approximate the possible executions of a program. Another source of incompleteness is that static analyses – much like deductive verification – check whether programs comply with a given methodology, such as lock-based synchronization or static lock ordering. Programs that do not comply might lead to false positives.

Whether or not a static analysis is modular depends mostly on the existence of program annotations. Most data flow analyses are whole-program analyses, but there exist a number of analyses that compute procedure summaries to enable modular checking. By contrast, type systems typically require annotations, which are then checked modularly.

Aside from annotations that may be required from programmers, static analyses are fully automatic. However, they often produce a large number of false positives, which need to be inspected manually or filtered aggressively (which may compromise soundness).

Static analyses are extremely efficient. Many static analyzers have been applied to programs consisting of hundreds of thousands or even millions of lines of code. This scale is orders of magnitudes larger than for deductive verification and model checking.

In summary, static analysis tools typically favor efficiency over soundness and completeness. This choice makes them ideal for finding errors in large applications, whereas deductive verification and model checking are more appropriate when strong correctness guarantees are needed.

Examples

Warlock [21], RacerX [8], and Relay [27] use the lock-set algorithm to detect data races in C programs. RacerX also computes a static lock-order graph to detect deadlocks. It uses various heuristics to improve the analysis results, for instance, to identify benign races. Relay is a modular analysis that limits the sources of unsoundness to four very specific sources, one of them being the syntactic filtering of warnings to reduce

the number of false positives. RacerX and Relay have been applied to millions of lines of code such as the Linux kernel.

Chord [18] is a context-sensitive, flow-insensitive race checker for Java that supports three common synchronization idioms: lexically scoped lock-based synchronization, fork/join, and wait/notify. Since Chord does not perform a must-alias analysis, it is not sound. Chord has been applied to hundreds of thousands of lines of code.

rccjava [9] uses a type system to check for data races in Java programs. The annotations required from the programmer include declarations of thread-local variables and of the locks that guard accesses to a shared variable. The type system has been extended to atomicity checking [10]. rccJava scales to hundreds of thousands of lines of code.



Related Entries

- [Deadlocks](#)
- [Determinacy](#)
- [Intel Parallel Inspector](#)
- [Monitors, Axiomatic Verification of](#)
- [Race Conditions](#)
- [Race Detectors for Cilk and Cilk++ Programs](#)
- [Race Detection Techniques](#)
- [Owicki-Gries Method of Axiomatic Verification](#)

Bibliographic Notes and Further Reading

This entry focuses entirely on tools that detect concurrency errors present in programs. Two related areas are (1) tools that prevent concurrency errors during the construction of the program and (2) tools that detect concurrency errors in other artifacts.

Tools of the first kind support correctness by construction. Programs are typically constructed through a stepwise refinement of some high-level specification to executable code. Since the code is derived systematically from a formal specification, errors including concurrency errors are prevented. Interested readers may start from Abrial's book [1].

Tools of the second kind do not operate on programs but on other representations of software and hardware. Races and deadlocks may occur for instance also in workflow graphs; they can be detected by the

approaches described here, but also through Petri-net and graph algorithms [23]. Similarly, hardware can be verified not to contain concurrency errors such as deadlock [3], an area in which model checking has been applied with great success.

Acknowledgments

Thanks to Felix Klaedtke and Christoph Wintersteiger for their helpful comments on a draft of this entry.

Bibliography

1. Abrial J-R (2010) Modeling in Event-B. Cambridge University Press, Cambridge
2. Andrews T, Qadeer S, Rajamani SK, Xie Y (2004) Zing: exploiting program structure for model checking concurrent software. In: Gardner P, Yoshida N (eds) Concurrency Theory (CONCUR). Lecture Notes in Computer Science, vol 3170. Springer, Berlin, pp 1–15
3. Bryant R, Kukula J (2003) Formal methods for functional verification. In: Kuehlmann A (ed) The best of ICCAD: 20 years of excellence in computer aided design. Kluwer, Norwell, pp 3–16
4. Calcagno C, Parkinson MJ, Vafeiadis V (2007) Modular safety checking for fine-grained concurrency. In: Nielson HR, Filé G (eds) Static analysis (SAS). Lecture Notes in Computer Science, vol 4634. Springer, Berlin, pp 233–248
5. Chaki S, Clarke EM, Groce A, Jha S, Veith H (2004) Modular verification of software components in C. IEEE Trans. Softw Eng 30(6):388–402
6. Clarke EM, Grumberg O, Jha S, Lu Y, Veith H (2003) Counterexample-guided abstraction refinement for symbolic model checking. J ACM 50(5):752–794
7. Cohen E, Dahlweid M, Hillebrand M, Leinenbach D, Moskal M, Santen T, Schulte W, Tobies S (2009) VCC: a practical system for verifying concurrent C. In: Berghofer S, Nipkow T, Urban C, Wenzel M (eds) Theorem proving in higher order logics (TPHOLs 2009). Lecture Notes in Computer Science, vol 5674. Springer, Berlin, pp 23–42
8. Engler DR, Ashcraft K (2003) RacerX: effective, static detection of race conditions and deadlocks. In: Scott ML, Peterson LL (eds) Symposium on operating systems principles (SOSP). ACM, New York, pp 237–252
9. Flanagan C, Freund SN (2000) Type-based race detection for Java. In: Proceedings of the ACM conference on programming language design and implementation (PLDI). ACM, New York, pp 219–232
10. Flanagan C, Freund SN, Lifshin M, Qadeer S (2008) Types for atomicity: static checking and inference for Java. ACM Trans Program Lang Syst 30(4):1–53
11. Henzinger TA, Jhala R, Majumdar R (2004) Race checking by context inference. In: Pugh W, Chambers C (eds) Programming Language Design and Implementation (PLDI). ACM, New York, pp 1–13
12. Jacobs B, Leino KRM, Piessens F, Schulte W, Smans J (2008) A programming model for concurrent object-oriented programs. ACM Trans Program Lang Syst 31(1):1–48
13. Jacobs B, Piessens F (2008) The VeriFast program verifier. Technical Report CW-520, Department of computer science, Katholieke Universiteit Leuven
14. Jones CB (1983) Specification and design of (parallel) programs. In: Proceedings of IFIP'83. North-Holland, pp 321–332
15. Leino KRM, Müller P, Smans J (2009) Verification of concurrent programs with Chalice. In: Aldini A, Barthe G, Gorrieri R (eds) Foundations of security analysis and design V. Lecture Notes in Computer Science, vol 5705. Springer, Berlin, pp 195–222
16. Lipton RJ (1975) Reduction: a method of proving properties of parallel programs. Commun ACM 18(12):717–721
17. Musuvathi M, Qadeer S, Ball T, Basler G, Nainar PA, Neamtiu I (2008) Finding and reproducing heisenbugs in concurrent programs. In: Draves R, van Renesse R (eds) Operating systems design and implementation (OSDI). USENIX Association, pp 267–280
18. Naik M, Aiken A, Whaley J (2006) Effective static race detection for Java. In: Schwartzbach MI, Ball T (eds) Programming language design and implementation (PLDI). ACM, pp 308–319
19. O'Hearn PW (2007) Resources, concurrency, and local reasoning. Theor Comput Sci 375(1–3):271–307
20. Siegel SF (2007) Model checking nonblocking MPI programs. In: Cook B, Podelski A (eds) Verification, model checking, and abstract interpretation (VMCAI). Lecture Notes in Computer Science, vol 4349, pp 44–58
21. Sterling N (1993) Warlock – a static data race analysis tool. In: USENIX Winter, pp 97–106
22. Tofan B, Bäumler S, Schellhorn G, Reif W (2009) Verifying linearizability and lock-freedom with temporal logic. Technical report, Fakultät für Angewandte Informatik der Universität Augsburg, 2009
23. van der Aalst WMP, Hirnschall A, Verbeek HMWE (2002) An alternative way to analyze workflow graphs. In: Piddick AB, Mylopoulos J, Woo CC, Özsu MT (eds) Advanced information systems engineering (CAiSE). Lecture Notes in Computer Science, vol 2348. Springer, pp 535–552
24. Villard J, Lozes É, Calcagno C (2010) Tracking heaps that hop with Heap-Hop. In: Esparza J, Majumdar R (eds) Tools and algorithms for the construction and analysis of systems (TACAS). Lecture Notes in Computer Science, vol 6015. Springer, 275–279
25. Visser W, Havelund K, Brat GP, Park S, Lerda F (2003) Model checking programs. Autom Softw Eng 10(2):203–232
26. Vo A, Vakkalanka S, DeLisi M, Gopalakrishnan G, Kirby RM, Thakur R (2009) Formal verification of practical MPI programs. In: Principles and practice of parallel programming (PPoPP). ACM, pp 261–270
27. Young JW, Jhala R, Lerner S (2007) Relay: static race detection on millions of lines of code. In: Crnkovic I, Bertolino A (eds) European software engineering conference and foundations of software engineering (ESEC/FSE). ACM, pp 205–214

Fortran 90 and Its Successors

MICHAEL METCALF
Berlin, Germany

Definition

Fortran is a high-level programming language that is widely used in scientific programming in the physical sciences, engineering, and mathematics. The first version was developed by John Backus at IBM in the 1950s. Following an initial, rapid spread in its use, the language was standardized, first in 1966 and, after many intermediate standards, most recently in 2010. The language is a procedural, imperative, compiled language with a syntax well suited to a direct representation of mathematical formulae. Individual procedures may either be grouped into modules or compiled separately, allowing the convenient construction of very large programs and of subroutine libraries. Fortran contains features for array processing, abstract and polymorphic data types, and dynamic data structures. Compilers typically generate very efficient object code, allowing an optimal use of computing resources. Modern language provide extensive facilities for object-oriented and parallel programming. Earlier versions have relied on auxiliary standards to facilitate parallel programming.

Discussion

The Development of Fortran

Historical Fortran

Backus, at the end of 1953, began the development of the Fortran programming language (the name being an acronym derived from FORmula TRANslator), with the objective of producing a compiler that would generate efficient object code. The first version, known as FORTRAN I, contained early forms of constructs that have survived to the present day: simple and subscripted variables, the assignment statement, a DO-loop, mixed-mode arithmetic, and input/output (I/O) specifications. Many novel compiling techniques had to be developed.

Based on the experience with FORTRAN I, it was decided to introduce a new version, FORTRAN II, in 1958. The crucial differences between the two were the

introduction of subprograms, with their associated concept of shared data areas, and separate compilation.

In 1962, FORTRAN IV was released. It contained type statements, the logical IF statement, and the possibility to pass procedure names as arguments. In 1966, a standard for FORTRAN, based on FORTRAN IV, was published. This was the first programming language to achieve recognition as an American and subsequently international standard, and is now known as FORTRAN 66.

The permissiveness of the standards led to a proliferation of dialects and a new revision was published in 1978, becoming known as FORTRAN 77. It was adopted by the International Standards Organization (ISO) shortly afterwards. The new standard brought with it many new features, for instance the IF...THEN...ELSE construct, a character data type, and enhanced I/O facilities. Although only slowly adopted, this standard eventually gave Fortran a new lease of life that allowed it to maintain its position as the most widely used scientific programming language.

Modern Fortran

Fortran 90

Fortran's strength had always been in the area of numerical, scientific, engineering, and technical applications. In order that it be brought properly up to date, an American subcommittee, working as a development body for the ISO committee ISO/IEC JTC1/SC22/WG5 (or simply WG5), prepared a further standard, now known as Fortran 90. It was developed in the heyday of the supercomputer, when large-scale computations using vectorized codes were being carried out on machines like the Cray 1 and CDC Cyber 205, and the experience gained in this area was brought into the standardization process by representatives of computer hardware and software vendors, users, and academia. Thus, one of the main features of Fortran 90 was the array language, built on whole array operations and assignments, array sections, intrinsic procedures for arrays, and dynamic storage. It was designed with optimization in mind. Another was the notion of abstract data types, built on modules and module procedures, derived data types (structures), operator overloading and generic interfaces, together with pointers. Also important were new facilities for

numerical computation including a set of numeric inquiry functions, the parameterization of the intrinsic types, new control constructs (SELECT CASE and new forms of DO), internal and recursive procedures, optional and keyword arguments, improved I/O facilities, and new intrinsic procedures. The resulting language was a far more powerful tool than its predecessor and a safer and more reliable one too. Subsequent experience showed that Fortran 90 compilers detected errors far more frequently than previously, resulting in a faster development cycle. The array syntax in particular allowed compact code to be written, in itself an aid to safe programming. Fortran 90 was published by ISO in 1991.

Fortran 95

Following the publication of Fortran 90, the Fortran standards committees continued to operate, and decided on a strategy whereby a minor revision of Fortran 90 would be prepared, establishing a principle of having major and minor revisions alternating. At the same time, the High-Performance Fortran Forum (HPFF) was founded.

The HPFF was set up in an effort to define a set of extensions to Fortran such that it would be possible to write portable, single-threaded code when using parallel computers for handling problems involving large sets of data that can be represented by regular grids, allowing efficient implementations on SIMD (Single-Instruction-Multiple-Data) architectures. This version of Fortran was to be known as High Performance Fortran (HPF), and it was quickly decided, given its array features, that Fortran 90 should be its base language. Thus, the final form of HPF was of a superset of Fortran 90, the main extensions being in the form of directives. However, HPF included some new syntax too and, in order to avoid the development of divergent dialects of Fortran, it was agreed to include some of this new syntax into Fortran 95. Apart from this syntax, only a small number of other pressing but minor changes were made, and Fortran 95 was adopted as a formal standard in 1997. It is currently the most widely used version of Fortran.

Auxiliary Standards

With few exceptions, no Fortran standard up to and including Fortran 2003 included any feature intended

directly to facilitate parallel programming. Rather, this has had to be achieved through the intermediary of ad hoc industry standards, in particular HPF, MPI, OpenMP, and Posix Threads.

The HPF directives, already introduced, take the form of Fortran 90 comment lines that are recognized as directives only by an HPF processor. An example is

```
!HPF$ ALIGN WITH b :: a1, a2, a3
```

to align three conformable arrays with a fourth, thus ensuring locality of reference. Further directives allow, for instance, aligned arrays to be distributed over a set of processors.

MPI is a library specification for message passing. OpenMP supports multi-platform, shared-memory parallel programming and consists of a set of compiler directives, library routines, and environment variables that determine run-time behavior. Posix Threads is again a library specification, for multithreading.

In any event, outside Japan, HPF met with little success, whereas the use of MPI and OpenMP has become widespread.

Fortran 2003

The following language standard, Fortran 2003, was published in 2004. Its main features were: the handling of floating-point exceptions, based on the IEEE model of floating-point arithmetic; allowing allocatable arrays as structure components, dummy arguments, and function results; interoperability with C; parameterized data types; object-orientation via constructors/destructors, inheritance, and polymorphism; derived type I/O; asynchronous I/O; and procedure variables. Although all this all represented a major upgrade, especially given the introduction of object-oriented programming, there was no new development specifically related to parallel programming.

Fortran 2008

Notwithstanding the fact that Fortran 2003-conformant compilers had been very slow to appear, the standardization committees thought fit to proceed with yet another standard, Fortran 2008. In contrast to the previous one, its single most important new feature is directly related to parallel processing – the addition of coarray handling facilities. Further, the DO CONCURRENT form of loop control and the CONTIGUOUS attribute

are introduced. These are further described below. Other major new features include: sub-modules, enhanced access to data objects, enhancements to I/O and to execution control, and more intrinsic procedures, in particular for bit processing. Fortran 2008 was published in 2010.

Selected Fortran Features

Given that all the actively supported Fortran compilers on the market offer at least the whole of Fortran 95, that version is the one used as the basis of the descriptions in this section, except where otherwise stated. Only those features are described that are relevant to vectorization or parallelization. Under Fortran 95, achieving effective parallelization requires the exploitation of a combination of efficient Fortran serial execution on individual processors with, say, MPI, controlling the communication between processors.

The DO Constructs

Many iterative calculations can be carried out in the form of a DO construct. A simplified but sufficient nested form is illustrated by:

```
outer: DO
inner:   DO i = j, k, m ! from j to
           ! k in steps of m
           ! (m is optional)
           :
           IF (...) CYCLE ! take
           ! next iteration of
           ! construct 'inner'
           :
           IF (...) EXIT outer ! exit
           ! the construct 'outer'
         END DO inner
      END DO outer
```

(The Fortran language is case insensitive; however, to enhance the clarity of the code examples, Fortran keywords are distinguished by being written in upper case.) As shown, the constructs may be optionally named so that any EXIT or CYCLE statement may specify which loop is intended.

Potential data dependencies between the iterations of a DO construct can inhibit optimization, including vectorization. To alleviate this problem, Fortran 2008 introduces the DO CONCURRENT form of the DO construct, as in

```
DO CONCURRENT (i = 1:m)
  a(k + i) = a(k + i) + scale * a(n + i)
END DO
```

that asserts that there is no overlap between the range of values accessed (on the right-hand side of the assignment) and the range of values altered (on the left-hand side). The individual iterations are thus independent.

Array Handling

Array Variables

Arrays are variables in their own right; typically they are specified using the DIMENSION attribute. Every array is characterized by its type, rank (dimensionality), and shape (defined by the extents of each dimension). The lower bound of each dimension is by default 1, but arbitrary bounds can be explicitly specified. The DIMENSION keyword is optional; if omitted, the array shape must be specified after the array-variable name. The examples

```
REAL :: x(100)
INTEGER, DIMENSION(0:1000, -100:100) :: plane
```

declare two arrays: x is rank-1 and plane is rank-2. Fortran arrays are stored with elements in column-major order. Elements of arrays are referenced using subscripts, for example,

```
x(n) x(i*j)
```

Array elements are scalar. The subscripts may be any scalar integer expression.

A section is a part of an array variable, referenced using subscript ranges, and is itself an array:

x(i:j)	! rank one
plane(i:j, k:l:m)	! rank two
x(plane(7, k:l))	! vector subscript zero
x(5:4)	! zero length

Array-valued constants and variables (array constructors) are available, enclosed in (/ ... /) or, in Fortran 2003, in square brackets, []:

```
(/ 1, 2, 3, 4, 5 /)
(/ ( / 1, 2, 3 /), i = 1, 10) /
(/ (i, i = 1, limit, 2) /)
[ (0, i = 1, 1000) ]
[ (0.01*i, i = 1, 100) ]
```

They may make use of an implied-DO loop notation, as shown, and can be freely used in expressions and assignments.

A derived data type may contain array components. Given

```
TYPE point
    REAL, DIMENSION(3) :: vertex
END TYPE point
TYPE(point), DIMENSION(10) :: points
TYPE(point), DIMENSION(10, 10):: &
    points_array
```

`points(2)` is a scalar (a structure) and `points(2)%vertex` is an array component of a scalar. A reference like `points_array(n, 2)` is an element (a scalar) of type `point`. However, `points_array(n, 2)%vertex(2)` is an array of type `real`, and `points_array(n, 2)%vertex(2)` is a scalar element of it. An array element always has a subscript or subscripts qualifying at least the last name. Arrays of arrays are not allowed.

The general form of subscripts for an array section is

`[lower] : [upper] [:stride]`

where `[]` indicates an optional item. Examples for an array `x(10, 100)` are

```
x(i, 1:n)          ! part of one row
x(1:m, j)          ! part of one column
x(i, :)            ! whole row i
x(i, 1::3)          ! every third
                   ! element of row i
x(i, 100:1:-1)     ! row i in reverse
                   ! order
x((/1, 7, 3/), 10) ! vector
                   ! subscript
x(:, 1:7)          ! rank-2 section
```

A given value in an array can be referenced both as an element and as a section:

```
x(1, 1)      ! scalar (rank zero)
x(1:1, 1)    ! array section (rank one)
```

depending on the circumstances or requirements.

Operations and Assignments

As long as they are of the same shape (conformable), scalar operations and assignments are extended to arrays on an element-by-element basis. For example, given declarations of

```
REAL, DIMENSION(10, 20) :: x, y, z
REAL, DIMENSION(5)      :: a, b
```

it can be written:

```
x = y           ! whole array
z = x/y         ! assignment
                ! whole array
                ! division and
                ! assignment
z = 10.0        ! whole array
                ! assignment of
                ! scalar value
b = a + 1.3    ! whole
                ! array addition
                ! to scalar value
b = 7/a + x(1:5, 5) ! array
                    ! division, and
                    ! addition to
                    ! section
z(1:8, 5:10) = x(2:9, 5:10) + &
               z(1:8, 15:20)
                    ! array section
                    ! addition
                    ! and assignment
a(2:5) = a(1:4) ! overlapping
                  ! section
                  ! assignment
```

An array of zero size is a legitimate object, and is handled without the need for special coding.

Elemental Procedures

In an assignment like

```
y = SQRT(x)
```

an intrinsic function, `SQRT`, returns an array-valued result for an array-valued argument. It is an elemental procedure.

Elemental procedures may be called with scalar or array actual arguments. The procedure is applied to array arguments as though there were a reference for each element. In the case of a function, the shape of the result is the shape of the array arguments.

Most intrinsic functions are elemental and Fortran 95 extended this feature to non-intrinsic procedures. It is a further aid to optimization on parallel processors. An elemental procedure requires the `ELEMENTAL` attribute and must be pure (i.e., have no side effects, see below).

Array-Valued Functions and Arguments

Users can write functions that are array valued; they require an explicit interface and are usually placed in a module. This example shows such a module function, accessed by a main program:

```
MODULE show
CONTAINS
  FUNCTION func(s, t)
    REAL, DIMENSION(:) :: s, t
    ! An assumed-shape
    ! array, see below
    REAL, DIMENSION(SIZE(s)) :: func
    func = s * t**2
  END FUNCTION func
END MODULE show

PROGRAM example
USE show
REAL, DIMENSION(3)
  :: x = (/ 1., 2., 3./),           &
        y = (/ 3., 2., 1. /), s
  s = func(x, y)
  PRINT *, s
END PROGRAM example
```

In order to allow for optimization, especially on parallel and vector machines, the order of expression evaluation in executable statements is not specified. However, some potential optimizations might be lost if it is not certain that an array is in contiguous storage; for instance, if the array is an assumed-shape dummy argument array or a pointer array. In Fortran 2008, it is open to a programmer to assert that an array is contiguous by adding the CONTIGUOUS attribute to its specification, as in

```
REAL, CONTIGUOUS, DIMENSION(:, :) :: x
```

The many intrinsic functions that accept array-valued arguments should be regarded as an integral part of the array language. A brief summary of their categories appears in [Table 1](#).

Assumed-Shaped Arrays

Given an actual argument in a procedure reference, as in:

```
REAL, DIMENSION(0:10, 0:20) :: x
:
CALL calc(x)
```

Fortran 90 and Its Successors. Table 1 Classes of intrinsic procedures, with examples

Class of intrinsic array function	Examples
Numeric	abs, modulo
Mathematical	acos, log
Floating-point manipulation	exponent, scale
Vector and matrix multiply	dot_product, matmul
Array reduction	maxval, sum
Array inquiry	allocated, size
Array manipulation	cshift, eoshift
Array location	maxloc, minloc
Array construction	merge, pack
Array reshape	reshape

the corresponding dummy argument specification must define the type and rank of the array, but it may omit the shape. Thus in

```
SUBROUTINE calc(d)
REAL, DIMENSION(:, :) :: d
```

it is as if d were dimensioned (11, 21). The shape, not the bounds, is passed, meaning that the default lower bound is 1 and the default upper bound is the corresponding extent. However, any lower bound can be specified and the array maps accordingly. Assumed-shape arrays allow great flexibility in array argument passing.

Automatic Arrays

Automatic arrays are useful for defining local, temporary work space in procedures, as in

```
SUBROUTINE exchange(x, y)
  REAL, DIMENSION(:)          :: x, y
  REAL, DIMENSION(SIZE(x))   :: work
  work = x
  x = y
  y = work
END SUBROUTINE exchange
```

Here, the array work is created on entering the procedure and destroyed on leaving. The actual storage is typically maintained on a stack.

Allocatable Arrays

Fortran provides dynamic allocation of storage, commonly implemented using a heap storage mechanism.

An example, for establishing a work array for a whole program, is

```
MODULE global
  INTEGER :: n
  REAL, DIMENSION(:, :, ), ALLOCATABLE :: &
    work
END MODULE global
PROGRAM analysis
  USE global
  READ (*, *) n
  ALLOCATE(work(n, 2*n), STAT=status)
  :
  DEALLOCATE (work)
```

The work array can be propagated throughout the whole program via a USE statement in each program unit. An explicit lower bound may be specified and several entities may be allocated in one statement. Deallocation of arrays is automatic when they go out of scope.

Masked Assignment

Often, there is a need to mask an assignment. This can be done using either a WHERE statement:

```
WHERE (x /= 0.0) x = 1.0/x
```

an example that avoids division by zero by replacing only non-zero values by their reciprocals, or a WHERE construct:

```
WHERE (x /= 0.0)
  x = 1.0/x
  y = 0.0 ! all arrays assumed
        conformable
ELSEWHERE
  x = HUGE(0.)
  y = 1.0 - y
END WHERE
```

The ELSEWHERE statement may also have a mask clause, but at most one ELSEWHERE statement can be without a mask, and that must be the final one; WHERE constructs may be nested within one another.

The FORALL Statement and Construct

When a DO construct is executed, each successive iteration is performed in order and one after the other – an impediment to optimization on a parallel processor. To help in this situation, Fortran 95 introduced the

FORALL statement and construct, which can be considered as an array assignment expressed with the help of indices. An example is

```
FORALL(i = 1:n) x(i, i) = s(i)
```

where the individual assignments may be carried out in any order, and even simultaneously. In

```
FORALL(i=1:n, j=1:n, y(i,j)/=0.)
  x(j,i) = 1.0/y(i,j)
```

the assignment is subject also to a masking condition.

The FORALL construct allows several assignment statements to be executed in order, as in

```
FORALL(i = 2:n-1, j = 2:n-1)
  x(i,j) = x(i,j-1) + x(i,j+1)
  + x(i-1,j) + x(i+1,j)
  y(i,j) = x(i,j)
END FORALL
```

Assignment in a FORALL is like an array assignment: it is as if all the expressions were evaluated in any order, held in temporary storage, then all the assignments performed in any order. The first statement in a construct must fully complete before the second can begin. Procedures referenced within a FORALL must be pure (see below).

In general, the FORALL has been poorly implemented and the DO CONCURRENT is seen as a better alternative: using DO CONCURRENT requires the programmer to ensure that the iterations are independent, whereas, with FORALL, the compiler is expected to determine whether a temporary copy is needed, which is often unrealistic.

Pure Procedures

This is another Fortran 95 feature expressly for parallel computing, ensuring that execution of a procedure referenced from within a FORALL statement or construct (or, in Fortran 2008, from within a DO CONCURRENT construct) cannot have any side effects (whereby the result of one reference could cause a change to the result of another). Any such side effects in a function could impede optimization on a parallel processor – the order of execution of the assignments could affect the results. To control this situation, a PURE keyword may be added to the SUBROUTINE or FUNCTION statement – an assertion that the procedure: alters no global variable;

performs no input/output operations; has no saved variables; and, in the case of functions, alters no argument. These constraints are designed such that they can be verified by a compiler.

All the intrinsic functions are pure.

The FORALL statement and construct and the PURE keyword were adopted from HPF.

Coarrays (Fortran 2008 Only)

The objective of coarrays is to distribute over some number of processors not only *data*, as in an SIMD model, but also *work*, using the SPMD (Single-Program-Multiple-Data) model. The syntax required to implement this facility has been designed to make the smallest possible impact on the appearance of a program and to require a programmer to learn just a modest set of new rules.

Data distribution is achieved by specifying the relationship among memory images using an elegant new syntax similar to conventional Fortran. Any object *not* declared using this syntax exists independently in all the images and can be accessed only from within its own image. Objects specified *with* the syntax have the additional property that they can be accessed directly from any other image. Thus, the statement

```
REAL, DIMENSION(1024)[*] :: a, b
```

specifies two coarrays, a and b, that have the same size (1024) in each image. Execution by an image of the statement

```
a(:) = b(:, [j])
```

causes the array b from image j to be copied into its own array a, where square brackets are the notation used to access an object on another image. On a shared-memory machine, an implementation of a coarray might be as an array of a higher dimension. On a distributed-memory machine with one physical processor per image, a coarray is likely to be stored at the same address in each physical processor.

Work is distributed according to the concept of *images*, which are copies of the program that each have a separate set of data objects and a separate flow of control. The number of images is normally chosen on the command line, and its fixed value is available at execution time via an inquiry function, NUM_IMAGES. The images execute asynchronously and the execution

path in each may differ, possibly under the control of an image index to which the programmer has access (via the THIS_IMAGE function). When synchronization between two images is required, use can be made of a set of intrinsic synchronization procedures, such as SYNCH_ALL or LOCK. Thus, it is possible in particular to avoid race conditions whereby one image alters a value still required by another, or one image requires an altered value that is not yet available from another. Handling this is the responsibility of the programmer. Between synchronization points, an image has no access to the fresh state of any other image. Flushing of temporary memory, caches, or registers is normally handled implicitly by the synchronization mechanisms themselves. In this way, a compiler can safely take advantage of all code optimizations on all processors between synchronization points without compromising data integrity.

Where it might be necessary to limit execution of a code section to just one image at a time, a critical section may be defined using a CRITICAL ... END CRITICAL construct.

The codimensions of a coarray are specified in a similar way to the specifications of assumed-size arrays, and coarray sub-objects may be referenced in a similar way to sub-objects of normal arrays.

The following example shows how coarrays might be used to read values in one image and distribute them to all the others:

```
REAL :: val[*]
...
IF (THIS_IMAGE() == 1) THEN
  ! Only image 1 executes this construct
  READ(*, *) val
  DO image = 2, NUM_IMAGES()
    val[image] = val
  END DO
END IF
CALL SYNC_ALL()
  ! Execution on all images
  ! pauses here until all
  ! images reach this point
```

Coarrays can be used in most of the ways that normal arrays can. The most notable restrictions are that they cannot be automatic arrays, cannot be used for a function result, cannot have the POINTER attribute, and cannot appear in a pure or elemental procedure.

Related Entries

- [Code Generation](#)
- [Data Distribution](#)
- [Dependences](#)
- [HPF \(High Performance Fortran\)](#)
- [Loop Nest Parallelization](#)
- [MPI \(Message Passing Interface\)](#)
- [OpenMP](#)
- [Parallelization, Basic Block](#)
- [Run Time Parallelization](#)

Bibliographic Notes and Further Reading

The development of Fortran 90 was very contentious and the entire issue of the journal [2] is devoted to its various aspects.

Fortran 2003, is defined in the ISO publication [3]. Fortran 2008 was published in 2010 as [4]. Fortran 95 and Fortran 2003 are informally but completely described in [1] and [5] and, with the latest additions contained in Fortran 2008, in [6].

Coarrays have evolved from a programming model initially intended for the Cray-T3D. They were extended to Fortran 90 by Numrich and Steidel [7] and subsequently defined for Fortran 95 by Numrich and Reid [8]. Their formal definition is in the Fortran 2008 standard [4] and they are described informally also in [6].

Bibliography

1. Adams JC et al (2008) The Fortran 2003 handbook. Springer, London/New York
2. (1996) Comput Stand Interfaces 18
3. ISO/IEC 1539-1:2004. ISO, Geneva
4. ISO/IEC 1539-1:2010. ISO, Geneva
5. Metcalf M, Reid J, Cohen M (2004) Fortran 95/2003 explained. Oxford University Press, Oxford/New York
6. Metcalf M, Reid J, Cohen M (2011) Modern Fortran explained. Oxford University Press, Oxford/New York
7. Numrich RW, Steidel JL (1997) F⁻⁻: a simple parallel extension to Fortran 90. SIAM News 30(7):1–8
8. Numrich RW, Reid JK (1998) CoArray Fortran for parallel programming. ACM Fortran Forum 17:2 and Rutherford Appleton Laboratory report RAL-TR-1998-060. [ftp://ftp.numerical.rl.ac.uk/pub/reports/nrRAL98060.pdf](http://ftp.numerical.rl.ac.uk/pub/reports/nrRAL98060.pdf)

Fortran, Connection Machine

- [Connection Machine Fortran](#)

Fortress (Sun HPCS Language)

GUY L. STEELE JR.¹, ERIC ALLEN², DAVID CHASE¹, CHRISTINE FLOOD¹, VICTOR LUCHANGCO¹, JAN-WILLEM MAESSEN³, SUKYOUNG RYU⁴

¹Oracle Labs, Burlington, MA, USA

²Oracle Labs, Austin, TX, USA

³Google, Cambridge, MA, USA

⁴Korea Advanced Institute of Science and Technology, Daejeon, Korea

Definition

Fortress is a programming language designed to support parallel computing for scientific applications at scales ranging from multicore desktops and laptops to petascale supercomputers. Its generality and features for user extension also make it a useful framework for experimental design of domain-specific languages. The project originated at Sun Microsystems Laboratories in 2003 as part of their work on the DARPA HPCS (High Productivity Computing Systems) program [6] and became an open-source project in 2007.

Discussion

Design Principles

Fortress provides the programmer with a single global address space, an automatically managed heap, and automatically managed thread parallelism. Programs can also spawn threads explicitly.

The three major design principles for Fortress are:

- **Mathematical syntax:** Fortress syntax is inspired by mathematical tradition, using the full Unicode character set and two-dimensional notation where relevant. The goal is to simplify the desk-checking of code against what working scientists and mathematicians tend to write in their notebooks or on their whiteboards.
- **Parallelism by default:** Rather than grafting parallel features onto a sequential language design, Fortress

uses parallelism wherever possible and reasonable. The design intentionally makes sequential idioms a little more verbose than parallel idioms, and intentionally makes side effects a little more verbose than “pure” (side effect free) code.

- **A growable language [7]:** As few features as possible are built into the compiler, and as many language design decisions as possible are expressed at the source-code level. In many ways, Fortress is a framework for constructing domain-specific languages, and while the first such language is aimed at scientific programmers, the intent is that radically different design choices can also be supported.

Fortress is a strongly typed language. The type system is object oriented, with parameterized polymorphic types but also with type inference so that programmers frequently need not declare the types of bound variables explicitly. The objects enjoy multiple inheritance of code but not of data fields. A Fortress *trait* is much like an interface in the Java™ programming language but can contain concrete implementations of methods that can be inherited. A Fortress *object* can declare data fields and in other respects is also much like a Java class, but a Fortress object can extend (and therefore inherit from) only traits, not other objects. Methods may be overloaded; overload resolution is performed by multimethod dispatch using the dynamic types of *all* arguments. (In contrast, the Java programming language uses the dynamic type of the invocation target “before the dot” but the static types of the arguments in parentheses.)

Fortress supports and favors the use of pure, functional object-oriented programming, but also supports side effects, including assignment to variables and to fields of objects that have been declared mutable. Atomic blocks are used to manage the synchronization of side effects.

Fortress also favors divide-and-conquer approaches to parallelism (as opposed to, say, pipelining); *generators* split aggregate data structures and numeric ranges into portions that can be processed in parallel, and *reducers* combine or aggregate the results for further processing. Generators and reducers are classified according to their algebraic properties, such as associativity and commutativity; this classification is expressed through,

and enforced by, specific traits defined by a standard library, enabling a generator to choose from a variety of implementation strategies.

Salient Features

Syntax

Fortress syntax is designed to resemble standard mathematical notation as much as possible while integrating programming-language notions of variable binding, assignment, and control structures. All mathematical operator symbols in the Unicode character set [8], such as $\odot \oplus \otimes \leq \geq \cap \cup \sqcap \sqcup \subseteq \subseteq \prec \succ \sum \prod$, may be defined and overloaded as user-defined infix, prefix, and postfix operators; a large number of bracketing symbols such as $\{ \} \langle \rangle \lceil \rceil \lfloor \rfloor \ll \gg$ are likewise available for library or user (re-)definition. Relational operators may be chained, as in $1 \leq n \leq 10$; the compiler treats it as if it had been written $(1 \leq n) \wedge (n \leq 10)$ but arranges to evaluate the expression n just once. Simple juxtaposition is itself a binary operator that can be defined and overloaded by Fortress source code; the standard library defines juxtaposition to perform function application (as in $f(x,y)$ or $print x$), numerical multiplication (as in $3n$ and $5x^2y$), and string concatenation (as in “My name is” $myName$ “.”).

Unicode characters may be used directly or represented in ASCII through a Wiki-inspired syntax that is reasonably readable in itself. Fortress also provides concise syntax to mimic mathematical notation: semicolons and parentheses may often be omitted, and curly braces $\{ \}$ denote sets (rather than blocks of statements). The notation is whitespace sensitive but not indentation sensitive. Fortress is expression oriented, and it favors the use of immutable bindings over mutable variables, requiring a slightly more verbose syntax for declaration of mutable variables and for assignment.

The left-hand side of Fig. 1 shows a sample Fortress function that performs a matrix/vector “conjugate gradient” calculation (based on the NAS “CG” conjugate gradient benchmark for parallel algorithms) [4]. The function *conjGrad* takes a matrix and a vector, performs a number of calculations, and returns two values: a result vector and a measure of the numerical error. The right-hand side of Fig. 1 presents the original

<pre> conjGrad[E extends Number, nat N, V extends Vector[E,N], M extends Matrix[E,N,N]] (A : M, x : V) : (V,E) = do z : V := 0 r : V := x ρ : E := r·r p : V := r for j ← seq(1 # 25) do q = A p α = ρ/(p·q) z += α p ρ₀ = ρ r -= α q ρ := r·r β = ρ/ρ₀ p := r + β p end (z, \ x - A z\) end </pre>	$ \begin{aligned} z &= 0 \\ r &= x \\ \rho &= r^T r \\ p &= r \\ \text{DO } i &= 1, 25 \\ q &= A p \\ \alpha &= \rho / (p^T q) \\ z &= z + \alpha p \\ \rho_0 &= \rho \\ r &= r - \alpha q \\ \rho &= r^T r \\ \beta &= \rho / \rho_0 \\ p &= r + \beta p \\ \text{ENDDO} \\ \text{compute residual norm explicitly:} \\ \ r\ &= \ x - A z\ \end{aligned} $
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Fortress (Sun HPCS Language). Fig. 1 Sample Fortress code for a conjugate gradient calculation (left) and the original NAS CG pseudocode specification (right)

specification of the conjugate gradient calculation to show how the Fortress program resembles the problem specification. The principal differences are that the Fortress code provides type declarations – including a generic (type-parametric) function header – and distinguishes between binding and assignment of variables (within the loop body, q and α and ρ_0 are read-only variables that are locally bound using $=$ syntax, whereas z and r and ρ and p are mutable variables that are updated by the assignment operator $:=$ and the compound assignment operators $+=$ and $-=$).

Fortress is designed to grow over time to accommodate the changing needs of its users via a syntactic abstraction mechanism [3] that, in effect, allows source-code declarations to augment the language grammar by specifying rewriting rules for new language constructs. Parsing of new constructs is done alongside parsing of primitive constructs, allowing programmers to detect syntax errors in use sites of new constructs early. Programs in domain-specific languages can be embedded in Fortress programs and parsed as part of their host programs. Moreover, the definition of many constructs

that are traditionally defined as core language primitives (such as `for` loops) can be moved into Fortress' own libraries, thereby reducing the size of the core language.

Type System

Fortress has a rich type system designed to give programmers the expressive power they need to write libraries that provide features often built into other languages. The Fortress type system is a trait-based, object-oriented system that supports parametrically polymorphic named types (also known as *generic traits*) organized as a multiple-inheritance hierarchy. Fortress code is statically type-checked: it is a static (“compile-time”) error if the type of an expression is not a subtype of the type required by its context.

Objects may be declared to extend traits, from which they can inherit method declarations and definitions. If an object inherits an abstract method declaration, then it has the obligation to provide (or inherit) a concrete implementation of that method. A trait may also extend one or more other traits. Extension creates a subtype relationship; if type A extends type B , then every

object belonging to type A (that is, every *instance* of A) necessarily also belongs to type B .

In addition to subtyping, Fortress allows the declaration of *exclusion* and *comprises* relationships among types. If type A excludes type B , then no value can be an instance of both A and B . If a type T comprises a set of types $\{U_1, U_2, \dots, U_n\}$, then every instance of T is an instance of some U_i (possibly more than one). This allows a more flexible description of important relationships among types than simply the ability to declare that a class is *final*.

Types, values, and expressions Fortress provides several kinds of types, including *trait types*, *tuple types*, *arrow types*. At the top of the type hierarchy is the type *Any*, which comprises three types, *Object*, *Tuple*, and $()$ (pronounced “void”); at the bottom of the hierarchy is the type *BottomType*. Every expression in Fortress has a *static type*. Every value in Fortress is either an *object*, a *tuple*, or the value $()$, and has a (*run-time*) *tag*, or *ilk*, which is a “leaf” in the type hierarchy: the only strict subtype of a value’s tag is *BottomType*. Fortress guarantees that when an expression is evaluated, the *ilk* of the resulting value is a subtype of the type of the expression.

Traits Like an interface in the Java programming language, a *trait* is named program construct that declares a set of *methods*, and which may *extend* one or more other traits, inheriting their methods. However, unlike Java interfaces, whose methods are always abstract (i.e., they do not provide bodies), the methods of Fortress traits may be either *abstract* or *concrete* (having bodies). Like functions, methods may be *overloaded*, and a call to such a method is resolved using the run-time tags of its arguments.

Objects Objects form the leaves of the trait hierarchy: no trait may extend an object trait type. Thus object declarations are analogous to `final` classes in the Java programming language. In addition to methods, *objects* may have *fields*, in which state may be stored.

Numbers and booleans and characters are all objects. This does not imply that they are always heap-allocated. Objects declared with the `value` keyword have no object identity, must have only immutable fields, and may be freely copied by the implementation.

Tuples A tuple is written as a comma-separated series of expressions within parentheses, for example, $(a, 3, b + 7)$; they are convenient for passing multiple arguments and for returning multiple results. Tuples are first-class values but are not objects. Tuple types are covariant: if type A extends type B , and C extends D , and E extends F , then the tuple type (A, C, E) extends the tuple type (B, D, F) . The type of a variable, of a field, or of array elements may be a tuple type. Tuples of variables may appear on the left-hand side of binding and assignment constructs.

Functions and Methods Functions are also first-class values, and furthermore, they are objects. However, their types are not traits, but *arrow types*, constructed from other types using the arrow combinator \rightarrow . A function that is declared to return an instance of type B whenever it is passed an instance of type A as an argument is an instance of the arrow type $A \rightarrow B$. Because a function can be overloaded, having multiple definitions with different parameter types, it can be an instance of several different arrow types (Fig. 2). Because arrow types are covariant in their return types and contravariant in their parameter types, every arrow type is a subtype of $\text{BottomType} \rightarrow \text{Any}$ and a supertype of $\text{Any} \rightarrow \text{BottomType}$.

Fortress provides two kinds of methods that may be declared in objects and traits: *dotted methods*, which are similar to methods in other object-oriented languages and are invoked using the syntax `target.methodName(arguments)`, and *functional methods*, which are overloadings of global function names and therefore are invoked using the syntax `methodName(arguments)`, but one of the arguments is treated as the `target` just as for a dotted method (Fig. 3).

Ignoring the possibility of automatic coercion, a function declaration is applicable to a call to that function if its parameter type is a supertype of (possibly equal to) the run-time tag of the argument, and it is more specific than another declaration if its parameter type is a strict subtype of the parameter type of the other. (If a declaration of a function or method takes more than one argument, then its “parameter type” (singular) is considered to be a tuple type.)

The Java programming language allows any set of overloaded method declarations, resolves overloading based on the static types of the arguments and the

```
(* The function pad is an instance of the arrow type String → String and also of the arrow type  $\mathbb{Z}32 \rightarrow \mathbb{Z}64$ ,  
neither of which is a subtype of the other. *)  
pad(x: String) : String = " " x  
pad(y:  $\mathbb{Z}32$ ) :  $\mathbb{Z}64$  = y  
(* Arrow types can include the types of exceptions that can be thrown. The function pad' in  
an instance of the arrow type String → String throws EmptyString and also of the arrow type  
 $\mathbb{Z}32 \rightarrow \mathbb{Z}64$  throws ZeroNumber. *)  
pad'(x: String) : String throws EmptyString =  
  if x = "" then throw EmptyString else " " x " " end  
pad'(y:  $\mathbb{Z}32$ ) :  $\mathbb{Z}64$  throws ZeroNumber =  
  if y = 0 then throw ZeroNumber else y end
```

Fortress (Sun HPCS Language). Fig. 2 Two examples of a function that belongs to more than one arrow type

```
(* This example object has one field, v, and four method definitions. The method add is a conventional  
"dotted method"; the method name maximum is overloaded with three functional method definitions.  
The parameter self indicates that a method is a functional method that is invoked using functional  
rather than dot syntax. *)  
  
object Example(v:  $\mathbb{Z}32$ )  
  add(n:  $\mathbb{Z}32$ ) = n + v                                ④ Dotted method (no self parameter)  
  maximum(self, n:  $\mathbb{Z}32$ ) =  
    if n > v then n else v end  
  maximum(n:  $\mathbb{Z}32$ , self) =  
    maximum(self, n)  
  maximum(self, other: Example) =  
    maximum(self, other.v)  
end  
  
do  
  o = Example(5)  
  p = Example(6)  
  ④ Examples of method invocation  
  println o.add(7)          ④ Prints "12"  
  println p.add(7)          ④ Prints "13"  
  println maximum(o, 3)      ④ Prints "5"  
  println maximum(7, o)      ④ Prints "7"  
  println maximum(o, p)      ④ Prints "6"  
end
```

Fortress (Sun HPCS Language). Fig. 3 Example definition of a dynamically parameterized object

dynamic type of the target, and requires this resolution be unambiguous for calls that actually appear in the program (i.e., ambiguity is separately checked at every call site). In contrast, when a call to an overloaded Fortress function or method is executed, the most specific applicable declaration is selected based on the tags of the arguments passed to the function; that is, Fortress performs full multimethod dynamic dispatch on the target and all arguments. Furthermore, Fortress imposes rules on the set of overloaded declarations so that dispatch is always unambiguous (i.e., ambiguity is checked for each set of declarations and no separate check is needed at each call site). In particular, any pair of declarations of overloaded functions or method must satisfy one of three conditions:

Subtype rule: The parameter type of one declaration is a strict subtype of the parameter type of the other, and the return type of the first declaration is a subtype of (possibly the same as) the return type of the second declaration.

Exclusion rule: The parameter types of the two declarations exclude each other.

Meet rule: The parameter types of the two declarations are incomparable (there is neither a subtype nor an exclusion relationship between them) and there is another declaration of that function whose parameter type “covers” the intersection of the parameter types of the two incomparable declarations.

If all pairs of declarations satisfy one of these conditions, there can be no ambiguity for any particular call to that function or method: in the first case, one declaration is more specific than the other; in the second case, there is no call to which both declarations are applicable; and in the third case, whenever both declarations are applicable, there is another declaration that is more specific than both that is also applicable, so neither of the two incomparable declarations can be the most specific applicable declaration. These rules are how the design of Fortress solves the problem of multiple inheritance of methods.

Access to fields of objects is abstract: the field access syntax `x.fieldName` actually invokes a `getter` method that takes no arguments but returns a value. Declaring a field by default provides a trivial `getter` method that simply fetches the contents of the field, but this method can be suppressed or displaced by

an explicit `getter` declaration. Similarly, the assignment operator `:=` actually invokes a `setter` method, and the declaration of a mutable field by default provides a trivial `setter` method. In this manner, the behavior of field access and assignment is completely programmable.

Parametric polymorphism (or Generic types and functions) Fortress supports *generic traits* (and objects), as well as generic functions (and methods). All these entities can have static parameters of various kinds: type parameters (for which any type can serve as an actual argument), `nat` and `int` and `bool` parameters (for which static arithmetic and Boolean expressions can serve as actual arguments), `unit` and `dim` parameters (for which declared physical dimensions and units can serve as actual arguments), and `opr` parameters (for which operator symbols can serve as actual arguments).

Type parameters may be given one or more bounds, thereby requiring that they be instantiated only with subtypes of all the bounds. If no bound is given, it is implicitly assumed to be bound by `Object`. Note that `Object` is not the root of the Fortress type hierarchy; a type parameter that may be instantiated with a tuple type, for example, must be given an explicit bound (Fig. 4). This unusual default for the bounds of type parameters helps to reduce confusion when functions and methods are overloaded with different numbers of parameters.

The Java programming language has a system of generic types that *erases* type parameter information at run time. In contrast, Fortress types are *not* erased, and it is possible to perform the equivalent of an `instanceof` check to distinguish a list of strings from a list of booleans. Thus, there is no significant disparity between the static type system and the run-time tags associated with Fortress values: tags are simply a special class of types corresponding to leaves in the type hierarchy.

In addition to standard parametric polymorphism, Fortress provides `where` clauses, in which *hidden* type variables can be declared and used within a trait (or object) or function declaration; they are called “hidden” because they are not static parameters of the trait or function – rather, they are universally quantified within their declared bounds.

```
(* The trait LinkedList can contain elements of type T. Because the declared bound on the type T is Any, the
actual element type may be a tuple type. Because of the comprises clause, only instances of the object
types Cons[T] and Empty can be instances of LinkedList[T]. *)
```

```
trait LinkedList[T extends Any] comprises { Cons[T], Empty } end
```

```
(* A Cons instance has two fields, first and rest. This object declaration implicitly declares a constructor
function called Cons that takes two arguments and creates a new instance with its two fields initialized to
the given arguments. *)
```

```
object Cons[T extends Any](first : T, rest : List[T]) extends LinkedList[T] end
```

```
@ The single object Empty belongs to all LinkedList types.
```

```
object Empty extends LinkedList[T] where { T extends Any } end
```

Fortress (Sun HPCS Language). Fig. 4 Example definitions of a statically parameterized trait and two objects that implement it

Having to write out all static type arguments and complete type signatures for all functions and methods, especially local “helper” functions, can be tedious, and often results in code that is difficult to read and maintain. To mitigate these problems, Fortress (like ML and Haskell) has a type-inference mechanism that allows many types and static parameters to be elided in practice.

Coercion A trait T may contain one or more *coercion* declarations; such a declaration specifies a computation that, when given any instance of another type U , will return a corresponding instance of type T . Such declarations should be used with care, only in situations where it is desirable that *every* instance of type U also be regarded as, in effect, an instance of type T in every computational context whatsoever. Typically this is used to convert among specialized numerical representations, and in particular to convert numeric literals to other numeric types. (Fortress is unusual in that the type of a literal such as 43 is not any of the standard computational integer types such as \mathbb{Z} or $\mathbb{Z}32$ or $\mathbb{Z}64$, but a special numeric-literal trait.)

Coercion comes into play during multimethod dispatch: if there is *no* applicable function or method declaration for a given call, then the net is cast wider by considering declarations that would be applicable if the arguments were automatically coerced to some other type; if there is a unique most specific applicable

declaration in this larger set, then that is the one that will be invoked. To help prevent surprises, coercions in Fortress are *not* transitive.

Parallelism

Many constructs in the Fortress language give rise to multiple implicit *tasks* to execute subexpressions, such as the components of a tuple, blocks of the do-also construct, function arguments, operands of binary operators, and the target and arguments of a method call (Fig. 5). This is *fork-join* parallelism: all tasks for a construct execute to completion before execution of the construct itself can complete. Tasks are implicitly scheduled and provide no fairness guarantees; while tasks *may* execute concurrently, a Fortress compiler or run-time may choose to schedule tasks for sequential execution instead. This allows flexible allocation of hardware resources.

Comprehensions, reduction expressions, and for loops are all syntactic sugar for uses of generators and reducers. A *generator* is an object capable of supplying some number of separate values to a reducer through one or more library-defined protocols; a *reducer* is an object that can accept values from a generator and produce a combined or aggregated result.

For example, the nested for loop in Fig. 6 contains two generator expressions $0 : 9$ and $(i + 1) : 9$ representing parallel counted ranges; the loop desugars into a nested pair of calls to their respective *loop* methods.

(* Five examples of situations in which expressions such as $f(x)$ and $g(y)$ and $h(z)$ may be executed in parallel as concurrent tasks.
*)

$(a, b, c) = (f(x), g(y), h(z))$	⊗ Elements of a tuple
do	⊗ Separate blocks of a do -also construct
$f(x)$	
also do	
$g(y)$	
also do	
$h(z)$	
end	
$p(f(x), g(y), h(z))$	⊗ Arguments of a function call
$f(x) + g(y)$	⊗ Operands of a binary operator
$f(x).m(g(y), h(z))$	⊗ Target and arguments of a method call

Fortress (Sun HPCS Language). Fig. 5 Implicit parallelism in Fortress

The actual parallelism is provided by using implicit tasks in the implementation of the *loop* method, generally using a divide-and-conquer coding style. Most Fortress data structures are generators and support structure-respecting parallel traversal. It is a library convention that generators are implicitly parallel by default; the *seq(self)* method conventionally constructs a sequential version of a generator. Definitions of BIG operators typically define the behavior of reductions (Fig. 7) and comprehensions (Fig. 8) in terms of an underlying monoid.

Note that while generators run computations in parallel, there is still a well-defined spatial order to the results produced by a generator (Fig. 9).

Fortress also provides explicit parallelism using the spawn construct (Fig. 10). Spawns threads are scheduled fairly with respect to one another, and implicit tasks of separately spawned threads are scheduled fairly with respect to one another.

The programmer is responsible for wrapping concurrent accesses to shared data in an atomic block. The semantics of an atomic block is that either all the reads and writes to memory happen simultaneously, or none of them do; in effect, all the memory reads and memory writes of an atomic block appear to all other tasks to occur as a single atomic memory transaction. Such transactions are in fact *strongly atomic* (transactional and non-transactional accesses can coexist).

Implicit parallelism may be exploited inside of a transaction. Inner transactions may be retried independently of the outer transaction (“closed nesting”). Contention may be managed by the run-time system, or at the Fortress language level using the *tryAtomic* construct.

The current implementation of Fortress uses an automatic dynamic load-balancing algorithm (based on work done at MIT as part of the Cilk project) that uses work-stealing to move pending tasks from one processor to another.

Programming in the Large

Fortress supports a number of features for programming in the large.

Components and APIs Fortress source code is encapsulated in *components*, which contain definitions of program elements. Each component imports a collection of *APIs* (Application Programming Interfaces) that declare the program elements used by that component. Each component also exports a collection of APIs. If a component *C* exports an API *A*, then *C* must include a definition for every element declared in *A*. Components can be linked together into larger components. If a component *C* exports an API *A* that is imported by a component *D*, and *C* and *D* are linked, then references in *D* to the declarations in *A* are resolved to the corresponding definitions in *C*.

```

(* This for loop controls two index variables in nested fashion. The : operator, given two integers,
produces a CountedRange object that can serve as a generator of integer index values. *)  

for i ← 0 : 9, j ← (i + 1) : 9 do  

  ai,j := 3i - 2j  

end  

(* The for loop is desugared into nested calls to higher-order methods that take functions as arguments.  

These functions bind the index variables. The fn syntax is analogous to Lisp's lambda expression syntax  

for anonymous functions. *)  

(0 : 9).loop(fn (i) =>  

  ((i + 1) : 9).loop(fn (j) =>  

    do ai,j := 3i - 2j end))  

(* The loop method of the CountedRange generator object uses a recursive divide-and-conquer strategy that  

exploits implicit task parallelism (do-also) to permit concurrent execution of loop iterations. *)  

object CountedRange(lo: Z32, hi: Z32) extends Generator[Z32]  

  ...  

  loop(body: Z32 →()): () =  

    if lo = hi then body(lo)  

    elif lo < hi then  

      mid = ⌊ (lo + hi) / 2 ⌋  

      do  

        CountedRange(lo, mid).loop(body)  

      also do  

        CountedRange(mid + 1, hi).loop(body)  

      end  

    end  

  ...  

  seq(self) = ...  

end

```

Fortress (Sun HPCS Language). Fig. 6 Implementation of for loops through desugaring

Contracts Fortress functions and methods can be annotated with *contracts* that declare preconditions and post-conditions (Fig. 11).

Test code, properties, and automated unit testing Fortress includes support for defining tests inline in programs alongside the data and functionality they test. Similar to tests are *property declarations*, which document conditions that a program is expected to obey. The parameters in a property declaration denote the types of values over which the property is expected to hold.

When test data is provided with a property, the property is checked against all values in the test data when the program tests are run.

One use of the design-by-contract features is to document and enforce algebraic relationships that are necessary for correct parallel execution of generators and reducers (Fig. 12).

Innovations

- Fortress introduced `comprises` and `excludes` clauses that allow the type checker to deduce that two types are disjoint; this matters for certain kinds

- ⊗ Summation is one kind of reduction expression.

$$\sum_{\substack{i \leftarrow 0 : 9 \\ j \leftarrow (i+1) : 9}} a_{i,j}$$

- ⊗ The summation is desugared into nested calls to higher-order methods.

```
 $\sum((0 : 9).nest(\text{fn } (i) \Rightarrow ((i + 1) : 9).map(\text{fn } (j) \Rightarrow a_{i,j})))$ 
```

- (*) The unary \sum operator takes a generator and gives its *reduce* method an object that implements the standard reduction protocol using addition. The *reduce* method, like the *loop* method, provides implicit parallelism. This \sum operator is generic; it can be used with any data type T that implements the Additive trait, meaning that type T implements a binary $+$ operator and can supply an identity value named *zero*. *)

```
opr  $\sum[[T \text{ extends Additive}[T]]](g : \text{Generator}[T]) : T =$   
 $g.\text{reduce}(\text{SumReduction}[T])$ 
```

```
object SumReduction[[T extends Additive[T]]] extends Reduction[T, T]  
  empty() : T = zero          ⊗ Additive[T] means type T has a zero value  
  join(a : T, b : T) : T = a + b    ⊗ that is the identity for the + operator on type T.  
  singleton(a : T) : T = a        ⊗ The sum of one number is the number itself.
```

end

Fortress (Sun HPCS Language). Fig. 7 Implementation of reduction operations through desugaring

- ⊗ List comprehension $\langle \dots | \dots \rangle$ is one kind of comprehension expression.

- ⊗ It produces an ordered list of values.

$$\langle a_{i,j} \mid i \leftarrow 0 : 9, j \leftarrow (i+1) : 9 \rangle$$

- (*) Other kinds of comprehension brackets construct arrays, sets, or multisets. They are desugared in much the same manner as reduction expressions. *)

- (*) The meaning of a comprehension is defined by library code in the same way as for a reduction operator. *)

```
opr BIG  $\langle [T] g : \text{Generator}[T] \rangle : \text{List}[T] =$   
 $g.\text{reduce}(\text{ListReduction}[T])$ 
```

- (*) List aggregation is essentially reduction of singleton lists using the list concatenation operator $\|$. The library definitions of the $\|$ operator and the (noncomprehension) list constructors $\langle \rangle$ and $\langle x \rangle$ are not shown here. *)

```
object ListReduction[T] extends Reduction[T, List[T]]  
  empty() : List[T] =  $\langle \rangle$           ⊗ Produce an empty list.  
  join(a : List[T], b : List[T]) : List[T] = a  $\|$  b    ⊗ Concatenate two lists.  
  singleton(a : T) : List[T] =  $\langle a \rangle$         ⊗ Produce a singleton list.
```

end

Fortress (Sun HPCS Language). Fig. 8 Implementation of comprehension expressions through desugaring

of code optimization. Such declarations can also improve code maintainability.

- Lexical syntax is inspired in part by Wiki notation: the visual appearance of “rendered” code depends

(* The squares of the nine values 1, 2, 3, 4, 5, 6, 7, 8, 9 may be computed concurrently or in any temporal order, but the *logical* or *spatial* order of the results within ordered list will always be {1, 4, 9, 16, 25, 36, 49, 64, 81}. *)

$\langle k^2 \mid k \leftarrow 1 : 9 \rangle$

Fortress (Sun HPCS Language). Fig. 9 The distinction between logical order and temporal order

on whether an identifier is lowercase, uppercase, or mixed case, and whether underscore characters are used in particular ways. Many single-character operators and brackets can be represented by multicharacter ASCII sequences, but there is no single “escape” character (such as backslash) that introduces such sequences. As a rule, if an operator has a standard name in \TeX or \LaTeX , then that same name converted to uppercase, omitting the backslash, is a Fortress name for that operator. See Fig. 13. Text in comments uses a markup syntax based on Wiki Creole 1.0 [9]. See Fig. 14.

- Most programming languages handle operator precedence by imposing a total order on operators (sometimes by mapping them to specific integer values). Fortress uses only precedence rules that would

do

- ⊗ The `spawn` construct forks a fair concurrent thread
- ⊗ and immediately returns a handle object for the thread.

$firstThread = \text{spawn } f(x)$
 $secondThread = \text{spawn } g(y)$

- ⊗ Other computations here will run concurrently with
- ⊗ the spawned threads. The `val` method waits for a
- ⊗ spawned thread to complete and then produces whatever
- ⊗ value or exception the thread may have produced.

$firstThread.val + secondThread.val$
end

Fortress (Sun HPCS Language). Fig. 10 Explicit spawning of fair concurrent threads

(* This definition of `factorial` has a contract that requires the argument to be nonnegative. The contract also guarantees that the result will be nonnegative. *)

$\text{factorial}(n) \text{ requires } \{ n \geq 0 \} \text{ ensures } \{ outcome \geq 0 \} =$
 $\quad \text{if } n = 0 \text{ then } 1 \text{ else } n \text{factorial}(n - 1) \text{ end}$

(* This test-only code (indicated by the keyword `test`) calls the `factorial` function for all values from 0 to 100, and checks that the result for each is not less than the argument. *)

$\text{test factorialNotLessThanInput } [x \leftarrow 0 : 100] = (x \leq \text{factorial}(x))$

(* This property declaration requires that the result of `factorial` be not less than the argument for *every* argument of type \mathbb{Z} . Test data for spot-checking a property may be provided separately. *)

$\text{property factorialNeverLessThanInput} = \forall (x : \mathbb{Z}) (x \leq \text{factorial}(x))$

Fortress (Sun HPCS Language). Fig. 11 Examples of contracts, test code, and declared properties

(* This trait imposes upon any trait T that extends $\text{BinaryOperator}[\![T, \odot]\!]$ the requirement to provide a concrete implementation of the specified binary operator \odot . Note that both T and \odot are parameter names, for which an actual type and actual operator are substituted by an invocation such as $\text{BinaryOperator}[\![\mathbb{Q}, +]\!]$ or $\text{BinaryOperator}[\![\mathbb{Z}32, \text{MAX}]\!]$. *)

```
trait BinaryOperator[\!T, opr \odot\!] comprises T
  opr \odot(self, other : T) : T
end
```

⊕ An associative operator is a binary operator with the associative property.

```
trait Associative[\!T, opr \odot\!] comprises T
  extends { \text{BinaryOperator}[\!T, \odot\!], \text{EquivalenceRelation}[\!T, =\!] }
  property \forall(A; T, b: T, c: T)((a \odot b) \odot c) = (a \odot (b \odot c))
end
```

⊖ A monoid is a type with an operator that is associative and has an identity.

```
trait Monoid[\!T, opr \odot\!] comprises T
  extends { \text{Associative}[\!T, \odot\!], \text{HasIdentity}[\!T, \odot\!] }
end
```

Fortress (Sun HPCS Language). Fig. 12 Use of declared properties to enforce algebraic relationships

be familiar from high-school or college mathematics courses. A consequence is that operator precedence in Fortress is not transitive (Fig. 15).

- The “meet rule” eliminates the need for arbitrary “tie-break” or “ordering” rules found in other multiple-inheritance type systems.
- Fortress addresses the “self-type problem” common to many type systems with inheritance by using generic traits that each have a static parameter that is a type that must extend the generic trait; the `comprises` clause is used to enforce this requirement. The parameter name then serves as a self-type adequate to solve the “binary method problem.”
- The Fortress libraries use the self-type idiom to encode and enforce (through use of design-by-contract features) a hierarchy of algebraic properties of data types and operators ranging from commutativity and associativity all the way up to partial and total orders, monoids and groups, and Boolean algebras. Type-checking ensures that, for example, a comparison operator given to a `sort` method actually implements a correct ordering criterion; overloaded method dispatch allows the `sort` method to use different algorithms depending on whether the order is partial or total.

- Fortress supports units and dimensions with a relatively small amount of built-in mechanism. All the Fortress compiler knows is that units and dimensions may be declared (Fig. 16), that they form a free Abelian group under multiplication, that these abstract values may be used as static arguments to generic types, and that such a generic type may optionally be declared to *absorb* units. That you can multiply apples and oranges, that apples may be added only to apples, and whether you can exclusive-or apples and apples, are all at the discretion of the library programmer. Dimensioned types can ensure that data is correctly scaled (Fig. 17).
- The Maybe type (instances are either `Just x` or `Nothing`), borrowed from Haskell, can be used as a generator of at most one item. A syntactic convenience allows it to be used in an `if` statement so as to bind a variable to the item in just the `then` clause (Fig. 18).
- The Java programming language allows a method of the superclass to be called using the *super* keyword. A language with multiple inheritance requires a more general feature. Using a type assumption expression `x asif T` as an argument causes method dispatch to use the specified static

ASCII	rendered	
<code>z</code>	z	normal variable (italic)
<code>z_</code>	z	roman
<code>_z</code>	\mathbf{z}	bold
<code>_z_</code>	\mathbf{z}	bold italic
<code>zz</code>	\mathbb{Z}	blackboard
<code>zz_</code>	\mathcal{Z}	calligraphic
<code>_zz</code>	Z	sans serif
<code>z_bar</code>	\check{z}	math accent
<code>z_hat</code>	\hat{z}	math accent
<code>z_dot</code>	\dot{z}	math accent
<code>z'</code>	z'	math accent
<code>_z_vec</code>	\vec{z}	math accent
<code>z_max</code>	z_{\max}	roman subscript
<code>z13</code>	z_{13}	numeric subscript
<code>_z13_bar'</code>	\check{z}'_{13}	combination of features
<code>and</code>	and	normal variable (italic)
<code>And</code>	And	type name (roman)
<code>AND</code>	\wedge	operator name
<code>OPLUS</code>	\oplus	operator name as in TeX
<code>SQCAP</code>	\sqcap	operator name as in TeX
<code><=</code>	\leq	less than or equal to
<code>>=</code>	\geq	greater than or equal to
<code><-</code>	\leftarrow	left arrow
<code>[\ \]</code>	$\llbracket \ \rrbracket$	white brackets
<code>< ></code>	$\langle \ \rangle$	angle brackets

Fortress (Sun HPCS Language). Fig. 13 Examples of rendering ASCII identifiers as mathematical symbols

This ASCII Fortress source code includes Wiki markup in the comment:

```
(* Compute the logical exclusive OR of two Boolean values.
=Arguments
; 'self' :: a first Boolean value
; 'other' :: a second Boolean value
>Returns
> The exclusive OR of the two arguments.
This is 'true' if **either** argument is 'true',
but not if **both** arguments are 'true'.

> |      ||||| 'other' ||
|      |||||-----|||
|      'OPLUS' ||||| false | true ||
| ====== ====== ====== ====== ||
| 'self' || false ||| false | true ||
|         || true ||| true | false ||
| ----- ----- ----- ||
*)

opr OPLUS(self, other: Boolean): Boolean
```

and it is rendered as follows:

(* Compute the logical exclusive OR of two Boolean values.

Arguments

self a first Boolean value

other a second Boolean value

Returns

The exclusive OR of the two arguments. This is *true* if **either** argument is *true*, but not if **both** arguments are *true*.

		<i>other</i>	
		false	true
\oplus	false	false	true
	true	true	false

*)

opr \oplus (*self,other*: Boolean): Boolean

Fortress (Sun HPCS Language). Fig. 14 Example of rendering Wiki-style markup in Fortress comments

type *T* for the argument expression *x* during method lookup. This general mechanism solves the multiple supertype problem as a special case (Fig. 19).

- Fortress provides an object-oriented implicit coercion feature. It differs from a conventional method such as *toString* in that it is declared in the type converted *to* rather than the type converted *from*, and

$a + b > c + d$	⊗ Correct: + has higher precedence than >
$p > q \wedge r > s$	⊗ Correct: > has higher precedence than \wedge
$w + x \wedge y + z$	⊗ Incorrect: no defined precedence between + and \wedge
$(w + x) \wedge (y + z)$	⊗ Correct: parentheses specify desired grouping
$w + (x \wedge y) + z$	⊗ Correct: parentheses specify desired grouping

Fortress (Sun HPCS Language). Fig. 15 Operator precedence in Fortress is not transitive

⊗ Declarations of several physical dimensions and their default units

```
dim Mass default kilogram
dim Length default meter
dim Time default second
dim ElectricCurrent default ampere
```

⊗ Declarations of derived physical dimensions

```
dim Velocity = Length/Time
dim Acceleration = Length/Time2
dim Force = Mass·Acceleration
dim Energy = Force·Length
```

⊗ Declarations of basic units and their abbreviations

```
unit kilogram kg : Mass
unit meter m : Length
unit second s : Time
unit ampere A : ElectricCurrent
```

⊗ Declarations of derived units

```
unit newton n : Force = meter·kilogram/second2
unit joule J : Energy = newton meter
```

⊗ Declarations of derived units that require scaling factors

```
unit gram g : Mass = 10-3 kilogram
unit kilometer km : Length = 103 meter
unit centimeter cm : Length = 10-2 meter
unit nanosecond ns : Time = 10-9 second

unit inch inches in : Length = 2.54 cm
unit foot feet ft : Length = 12 inches
```

Fortress (Sun HPCS Language). Fig. 16 Example declarations of physical dimensions and units

in that it is invoked implicitly by method dispatch if necessary. Among method overladings, one that requires no coercion to be applicable is always considered more specific than one that requires one or more coercions. Coercion declarations may also

be marked as *widening*; automatic rewriting rules on parts of the expression tree that include widening coercions provide the effect of “widest need evaluation” [5], but in a manner that is fully under user (or library) control.

- Declarations in terms of dimensions implicitly use default units.

$$\text{kineticEnergy}(m : \mathbb{R} \text{ Mass}, v : \mathbb{R} \text{ Velocity}) : \mathbb{R} \text{ Energy} = \frac{m v^2}{2}$$

- Alternatively, declarations may specify units explicitly.

$$\text{kineticEnergy}(m : \mathbb{R} \text{ kg}, v : \mathbb{R} \text{ m/s}) : \mathbb{R} \text{ J} = \frac{m v^2}{2}$$

do

mySpeed = 3.7 feet per second

- The function *kineticEnergy* requires a speed measured in meters per second, so *mySpeed* by itself is not a valid argument.
- The *in* operator multiplies by the necessary scale factor, which in this case is $12 \times 2.54 \times 10^{-2}$. The scale factor is determined automatically from the declarations of the units.

kineticEnergy(30 kg, *mySpeed* in m/sec)

end

Fortress (Sun HPCS Language). Fig. 17 Examples of the use of declared physical dimensions and units

- Assume the *position* method returns a value of type *Maybe*[\mathbb{Z}_{64}].

if p \leftarrow *myString.position*('=') *then*

myString[0 : p] • Here *p* is bound to a value of type \mathbb{Z}_{64} .

else

myString

end

Fortress (Sun HPCS Language). Fig. 18 Use of a generator expression in an *if* statement

- Fortress uses *where* clauses to bind additional type parameters and to constrain relationships among static type parameters. In particular, the constraint may be a subtype relationship. This general mechanism suffices to express covariance and contravariance (Fig. 20), which are important type relationships that require more specialized mechanisms in other languages.
- MPI allows a program to inquire how many hardware processors there are, but little else about the execution environment. High Performance Fortran provides alignment and distribution directives that describe how arrays should be laid out across multiple processors, but the processors must be organized as a flat Cartesian grid, and the class of distributions (such as block, cyclic, and block-cyclic) is fixed.

Fortress defines a *hierarchical* data structure (the *region*) that can describe hardware resources (both processors and memory) and relative communication costs, and can be queried by a running program. A Fortress *distribution* maps the parts of an aggregate data structure to one or more regions. The class of distributions is open ended and library programmers can define new distributions.

Influences from Other Programming Languages

Aspects of the Fortress type system, including inheritance and overloaded multimethod dispatch, were influenced by the ML, Haskell, Java, NextGen, Scala, CLOS, Smalltalk, and Cecil programming languages, among others. Inspirations for the use of mathematical syntax and a character set beyond ASCII include

```

trait FloorWax
  advertise(self) = print "Great shine!"
end

trait DessertTopping
  advertise(self) = print "Great taste!"
end

trait Shimmer extends { FloorWax, DessertTopping }
  advertise(self) = do
    @ Type assumption controls which “super-method” to call.
    advertise(self asif FloorWax)
    advertise(self asif DessertTopping)
  end
end

```

Fortress (Sun HPCS Language). Fig. 19 Example use of `asif` expressions

@ List is covariant in its type parameter T .

```

trait List[T] extends List[U] where { T extends U }
  ...
end

```

@ OutputStream is contravariant in its type parameter E .

```

trait OutputStream[E] extends OutputStream[G] where { G extends E }
  ...
end

```

Fortress (Sun HPCS Language). Fig. 20 Use of `where` clauses to express covariance and contravariance of type parameters

APL, MADCAP, MIRFAC, the Klerer-May System, and COLASL. The idea of having “publication” (beautifully rendered) textual form as well as one or more “implementation” forms goes back to Algol 60. Notions of data encapsulation, comprehensions, generators, and reducers can be traced to CLU, Alphard, KRC, Miranda, and Haskell. In the area of functional programming, higher-order functions, and divide-and-conquer parallelism, many languages had an influence, but especially Lisp, Haskell, and APL. The approach to data distribution was strongly influenced by experience with High Performance Fortran and its predecessors.

Future Directions

The current Fortress compiler targets the Java Virtual Machine (JVM) and uses a custom class loader to instantiate generic types as they are needed. The JVM provides good multiplatform native code generation, threads, and garbage collection, and hosts libraries implementing both work-stealing and transactions. The Fortress type system is embedded into the Java type system where it is convenient, but tuple types and arrow types are handled specially. Fortress value types are eligible for compilation into an unboxed (primitive) representation. Fortress traits compile to Java interfaces, and Fortress objects compile to Java final classes.

A straightforward analysis of traits comprising object singletons can demonstrate finiteness, which permits the general application of unboxing to new types.

A specialized virtual machine (VM) and run-time system may someday be desirable, but at this time the benefits of targeting the JVM far outweigh the overhead.

The design of regions and distributions for managing the mapping of data and threads to hardware resources has been worked out on paper, but has not yet been implemented.

The original design of Fortress includes a form of where clause that conditionally gates such type relationships as extension, comprising, and exclusion. For example, one might define an generic aggregate data structure $A[E]$ with a multiplication operator that is defined in terms of multiplication for the element type E , and one might well wish to declare that the multiplication of aggregates is commutative if and only if multiplication of the elements is commutative. This may be achieved by declaring

```
trait A[E] extends { Commutative[A[E], ×]
  where { E extends Commutative[E, ×] } }
```

The implications of such conditional type relationships for the implementation of a practical type checker are still a subject of research.

An object-oriented pattern-matching facility for easily binding multiple variables to the fields of an object, similar to the pattern-matching facilities in Haskell and Scala but with more provisions for user extensibility, has been sketched out but only recently implemented.

Related Entries

- Chapel (Cray Inc. HPCS Language)
- HPF (High Performance Fortran)
- PGAS (Partitioned Global Address Space) Languages

Bibliography

1. Allen E, Chase D, Flood C, Luchangco V, Maessen JW, Ryu S, Steele Jr GL (2007) Project Fortress Community website. <http://projectfortress.java.net>
2. Allen E, Chase D, Hallett J, Luchangco V, Maessen JW, Ryu S, Steele Jr GL, Hochstadt ST (2008) The Fortress Language Specification

Version 1.0. <http://research.sun.com/projects/plrg/fortress.pdf>, March 2008. See also <http://research.sun.com/projects/plrg/Publications/fortress1.0beta.pdf>, March 2007

3. Allen E, Culpepper R, Nielsen JD, Rafkind J, Ryu S (2009) Growing a syntax. In: ACM SIGPLAN Foundations of Object-Oriented Languages workshop, Savannah, 2009
4. Bailey DH, Barszcz E, Barton JT, Browning DS, Carter RL, Fatoohi RA, Frederickson PO, Lasinski TA, Simon HD, Venkatakrishnan V, Weeratunga SK (1991) The NAS parallel benchmarks. Technical report, Int J Supercomput Appl 5(3):63–73
5. Corbett RP (1982) Enhanced arithmetic for Fortran. ACM SIGPLAN Notices 17(12):41–48, <http://doi.acm.org/10.1145/988164.988168>
6. High Productivity Computer Systems program of the Defense Advanced Research Projects Agency (United States Department of Defense). <http://www.highproductivity.org/> and see also [//www.darpa.mil/IPTO/programs/hpcs/hpcs.asp](http://www.darpa.mil/IPTO/programs/hpcs/hpcs.asp)
7. Steele Jr GL (1998) Growing a Language. Invited talk. Abstract in OOPSLA '98 Addendum: Addendum to the 1998 Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications. ISBN 1-58113-286-7. ACM, New York, <http://doi.acm.org/10.1145/346852.346922> Transcript in Higher-Order and Symbolic Computation 12, 3 (October 1999), 221–236. Video at <http://video.google.com/videoplay?docid=-8860158196198824415#>
8. The Unicode Consortium (2006) The Unicode Standard, Version 5.0. Addison-Wesley, Boston
9. Wiki Creole project and website. Sponsored by the Wiki Symposium. <http://www.wikicreole.org/wiki/Creole1.0>

Forwarding

- Routing (Including Deadlock Avoidance)

Fujitsu Vector Computers

KENICHI MIURA

National Institute of Informatics, Tokyo,
Japan

Synonyms

Fujitsu vector processors; Fujitsu VPP systems; SIMD (single instruction, multiple data) machines

Definition

Fujitsu Vector Machines are the series of supercomputers developed by Fujitsu from 1976 to 1999. They are based on the pipeline architecture, and VPP systems have also incorporated a high degree of parallelism with distributed memory in the system architecture, utilizing the full crossbar network for non-blocking system interconnect.

Discussion

FACOM 230-75 Array Processor Unit (APU)

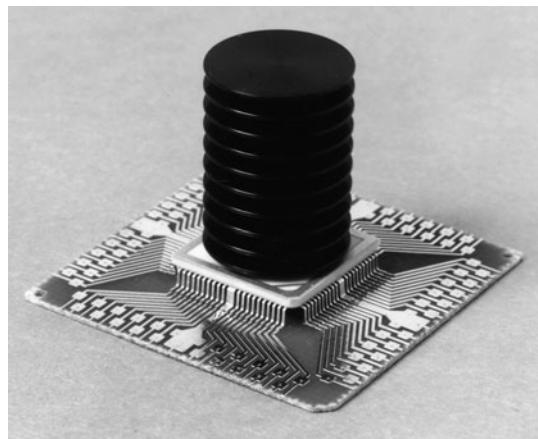
The first vector machine in Japan was FACOM 230-75 Array Processing Unit (APU) which was installed at the National Aerospace Laboratory (later became the Japanese Aerospace Exploration Agency (JAXA) in October 2003) in 1977 [1].

This system was organized as an asymmetric multi-processor system which consists of the mainframe computer FACOM 230-75 and the Array Processor Unit. The clock was 90 ns, and the peak performance of APU is 22 Mflop/s for addition/subtraction and 11 Mflop/s for multiplication. This system was a pipeline-based array processor with 256 data registers (scalar) and one vector register with 1,792 word (36 bits/word) for vector operations. The capacity of the main memory was 1 M words with 32-way interleave. AP-Fortran, an extended Fortran language, was developed for the 230-75 APU to allow parallelism description.

FACOM VP100/200 Series

The vector processor FACOM VP100 and FACOM VP200 [2–5] were announced in July 1982. The major goals of the VP100/200 Series were to achieve a very high performance, ease-of-use, and a good affinity with general-purpose computing environment with the FACOM M-Series mainframe computers. The state-of-the-art semiconductor technology was incorporated in the system; the ECL LSI with a gate delay time of 350 ps (Fig. 1) and 400 and 1,300 gates per chip, and high-speed RAM with an access time of 5.5 ns for vector registers. As for the memory technology, the main memory unit with high capacity and high data transfer capability was realized by using the 64 Kbit SRAM with access time of 55 ns.

Figure 2 illustrates the architecture of VP100/200 Series system. It consists of the scalar unit, vector unit,



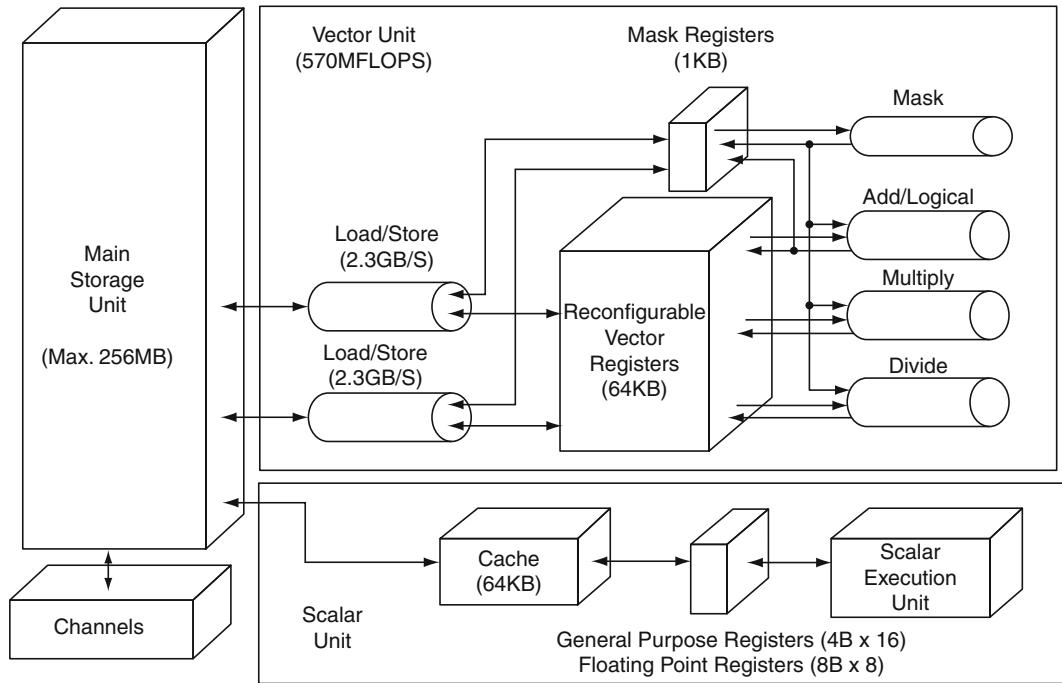
Fujitsu Vector Computers. Fig. 1 VP200 LSI packaging technology. (Photo Credit: Fujitsu Limited)

and main storage unit. The scalar unit fetches and decodes all the instructions and executes the scalar instructions. The vector unit consists of six functional pipeline units, vector registers, and mask registers. The functional pipes are add/logical pipe, multiply pipe, divide pipe, mask pipe, and two load/store ports, which are also pipelined. The first three pipes are for arithmetic operations. The load/store pipes support the contiguous access, strided access, and the indirect addressing (gather/scatter) for flexible vector operations. The mask pipes are used to handle the conditional branches within loops. One of the most unique features of the VP100/200 Series is the dynamically reconfigurable vector registers. The total capacity of the vector registers for VP200 is 8 K words (64 bits), but they can take such configurations as 32 (vector length) × 256 (number of vector registers), $64 \times 128, 128 \times 64, \dots, 1024 \times 8$.

The major features of the VP100/200 Series are summarized in Table 1. Figure 3 shows the cabinet of the VP200 system.

The first delivery of the VP-Series system took place in January 1984, that is, VP-100 at the Institute for Plasma Physics in Nagoya. Later, the high-end model FACOM VP-400, with improved performance, was added, and a processing performance exceeding 1Gflop/s was achieved. Also, the market was expanded by adding lower-end models: the FACOM VP30 and FACOM VP 50.

The VP-Series E Models, enhanced versions of the VP 100/200 Series, were announced in July 1987.



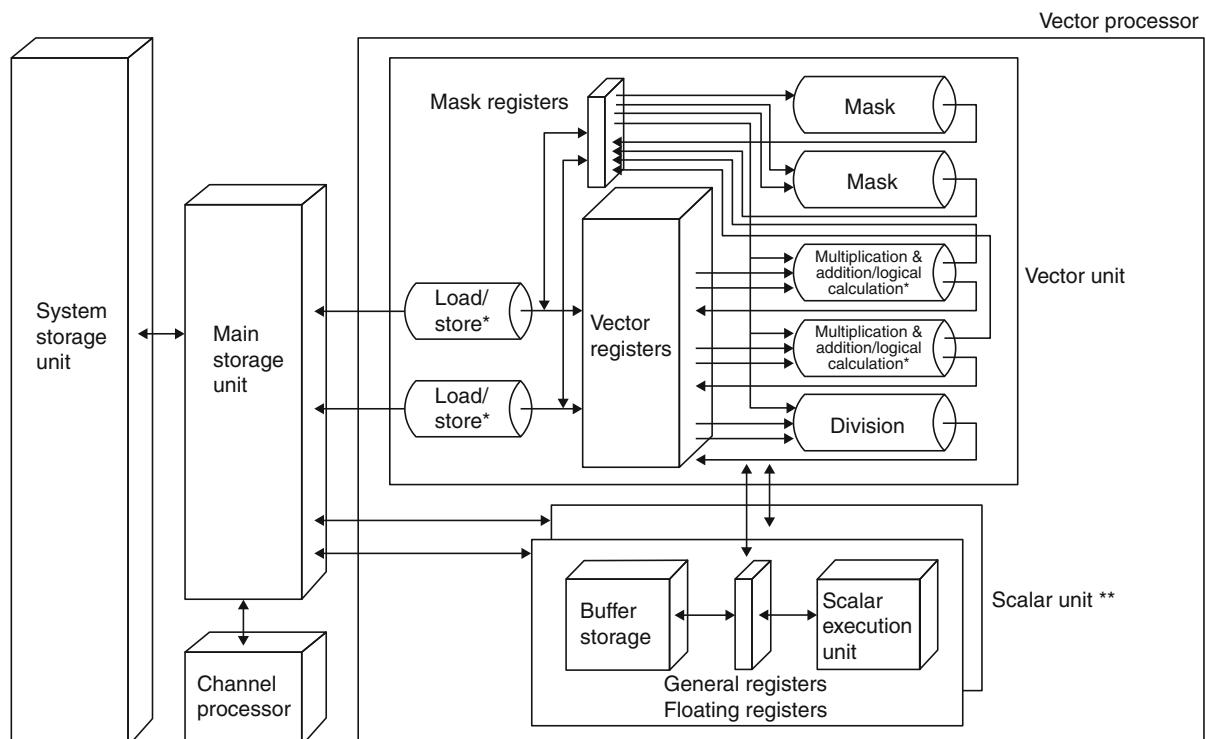
Fujitsu Vector Computers. Fig. 2 VP200 architectural block diagram

Fujitsu Vector Computers. Table 1 VP100 Series specifications

Model	VP50	VP100	VP200	VP400
Announcement	April, 1985	July, 1982	July, 1982	April, 1985
Machine cycle	7.0 ns	7.0 ns	7.0 ns	7.0 ns
Max. performance	140 Mflop/s	285 Mflop/s	570 Mflop/s	1,140 Mflop/s
Total vector regs. size	32 KB	32 KB	64 KB	128 KB
Basic vector length	16 elements	16 elements	32 elements	64 elements
Total Mask regs.	4 kbits	4 kbits	8 kbits	16 kbits
Logical no. times multiplicity of vector pipes				
Add/logical	1 × 1	1 × 1	1 × 2	1 × 4
Multiply	1 × 1	1 × 1	1 × 2	1 × 4
Divide	1 × 1	1 × 1	1 × 2	1 × 4
Mask	1 × 1	1 × 1	1 × 2	1 × 4
Load/Store	1 × 1	2 × 1	2 × 2	1 × 4
Ckt. technology (ECL) & delay	400 gates/chip 350 ps	400 gates/chip 350 ps	400 gates/chip 350 ps	400 gates/chip 350 ps
Memory technology	64 kbits SRAM 55 ns access			
Main memory size	32, 64, 128 MB	32, 64, 128 MB	64, 128, 256 MB	64, 128, 256 MB



Fujitsu Vector Computers. Fig. 3 VP200 system. (Photo Credit: Fujitsu Limited)



* Model VP2000 provides separate pipelines for loading/storing and addition/subtraction/ logic calculations.

**Uni-processor models have one scalar unit.

Fujitsu Vector Computers. Fig. 4 VP2000 architectural block diagram

Fujitsu Vector Computers. Table 2 VP2000 Series specifications

Models	VP2100/10 VP2100/20	VP2200/10 VP2200/20	VP2400/10 VP2400/20	VP2600/10 VP2600/20
Announcement	Dec. 1988	Dec. 1988	Dec. 1988	Dec. 1988
Max. performance	500 Mflop/s	1Gflop/s	2 Gflop/s	4 Gflop/s
No. scalar unit/ vector unit	1–2 1	1–2 1	1–2 1	1–2 1
Total capacity of vector registers	32 KB	32 KB	64 KB	128 KB
Total capacity of mask registers	4 kbits	4 kbits	8 kbits	16 kbits
No. vector pipes & multiplicity	5 1	5 1	7 2	7 4
Ckt. technology (ECL) & delay/gate	15 K gates 70 ps			
Ckt technology (Capacity & delay)	64 kbit SRAM 1.6 ns			
Capacity of main memory	32 MB–1 GB	64 MB–1 GB	128 MB–1 GB	128 MB–2 GB
Memory technology /delay	1Mbit SRAM/ 35 ns	1Mbit SRAM/ 35 ns	1Mbit SRAM/ 35 ns	1Mbit SRAM/ 35 ns
Capacity of system storage	1,2,4,6,8 GB	1,2,4,6,8 GB	1,2,4,6,8 GB	1,2,4,6,8 GB
I/O capabilities	Max. 1 GB/s	Max. 1 GB/s	Max. 1 GB/s	Max. 1 GB/s
No. channels	16–128	16–128	16–128	16–128
Block Mpx Ch./ Optical Ch.	4.5 MB/s 9.0 MB/s	4.5 MB/s 9 MB/s	4.5 MB/s 9.0 MB/s	4.5 MB/s 9.0 MB/s

There were five models : VP 30E, VP 50E, VP 100E, VP 200E, and VP 400E. These E models achieved performance which is 1.5 times more powerful than that of the previous VP 100/200 Series machines by installing high-speed vector memory and an additional arithmetic pipeline for the fused multiply-add operations.

FUJITSU VP2000 Series

The VP2000 Series [6–8] was announced in December 1988 as the successor to Fujitsu's VP-E Series supercomputers. Figure 4 illustrates the architecture of VP2000 Series systems. It carries most of the VP100/200 architectural features with various enhancements in the vector unit, such as two sets of fused multiply-add pipelines, doubled mask pipelines, and faster division circuit. The most notable feature of the VP2000 Series is that it can be configured with one or two scalar units

sharing one vector unit. This feature, called Dual Scalar Processors (DSP), may be regarded as two-threaded system. The objective of DSP is to utilize the vector unit more efficiently when the vectorization ratio of users' application programs is not very high [7].

The key features of VP2000 Series system are summarized in Table 2.

Numerical Wind Tunnel and VPP500-Series

Numerical Wind Tunnel

The National Aerospace Laboratory (NAL) of Japan developed the Numerical Wind Tunnel (NWT) [9] parallel supercomputer system jointly with Fujitsu. The system architecture was a distributed memory vector-parallel computer, in which vector supercomputers



Fujitsu Vector Computers. Fig. 5 NWT system. (Photo Credit: JAXA)

were connected with the non-blocking crossbar network. The NWT's final specifications were 166 processing elements, each with 1.7 Gflop/s of processing power to give a peak performance of 280 Gflop/s with 44.5 GB of the total main memory.

The NWT's processing element consisted of a CPU and memory. The CPU is a vector supercomputer by itself, comprising three types of LSIs: GaAs, ECL, and BiCMOS, 121 in total. It is one of the very few commercially successful GaAs-based system. The primary cooling method was water cooling, but the forced air-cooling method was also used for the memory units.

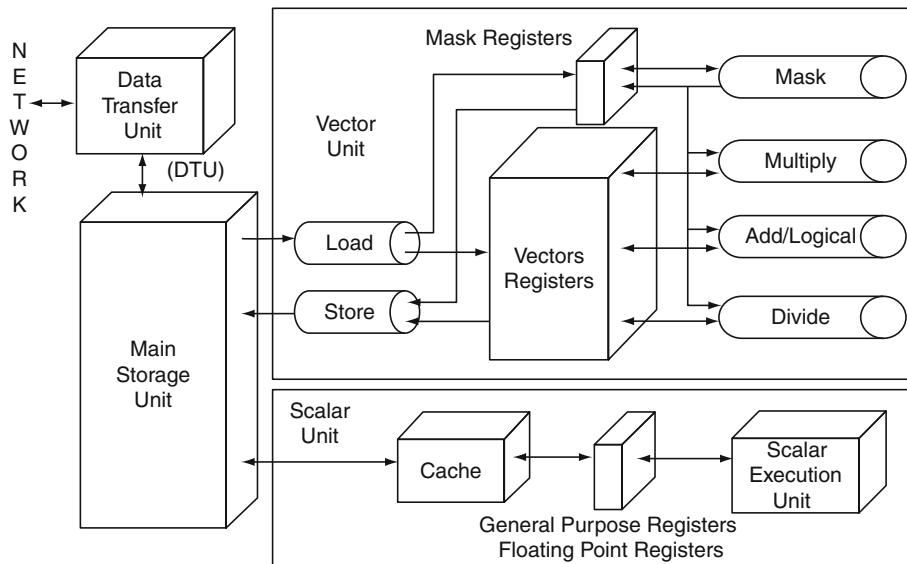
Figure 5 shows the NWT as installed at NAL. The NWT, which went into operation in January 1993, was rated as the most powerful supercomputer site in the world in the TOP500 list from 1993 to 1995. It should be noted that the NWT was the technology precursor to the VPP500, which will be described in more detail in the next section.

FUJITSU VPP500

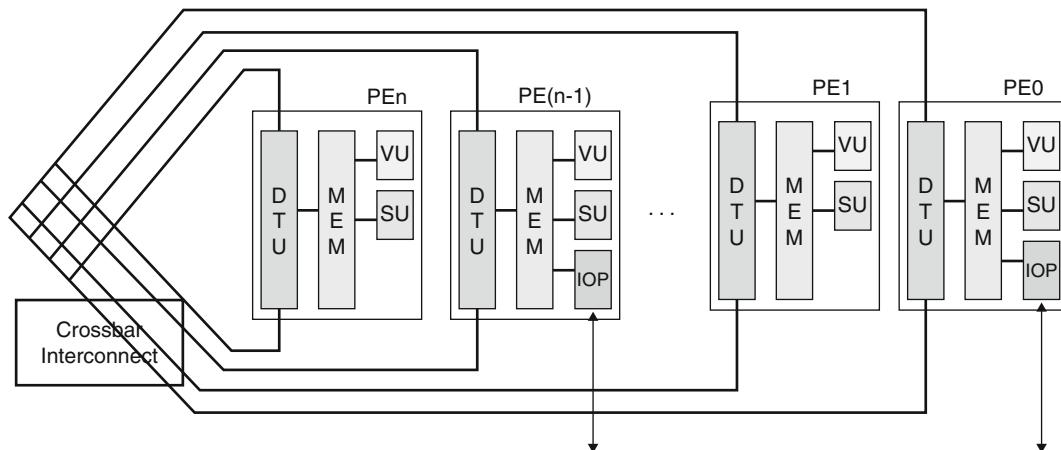
The VPP500 was Fujitsu's third generation supercomputer system [8, 10–14]. As stated in the previous section, it was a commercial version of NWT. Fujitsu announced the VPP500 in September 1992 as the world's most powerful supercomputer at that point in

time with a speed of 355 Gflop/s. Fujitsu has developed processor elements (PEs), each being capable of vector processing and also can operate in highly parallel fashion to obtain this performance. The maximum configuration was with 222 processing elements to reach 355 Gflop/s.

As shown in Fig. 6, each processing element consisted of a scalar unit, a vector unit, a main storage unit, and a Data Transfer Unit (DTU), built with Gallium Arsenide LSI's, Bipolar LSI's and BiCMOS LSI's, and ultrahigh-density circuit boards, and was able to deliver a maximum vector performance of 1.6 Gflop/s. The scalar unit is VLIW architecture and may execute up to four operations in parallel. In order to achieve very high system performance, PEs were connected with a high-speed crossbar network which allows non-blocking data transfer. The data transfer rate was 400 MB/s from one PE to another. Since a PE can simultaneously send and receive data, the aggregate data transfer rate per PE was 800 MB/s. Figure 7 illustrates the system configuration of VPP500, based around the crossbar network. In order to fully utilize the powerful capability of the interconnecting network, the Data Transfer Unit (DTU) was incorporated, which allows the mapping between the local and the global addresses, a very fast synchronization across PEs, and actual data transfer with proprietary message passing protocol. Various



Fujitsu Vector Computers. Fig. 6 VPP500 architectural block diagram



Fujitsu Vector Computers. Fig. 7 VPP500 system configuration

data transfer modes were supported by the DTU. They were: contiguous, constant-strided, sub-array, and indirect addressing (random gather/scatter).

Figure 8 shows the inside of one cabinet of VPP500, which contains 12 PEs. Plumbing for water cooling is also visible in this photo.

The specifications of the VPP500 system and the follow-on products are summarized in Table 3.

the distributed memory units of VPP500. The ability to define the array data in Fortran programs, distributed across multiple processing elements as a single global space, greatly simplified the porting and tuning of existing codes to VPP500 VPP Fortran may be regarded as an early implementation of PGAS (Partitioned Global Address Space). More details are described in [13].

New Parallel Language

Fujitsu developed a new parallel language, called VPP Fortran, which can define a single name space across

VPP300/VPP700/VPP5000 Series

After a great success with VPP500, Fujitsu developed the CMOS versions of the vector-parallel system. The



Fujitsu Vector Computers. Fig. 8 VPP500 cabinet. (Photo Credit: Fujitsu Limited)

Fujitsu VPP300 Series and VX Series (the single CPU model), which used CMOS (Complementary metal

oxide semiconductor) technology, were introduced in February 1995, the Fujitsu VPP700 Series in March 1996, and the VPP5000 Series in April 1999. All VPP Series supercomputers after the VPP300 Series used the CMOS technology. Packaging technology of VPP5000 is shown in Fig. 9.

The VPP5000 was the successor to the former VPP700/VPP700E systems (with E for extended, i.e., the clock cycle 6.6 instead of 7 ns). The clock cycle has been halved, and the vector pipes are able to deliver fused multiply-add results. With a multiplicity of 16 for these vector pipes, 32 floating-point results per clock cycle can be generated. In this way, a fourfold increase in speed per processor can be attained with respect to the VPP700E.

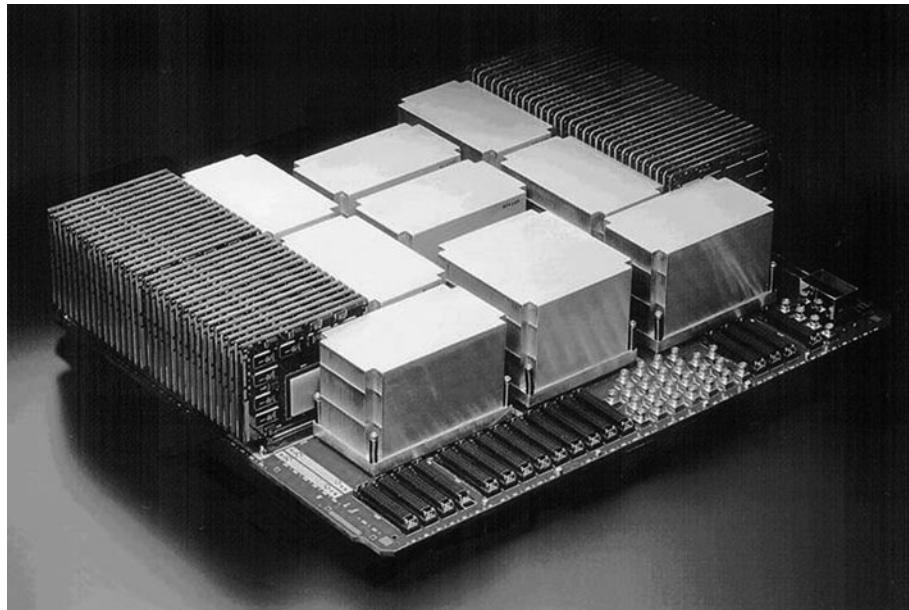
The architecture of the VPP5000 nodes is almost identical to that of the VPP700, but one of the major enhancements is in the performance of noncontiguous memory access by incorporating more address matching circuits.

Conclusion

While the vector processing with very sophisticated hardware pipelines together with highly interleaved memory subsystem was the dominant supercomputer architecture in the 1980s and 1990s, the wave of highly (or massively) parallel scalar systems have been gaining

Fujitsu Vector Computers. Table 3 VPP Series specifications (typical models)

Models	VPP500	VPP700E	VPP5000
Announcement	September, 1992	February, 1995	April, 1999
Ckt. technology	GaAs/ECL/Bipolar	CMOS(.35μ)	CMOS(.22μ)
Level of integration	100 K Tx's (GaAs)	8 M Tx's/chip	33 M Tx's/chip
Delay time/gate	60 ps	40 ps	24 ps
Cooling method	Liquid (water)	Forced air	Forced air
Clock frequency	100 MHz	154 MHz	303 MHz
Max. perf./PE	1.6 Gflop/s	2.4 Gflop/s	9.6 Gflop/s
Mem. capacity/PE	1GB	2 GB	2–16 GB
Memory technology	4 Mbit SRAM	16 Mbit SDRAM	128 Mbit SDRAM, 16 Mbit SSRAM
I/O capability/PE	400 MB × 2	615 MB × 2	3.2 GB × 2
Max. configuration	222 PEs	512 PEs	512 PEs
Max. system perf.	355 Gflop/s	1.2 Tflop/s	4.8 Tflop/s
CPU architecture	32 bit	32 bit	64 bit



Fujitsu Vector Computers. Fig. 9 VPP5000 PE/Memory board. (Photo Credit: Fujitsu Limited)

more popularities in this century. The major reasons may be stated as;

1. Pipelining has become the common techniques even in the scalar microprocessor design.
2. Larger market opportunities have arisen with the advent of high volume produced the powerful scalar microprocessors.
3. The highly interleaved memory structure in the vector architecture has become a more expensive implementation as compared with the cache-based approach in the scalar architecture, for the purpose of coping with the widening speed gap between CPU and memory.
4. More and more application programs have been tailored toward scalar architecture.

Thus, Fujitsu decided to change direction from vector architecture to highly parallel scalar Symmetric Multi-processor (SMP) and its constellations. The VPP5000 system was Fujitsu's last vector product.

But it should also be pointed out that the recent trends in SIMD instructions in the general purpose microprocessors and the popularity of GPGPU (General Purpose Graphic Processing Unit) could be regarded as "the return of vectors," in the sense that the

programming models and compiler technology developed for the vector architecture continue to be vital technologies in the new approaches.

Related Entries

- PGAS (Partitioned Global Address Space) Languages
- TOP500
- VLIW Processors

Bibliography

1. Uchida K, Seta Y, Tanakura Y (1978) The FACOM 230-75 Array Processor System. In: Proc 3rd USA-JAPAN Computer Conference, San Francisco, CA. AFIPS Press, Montvale, NJ, pp 369–373
2. Miura K, Uchida K (1984) FACOM vector processor VP-100/VP-200. In: Kowalik JS (ed) High-speed computation, NATO ASI Series F: computer and systems sciences, vol 7. Springer, Berlin, pp 127–138
3. Miura K (1986) Fujitsu's supercomputer: FACOM vector processor system. In: Fernbach S (ed) Supercomputers: class VI systems, hardware and software. North-Holland, Amsterdam, pp 137–152
4. Matsuura T, Miura K, Makino M (1985) Supervector performance without toil: Fortran implemented vector algorithms on the VP-100/200. Comput Phys Commun 37:101–107

5. Miura K (1990) Vectorization and parallelization of transport Monte Carlo simulation codes. In: NATO ASI Series vol F62, pp 307–324
6. Uchida N et al (1990) Fujitsu VP2000 Series. In: Proceedings of COMPCON Spring'90, San Francisco, CA. IEEE Computer Society Press, Washington, DC, pp 4–11
7. Miura K, Nagakura H, Tamura H (1991) VP2000 Series dual scalar and quadruple scalar models supercomputer systems – A new concept in vector processing. In: Proceedings of COMPCON Spring'91, San Francisco, CA. IEEE Computer Society Press, Washington, DC, pp 294–302
8. Hwang K (1993) Section 8.2.3 Fujitsu VP2000 and VPP500. In: Advanced computer architecture: parallelism, scalability, programmability. McGraw-Hill, Inc., New York, pp 425–429
9. Miyoshi H et al (1994) Development and achievement of NAL numerical wind tunnel (NWT) for CFD computations. In: Proceedings of 1994 ACM/IEEE conference on supercomputing (SC'94), Washington, DC. IEEE Computer Society Press (1994), Washington, DC, pp 685–692
10. Utsumi T, Ikeda M, Takamura M (1994) Architecture of the VPP500 parallel supercomputer. In: Proceedings of supercomputing'94, Washington, DC. IEEE Computer Society Press, Washington, DC, pp 478–486
11. Nakashima Y, Kitamura T, Tamura H, Takiuchi M, Miura K (1994) The scalar processor of the VPP500 parallel supercomputer. In: Proceedings of the 9th ACM international conference on supercomputing (ICS'94), Manchester, England. ACM, New York, pp 348–356
12. Nakanishi M, Ina H, Miura K (1994) A high performance linear equation solver on the VPP500 parallel supercomputer. In: Proceedings of 1994 ACM/IEEE conference on supercomputing (SC'94), Washington, DC. IEEE Computer Society Press, Washington DC, pp 803–810
13. Iwashita H et al (1994) VPP Fortran and parallel programming on the VPP500 supercomputer. In: Proceedings of 1994 international symposium on parallel architectures, algorithms and networks (ISPAN'94), Kanazawa, Japan, 14–16 Dec 1994, pp 165–172
14. Nodomi A, Ikeda M, Takamura M, Miura K (1995) Hardware performance of the VPP500 parallel supercomputer. In: Dongarra J, Grandinetti L, Joubert G, Kowaliak J (eds) High performance computing: technology, method and applications, vol 10. Advances in Parallel Computing. Elsevier, Amsterdam, pp 103–120

Fujitsu Vector Processors

- Fujitsu Vector Computers

Fujitsu VPP Systems

- Fujitsu Vector Computers

Functional Decomposition

- Domain Decomposition

Functional Languages

PHILIP TRINDER¹, HANS-WOLFGANG LOIDL¹,
KEVIN HAMMOND²

¹Heriot-Watt University, Edinburgh, UK

²University of St. Andrews, St. Andrews, UK

Definition

Parallel functional languages are parallel variants of functional languages, that is, languages that treat computation as the evaluation of mathematical functions and avoid state and mutable data. Functional languages are founded on the lambda calculus. The majority of parallel functional languages add a small number of high-level parallel coordination constructs to some functional language, although some introduce parallelism without changing the language.

Discussion

Introduction

The potential of functional languages for parallelism has been recognized for over 30 years, for example [44]. The key advantage of functional languages is that they mainly, or solely, contain stateless computations. Subject to data dependencies between the computations, the evaluation of stateless computations can be arbitrarily reordered or interleaved while preserving the sequential semantics.

Parallel programs must not only specify the *computation*, that is, a correct and efficient algorithm, it must also specify the *coordination*, for example, how the program is partitioned, or how parts of the program are placed on processors. As functional languages provide high level constructs for specifying computation, for example, higher-order functions and sophisticated type systems, they typically provide correspondingly high level coordination sublanguages. A wide range of parallel paradigms and constructs have been used,

for example, data-parallelism [5], and skeleton-based parallelism [37].

As with computation, the great advantage of high-level coordination is that it frees the programmer from specifying low-level coordination details. Where low-level coordination constructs encourage programmers to construct static, simple, or regular coordination, higher-level constructs encourage more dynamic and irregular coordination. The challenges of high-level coordination are that automatic coordination management complicates the operational semantics, makes the performance of programs opaque, requires a sophisticated language implementation, and is frequently less effective than hand-crafted coordination. Despite these challenges many new nonfunctional parallel programming languages also support high-level coordination, and examples include Fortress [1], Chapel [10] or X10 [11] that also support high-level coordination.

Coordination may be managed statically by the compiler as in PSML [31], dynamically by the runtime system as in GpH [41], or by both as in Eden [28]. Whichever mechanism is chosen, the implementation of sophisticated automatic coordination management is arduous, and there have been many more parallel language designs than well-engineered implementations.

The combination of a high-level computation and coordination language requires specialized parallel development tools and methodologies, as discussed below.

Strict and Non-strict Parallel Functional Languages

Most programming languages strictly evaluate the arguments to a function before evaluating the function. However, as purely functional languages have significant freedom of evaluation order, some use non-strict or *lazy* evaluation. Here the arguments to a function are evaluated only if and when they are demanded by the function body. In consequence more programs terminate, it is easy to program with infinite structures, and programming concerns are separated [23].

Like most languages, many parallel functional languages are strict, for example, NESL [5] or Manticore [19]. However there are many parallel variants of lazy

languages like Haskell [34] and Clean [32]. At first glance this is surprising as lazy evaluation is sequential and performs minimum work, but non-strict languages have a number of advantages for parallelism [42]. They can be easily married to many different coordination languages as the execution order of expressions is immaterial; they naturally support highly dynamic coordination where evaluation is performed and data communicated on demand; their implementations already have many of the mechanisms required for parallel execution. Laziness also exists in the language implementations, for example, distributing work and data on demand [41].

Paradigms

Parallel functional languages have adopted a range of paradigms. For the main parallel paradigms this section outlines the correspondence with the functional paradigm, and describes a representative language. Some languages support more than one paradigm, for example, Manticore [19] combines data and task parallelism, and Eden [28] combines semi-explicit and skeleton parallelism.

Skeleton-Based Languages

Algorithmic Skeletons [14, 37] are a popular parallel coordination construct, and as higher-order functions fit naturally in the functional model. Often these higher order functions work over compound data structures like lists or vectors and consequently the resulting parallel code often resembles data parallel code as discussed in section.

Example skeleton-based functional languages include SCL [15], P3L [3], Eden [28], PSML [31], and HDC. HDC [21] is a strictly evaluated subset of the Haskell language with skeleton-based coordination. HDC programs are compiled using a set of skeletons for common higher-order functions, like fold and map, and several forms of divide-and-conquer. The language supports two divide-and-conquer skeletons and a parallel map, and the system relies on the use of these higher-order functions to generate parallel code.

Data Parallel Languages

Data parallel languages [33] focus on the efficient implementation of the parallel evaluation of every element

in a collection. Functional languages fit well with the data parallel paradigm as they provide powerful operations on collection types, and in particular lists. Indeed, all of the languages discussed here use some parallel extension of list comprehensions and implicitly parallel higher order functions such as `map`. Compared to other approaches to parallelism, the data parallel approach makes it easier to develop good cost models, although, it is notoriously difficult to develop cost models for languages with a non-strict semantics.

Example data parallel functional languages include NESL [5] and Data Parallel Haskell [30]. NESL is a strict, strongly typed, language with implicit parallelism and implicit thread interaction. It has been implemented on a range of parallel architectures, including several vector computers. A wide range of algorithms have been parallelized in NESL, including a Delaunay algorithm for triangulation [8], several algorithms for the n-body problem [7], and several graph algorithms.

Semi-explicit Languages

Semi-explicit parallel languages provide a few high-level constructs for controlling key coordination aspects, while automatically managing most coordination aspects statically or dynamically. Historically, annotations were commonly used for semi-explicit coordination, but more recent languages provide compositional language constructs. As a result, the distinction between semi-explicit coordination and coordination languages is now rather blurred, but the key difference in the approach is that semi-explicit languages aim for *minimal* explicit coordination and minimal change to language semantics.

Example semi-explicit parallel functional languages include Eden [28] and GpH [40]. GpH is a small extension of Haskell with a parallel (`par`) composition primitive. `par` returns its second argument and indicates that the first argument *may* be executed in parallel. In this model the programmer only has to expose expressions in the program that can usefully be evaluated in parallel. The runtime-system manages the details of the parallel execution such as thread creation, communication, etc. Experience in implementing large programs in GpH shows that the unstructured use of `par` and `seq` operators often leads to

obscure programs. This problem is overcome by *evaluation strategies*: lazy, polymorphic, higher-order functions controlling the evaluation degree and the parallelism of a Haskell expression. They provide a clean separation between coordination and computation. The driving philosophy behind evaluation strategies is that it should be possible to understand the computation specified by a function without considering its coordination.

Coordination Languages

Parallel coordination languages [26] are separate from the computation language and thereby provide a clean distinction between coordination and computation. Historically, Linda [12] and PCN [18] have been the most influential coordination languages, and often a coordination language can be combined with many different computation languages, typically Fortran or C. Other systems such as SCL [15] and P3L [3] focus on a skeleton approach for introducing parallelism and employ sophisticated compilation technology to achieve good resource management.

Example coordination languages using functional computation languages include Haskell-Linda [35] and Caliban [25, 39]. Caliban has constructs for explicit partitioning of the computation into threads, and for assigning threads to (abstract) processors in a static process network. Communication between processors works on streams, that is, eagerly evaluated lists, similar to Eden.

Dataflow Languages

Historically a number of functional languages adopted the dataflow paradigm and some, like SISAL [9], achieved very impressive performance.

Example dataflow functional languages include SISAL [9], pHluid system [17]. SISAL [9] is a first-order, strict functional language with implicit parallelism and implicit thread interaction. Its implementation is based on a dataflow model and it has been ported to a range of parallel architectures. Comparisons of SISAL code with parallel Fortran code show that its performance is competitive with Fortran, without adding the additional complexity of explicit coordination [27].

Explicit Languages

In contrast to the high-level coordination found in the majority of parallel functional languages, a number of languages support low-level explicit coordination. For example, there are several bindings for explicit message passing libraries, such as PVM [36] and MPI [29] for languages like Haskell and OCaml [6, 43].

These languages use an open system model of explicit parallelism with explicit thread interaction. Since the coordination language is basically a stateful (imperative) language, stateful code is required at the coordination level. Although the high availability and portability of these systems are appealing, the language models suffer from the rigid separation between the stateful and purely functional levels.

Other functional languages support explicit coordination, for example, ERLANG [2]. ERLANG is probably the most commercially successful functional language, and was developed in the telecommunications industry for constructing distributed, real-time fault tolerant systems. ERLANG is strict, impure, weakly typed, and relatively simple: omitting features such as currying and higher order functions. However the language has a number of extremely useful features, including the OTP libraries, hot loading of new code into running applications, explicit time manipulation to support soft real-time systems, message authentication, and sophisticated fault tolerance.

Tools and Methodologies

Development methodologies for parallel functional programs exploit the amenability of the languages to derivation and analysis. For example, a programmer can reason about costs during program design using abstract cost models like BMF-PRAM [38]. Guided by the cost information the program design can relatively easily be transformed to reduce resource consumption before implementation, for example, [16]. Similarly, automated static resource analysis can provide information to improve coordination, for example, predicted task execution time can improve scheduling.

Parallel functional languages favour high-level and often dynamic coordination in contrast to detailed static

coordination. As a consequence the parallel behaviour of programs is often far from obvious, and hence hard to tune. Hence suites of profiling and visualization tools are very important for many parallel functional languages, for example [4, 24, 39].

Implicit parallelism, often promised in the context of functional languages, offers the enticing vision of parallel execution without changes to the program. In reality, however, the program must be designed with parallelism in mind to avoid unnecessary sequentiality. In theory, program analyses such as granularity, sharing, and usage analysis can be used to automatically generate parallelism. In practice, however, almost all current systems rely on some level of programmer control.

Current development methodologies, like [40], have several interesting features. The combination of languages with high-level coordination and good profiling tools facilitates the prototyping of alternative parallelisations. Obtaining good coordination at an early stage of parallel software development avoids expensive redesigns. In later development stages, detailed control over small but crucial parts of the program may be required, and profiling tools can help locate expensive parallel computations. During performance tuning the high level of abstraction may obscure key low-level features that could be usefully controlled by the programmer.

Parallel Functional Languages Today

Parallel functional languages are used in a range of domains including numeric, symbolic, and data intensive [22]. Commercially Erlang has been enormously successful for developing substantial telecoms and banking applications [2] and parallel Haskell for providing standardized parallel computational algebra systems.

The high-level coordination and computation constructs in parallel functional languages have been very influential. For example, algorithmic skeletons appear in MPI [29] and are the interface for cloud parallel search engines like Google MapReduce. Similarly, stateless threads or comprehensions occur in modern parallel languages like Fortress [1], Chapel [10], or X10 [11]. Moreover the dominance of multicore and emergence

of many-core architectures has focused attention on the importance of stateless computation.

Some trends are as follows. As clock speeds plateau but register width increases, data parallelism becomes attractive for functional, and other, parallel languages. In the functional community there are often multiple parallelizations of a single base language, for example, GpH and Eden are both Haskell variants, moreover, a single compiler often supports more than one parallel version, for example, the Glasgow Haskell Compiler has four experimental multicore implementations [20].

Related Entries

- [Data Flow Graphs](#)
- [Data Flow Computer Architecture](#)
- [MPI \(Message Passing Interface\)](#)
- [Parallel Skeletons](#)
- [Profiling](#)

Languages

- Chapel (Cray inc. HPCS Language)
- Concurrent ML
- Eden
- Fortress (Sun HPCS Language)
- Glasgow Parallel Haskell
- Multilisp
- NESL

Bibliographic Notes and Further Reading

A sound introduction to parallel functional language concepts and research can be found in [22]. A comprehensive survey of parallel and distributed languages based on Haskell is available in [42]. A recent series of workshops studies declarative aspects of multicore programming, for example [13].

Bibliography

1. Allen E, Chase D, Flood C, Luchangco V, Maessen J-W, Steele S, Ryu GL (2007) Project fortress: a multicore language for multicore processors. *Linux Magazine*, September 2007
2. Armstrong J (2007) Programming Erlang: software for a concurrent world. The Pragmatic Bookshelf, Raleigh, NC
3. Bacci B, Danelutto M, Orlando S, Pelagatti S, Vanneschi M (1995) P₃L: a structured high level programming language and its structured support. *Concurrency Practice Exp* 7(3):225–255
4. Berthold J, Loogen R (2007) Visualizing parallel functional program runs – case studies with the eden trace viewer. In: ParCo'07: Proceedings of the parallel computing: architectures, algorithms and applications, Jülich, Germany
5. Blelloch GE (1996) Programming parallel algorithms. *Commun ACM* 39(3):85–97
6. Breitinger S, Loogen R, Priebe S (1998) Parallel programming with Haskell and MPI. In: IFL'98 – International Workshop on the implementation of functional languages, University College London, UK, September 1998. Draft proceedings, pp 135–154
7. Blelloch GE, Miller GL, Talmor D (1996) Developing a practical projection-based parallel Delaunay algorithm. In: Symposium on Computational Geometry. ACM, Philadelphia, PA
8. Blelloch GE, Narlikar G (1997) A practical comparison of N-body algorithms. In: Parallel Algorithms, volume 30 of Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, University of Tennessee, Knoxville, TN
9. Cann D (1992) Retire Fortran? A debate rekindled. *Commun ACM* 35(8):81–89
10. Chamberlain BL, Callahan D, Zima HP (2007) Parallel programmability and the chapel language. *Int J High Perform Comput Appl* 21(3):291–312
11. Charles P, Donawa C, Ebcioğlu K, Grothoff C, Kielstra A, von Praun C, Saraswat V, Sarkar V (2005) X10: An object-oriented approach to non-uniform cluster computing. In: OOPSLA'05. ACM Press, New York
12. Carriero N, Gelernter D (1989) How to write parallel programs: a guide to the perplexed. *ACM Comput Surv* 21(3):323–357
13. Chakravarty MMT (ed) (2009) ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming (DAMP'09). ACM Press, New York
14. Cole M (1999) Algorithmic skeletons. In: Hammond K, Michaelson G (eds) Research directions in parallel functional programming. Springer-Verlag, New York, pp 289–303
15. Darlington J, Guo Y, To HW (1996) Structured parallel programming: theory meets practice. In Milner R, Wand I (eds) Research Directions in Computer Science. Cambridge University Press, Cambridge, MA
16. Ellmenreich N, Lengauer C (2007) Costing stepwise refinements of parallel programs. *Comp Lang Syst Struct* 33(3–4):134–167 (special issue on semantics and cost models for high-level parallel programming)
17. Flanagan C, Nikhil RS (1996) pHluid: the design of a parallel functional language implementation on workstations. In: ICFP'96 – International Conference on functional programming, ACM Press, Philadelphia, PA, pp 169–179
18. Foster I, Olson R, Tuecke S (1992) Productive parallel programming: The PCN approach. *J Scientific Program* 1(1):51–66
19. Fluet M, Rainey M, Reppy J, Shaw A (2008) Implicitly-threaded parallelism in Manticore. In: ICFP'08 – International Conference on functional programming, ACM Press, Victoria, BC

20. Glasgow Haskell Compiler. WWW page, 2009. <http://www.haskell.org/ghc/>
21. Herrmann C, Lengauer C (2000) HDC: a higher-order language for divide-and-conquer. *Parallel Processing Lett* 10(2–3): 239–250
22. Hammond K, Michaelson G (1999) Research directions in parallel functional programming. Springer-Verlag, New York
23. Hughes J (1989) Why functional programming matters. *Computer J* 32(2):98–107
24. Jones Jr D, Marlow S, Singh S (2009) Parallel performance tuning for haskell. In: Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell. ACM Press, New York
25. Kelly PHJ (1989) Functional programming for loosely-coupled multiprocessors. Research monographs in parallel and distributed computing. MIT Press, Cambridge, MA
26. Kelly P, Taylor F (2001) Coordination languages. In: Hammond K, Michaelson G (eds) Research directions in parallel functional programming. Springer-Verlag, New York, pp 305–321
27. Sisal Performance Data. WWW page, January 2001. <http://www.llnl.gov/sisal/PerformanceData>.
28. Loogen R, Ortega-Mallen Y, Pena R (2005) Parallel functional programming in Eden. *J Funct Program* 15(3):431–475
29. MPI-2: Extensions to the message-passing interface. Technical report, University of Tennessee, Knoxville, TN, July 1997
30. Chakravarty MMT, Leshchinskiy R, Jones SP, Keller G (2008) Partial vectorisation of haskell programs. In DAMP'08: ACM SIGPLAN workshop on declarative aspects of multicore programming, San Francisco, CA
31. Michaelson G, Horiguchi S, Scaife N, Bristow P (2005) A parallel SML compiler based on algorithmic skeletons. *J Funct Program* 15(4):615–650
32. Nöcker EGJMH, Smetsers JEW, van Eekelen MCJD, Plasmeijer MJ (1991) Concurrent clean. In: PARLE'91 — parallel architectures and languages Europe, LNCS 505. Springer-Verlag, Veldhoven, the Netherlands, pp 202–219
33. O'Donnell J (1999) Data parallelism. In: Hammond K, Michaelson G (eds) Research directions in parallel functional programming. Springer-Verlag, New York, pp 191–206
34. Jones SLP, Hughes J, Augustsson L, Barton D, Boutel B, Burton W, Fasel J, Hammond K, Hinze R, Hudak P, Johnsson T, Jones M, Launchbury J, Meijer E, Peterson J, Reid A, Runciman C, Wadler P (1999) Haskell 98: a non-strict, purely functional language. Electronic document available on-line at <http://www.haskell.org/>, February 1999
35. Peterson J, Trifonov V, Serjantov A (2000) Parallel functional reactive programming. In: PADL'00 – practical aspects of declarative languages, LNCS 1753. Springer-Verlag, New York, pp 16–31
36. Parallel Virtual Machine Reference Manual. University of Tennessee, August 1993
37. Rabhi FA, Gorlatch S (eds) (2002) Patterns and skeletons for parallel and distributed computing. Springer-Verlag, New York, 2002
38. Skillicorn DB, Cai W (1995) A cost calculus for parallel functional programming. *J Parallel Distrib Comput* 28(1):65–83
39. Taylor FS (1997) Parallel functional programming by partitioning. PhD thesis, Department of Computing, Imperial College, London
40. Trinder PW, Hammond K, Loidl H-W, Jones SLP (1998) Algorithm + Strategy = Parallelism. *J Funct Program* 8(1): 23–60
41. Trinder PW, Hammond K, Mattson Jr JS, Partridge AS, Jones SLP (1996) GUM: a portable implementation of Haskell. In: PLDI'96 — programming language design and implementation, Philadelphia, PA, May 1996
42. Trinder PW, Loidl H-W, Pointon RF (2002) Parallel and distributed Haskells. *JFP*, 2002. Special issue on Haskell
43. Weber M (2000) hMPI — Haskell with MPI. WWW page, July 2000. <http://www-i2.informatik.rwth-aachen.de/~michaelw/hmpi.html>
44. Wegner P (1971) Programming languages, information structures and machine organisation. McGraw-Hill, New York

Futures

CORMAC FLANAGAN

University of California at Santa Cruz, Santa Cruz, CA, USA

Synonyms

[Eventual values](#); [Promises](#)

Definition

The programming language construct “future E” indicates that the evaluation of the expression “E” may proceed in parallel with the evaluation of the rest of the program. It is typically implemented by immediately returning a *future object* that is a proxy for the eventual value of “E.” Attempts to access the value of an undetermined future object block until that value is determined.

Discussion

Introduction

The construct “future E” permits the run-time system to evaluate the expression “E” in parallel with the rest of the program. To initiate parallel evaluation, the run-time system forks a parallel task that evaluates “E,” and also creates an object known as a *future object* that will eventually contain the result of that computation of “E.” The future object is initially *undetermined*, and becomes

determined or *resolved* once the parallel task evaluates “E” and updates (or resolves/fulfills/binds) the future object to contain the result of that computation.

Parallel evaluation arises because the future object is immediately returned as the result of evaluating “future E,” without waiting for the future object to become determined. Thus, the continuation (or enclosing context) of “future E” executes in parallel with the evaluation of “E” by the forked task.

The result of performing a primitive operation on a future object depends on whether that operation is considered strict or not. Some program operations, such as assignments, initializing a data structure, passing an argument to a procedure, returning a result from a procedure, are *non-strict*. These operations can manipulate (determined or undetermined) future objects in an entirely transparent fashion, and do not need to know the underlying value of that future object.

In contrast, *strict* program operations, such as addition, need to know the value of their arguments, and so cannot be directly applied to future objects. Instead, the arguments to these operations must first be converted to a regular (non-future) value, by *forcing* or *touching* that argument.

Forcing a future object that has been determined extracts its underlying value (which may in turn be a future and so the force operation must operate recursively). Forcing an undetermined future object blocks until that future is determined, and so force operations introduce synchronization between parallel tasks.

Implicit Versus Explicit Futures

A central design choice in a language with futures is whether forcing operations should be explicit in the source code, or whether they should be performed implicitly by strict operations, such as addition, that need to know the values of their argument. Explicit futures can be implemented as a library that introduces an additional data type. For example, the polymorphic type Future<T> could denote a future object containing values of type T. However, explicit futures require explicit forcing operations that are tedious and error-prone for programs that use futures heavily.

In contrast, implicit futures require language support since futures objects should be indistinguishable from their final resolved value. Implicit futures do

introduce additional overheads, since every strict primitive operation must now check whether its operands are futures.

As an optimization, a static analysis can be used to translate from a language with implicit futures into one with explicit futures. Essentially, the translation introduces explicit touch operations on those primitive operations that, according to a static analysis, may be applied to futures. In the context of the Gambit compiler, this optimization reduces the overhead of implicit futures from roughly 90% to less than 10% [3]. As an orthogonal optimization, a copying garbage collector could replace a determined future object by its underlying value, saving space and reducing the cost of subsequent touch operations.

Task Scheduling

Future annotations are often used to introduce parallel evaluation into recursive, divide-and-conquer style computations such as quicksort. In this situation, there may be exponentially more tasks than processors, and consequently task scheduling has a significant impact on overall performance. A naïve choice of round-robin scheduling would interleave the executions of all available tasks and result in an essentially *breadth-first* exploration of the computation tree, which may require exponentially more memory than an equivalent sequential execution.

Unfair scheduling policies significantly reduce this memory overhead, by better matching the number of concurrently active tasks to the number of processors. In the Multilisp implementation of this technique [1], each task has two possible states: *active* and *pending*, and each processor maintains a LIFO queue of pending tasks. When evaluating “future E,” the newly forked task evaluating “E” is marked active, while the parent task is moved to the pending queue. In the absence of idle processors, the parent task remains in the pending queue until the task evaluating “E” terminates, resulting in the same evaluation order (and memory footprint) as sequential execution, and the pending task queue functions much like the control stack of a sequential execution.

The benefit of the pending task queue is that if another processor becomes idle, it can *steal* and start evaluating one of these pending tasks. In this manner, the number of active tasks in the system is roughly

dynamically balanced with the number of available processors. Each processor typically has just one active task. Once that task terminates, it removes and makes active the next task in its pending queue. If the processor's pending task queue becomes empty, it tries to *steal* and activate a pending task from one of the other processors in the system.

An active task may block by trying to force an undetermined future object. In this situation, the task is typically put on a waiting list associated with that future. Once the future is later determined, all tasks on the waiting list are marked as pending, and will become active once an idle processor is available.

Lazy Task Creation

Lazy task creation [2, 5] provides a mechanism for reducing task creation overhead. Under this mechanism, the evaluation of “future E” does not immediately create a new task. Instead, the expression “E” is evaluated as normal, except that the control stack is marked to identify that the continuation of this future expression is implicitly a pending task.

Later, an idle processor can inspect this control stack to find this implicit pending task, create the appropriate future object, and start executing this pending task. The control stack is also modified so that after the evaluation of “E” terminates, its value is used to resolve that future object.

In the common case, however, the evaluation of “E” will terminate without the implicit pending task having been stolen by a different processor. In this situation, the result of “E” will be returned directly to the context of “future E” which still resides on the control stack, with little additional overhead and without any need to create a new task or future object.

Futures in Purely Functional Languages

Programs in purely functional languages offer numerous opportunities for executing program components in parallel. For example, the evaluation of a function application could spawn a parallel task for the evaluation of each argument expression. Applying such a strategy indiscriminately, however, leads to many parallel tasks whose overhead outweighs any benefits of parallel execution.

Futures provide a simple method for taming the implicit parallelism of purely functional programs. A programmer who believes that the parallel evaluation of some expression outweighs the overhead of creating a separate task may annotate the expression with the keyword “future.” In a purely functional language, these future annotations are *transparent* in that they have no effect on the final values computed by the program, and only influence the amount of parallelism exposed during the program’s computation. Consequently, purely functional programs with future are *deterministic*, just like their sequential counterparts.

F

Futures in Mostly Functional Languages

Futures have been used effectively in “mostly functional” languages such as Multilisp [1] and Scheme [2], where most computation is performed in a functional style and assignment statements are limited to situations where they provide increased expressiveness. In this setting, futures are *not transparent*, as they introduce parallelism that may change the order in which assignments are performed. To preserve determinism, the programmer needs to ensure there are no race conditions between parallel computations, possibly by introducing additional forcing operations that block until a particular task has terminated and all of its side effects have been performed.

Mostly functional languages include a side effect-free subset with substantial expressive power, and so the future construct is still effective for exposing parallelism within functional sub-computations written in these languages.

Deterministic Futures in Imperative Languages

Futures have been combined with transactional memory techniques to guarantee determinism even in the presence of side effects. Essentially, each task is considered to be a transaction, and these transactions must commit in a fixed, as-if-sequential order. Each transaction tracks its read and write sets. If one transaction accesses data that another transaction is mutating, then the later transaction is rolled back and restarted [8].

Futures in Distributed Systems

Future objects have been used in distributed systems to hide the latency associated with a remote procedure call (RPC). Instead of waiting for a message send and a matching response from a remote machine, an RPC can immediately return a future object as a proxy for the eventual result of that call. Note that this future could be returned even before the first RPC message is sent over the network. Consequently, if one machine performs multiple RPCs to the same remote machine, then these calls can all be batched into a single message; this technique is referred to as *promise pipelining*.

Thread-Specific Futures, Resolvable Futures, and I-vars

The programming language construct “future E” described above does not expose the ability to resolve the future to an arbitrary value; it can only be resolved to the result of evaluating the expression “E.”

An *I-var* is analogous to a future object in that it functions as a proxy for a value that may not yet be determined. The key difference is that an I-var does not have an explicit associated thread that is computing its value (as in the case for future objects). Instead, any thread can update or *resolve* the I-var. Thus, an I-var is essentially a single entry, one-shot queue. Attempts to access an I-var before it is determined block, as with future objects.

Somewhat confusingly, the terms “future” and “promise” are sometimes used to mean an I-var object that also exposes this *resolve* operation. The terms “thread-specific future” and “resolvable future” help disambiguate these distinct meanings.

Futures and Lazy Evaluation

Futures are closely related to lazy evaluation, in that both immediately return a placeholder for the result of a pending computation. A major difference is that future expressions typically must be evaluated before program termination, but lazy expressions need not be evaluated before program termination. This semantic difference means that preemptive evaluation of lazy expressions is considered speculative, whereas evaluation of future expressions is considered mandatory.

Sometimes, as in the language Alice ML, the term “lazy future” is used to denote a future whose expression

is evaluated lazily, that is, when the future object is first touched.

Futures and Exceptions

In the construct “future E,” an interesting design choice is how to handle any exceptions that are raised during the evaluation of “E,” since enclosing exception handlers from the context of “future E” may be no longer active. A more general version of this question is how the future construct should interact with call/cc (call-with-current-continuation). This question has been studied by a number of researchers (see, e.g., Moreau [10]), with the goal of providing determinism guarantees in the presence of these constructs.

Some languages take a more pragmatic choice, whereby if “E” raises an exception, then any attempt to touch the corresponding future object will also raise that exception.

Related Entries

► [Cilk](#)

Bibliographic Notes and Further Reading

Friedman and Wise introduced the term “promise” in 1976 [7]. Peter Hibbard described a form of explicit futures, called “eventual values,” in the context of Algol [6]. Baker and Hewitt [9] proposed implicit futures for the parallel evaluation of functional languages. Implicit futures were implemented in the Multilisp version of Scheme in the mid-1980s [1], and futures have since been adapted to a variety of other languages and compilers, including the Gambit Scheme compiler [2], Mul-T [5], Butterfly Lisp, Portable Standard Lisp, Act 1, Alice ML, Habanero Java, and many others.

Libraries supporting futures have been implemented for several languages, including (to name just a few of many examples) Java, C++0x, OCaml, python, Perl, and Ruby.

The single-assignment I-var structure originated in Id [12] and was included in subsequent languages such as parallel Haskell and Concurrent ML.

Bibliography

1. Halstead R (Oct 1985) Multilisp: A language for concurrent symbolic computation. ACM Trans Program Lang Syst 7(4):501–538

2. Feeley M (1993) An efficient and general implementation of futures on large scale shared-memory multiprocessors. PhD thesis, Department of Computer Science, Brandeis University
3. Flanagan C, Felleisen M (1995) The semantics of future and its use in program optimization. In: Proceedings of the ACM SIGPLAN-SIGACT symposium on the principles of programming languages. POPL'95, pp 209–220
4. Moreau L (1994) Sound evaluation of parallel functional programs with first-class continuations. PhD thesis, Universite de Liege
5. Kranz D, Halstead R, Mohr E (1989) Mul-T: A high performance parallel lisp. In: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation, Portland, pp 81–90
6. Knueven P, Hibbard P, Leverett B (June 1976) A language system for a multiprocessor environment. In: Proceedings of the 4th international conference on design and implementation of algorithmic languages, Courant Institute of mathematical Studies, New York, pp 264–274
7. Friedman DP, Wise DS (1976) The impact of applicative programming on multiprocessing. 1976 International conference on parallel processing, pp 263–272
8. Welc A, Jagannathan S, Hosking A (2005) Safe futures for Java. In: Proceedings of the 20th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications (OOPSLA 2005), ACM, pp 439–453
9. Baker H, Hewitt C (Aug 1977) The incremental garbage collection of processes. In: Proceedings of symposium on AI and programming languages, ACM SIGPLAN Notices 12(8): 55–59
10. Moreau L (1996) The semantics of scheme with future. In: Proceedings of the first ACM SIGPLAN international conference on functional programming, pp 146–156
11. Mohr E, Kranz D, Halstead R (1990) Lazy task creation: a technique for increasing the granularity of parallel programs. In: Proceedings of the 1990 ACM conference on LISP and functional programming, pp 185–197
12. Arvind, Nikhil RS, Pingali KK (1989) I-Structures: Data structures for parallel computing. ACM Trans Program Lang Syst 11(4):598–632

G

GA

- Global Arrays Parallel Programming Toolkit

Gather

- Collective Communication

Gather-to-All

- Allgather

Gaussian Elimination

- Dense Linear System Solvers
- Sparse Direct Methods

GCD Test

- Banerjee's Dependence Test
- Dependences

Gene Networks Reconstruction

- Systems Biology, Network Inference in

Gene Networks Reverse-Engineering

- Systems Biology, Network Inference in

Generalized Meshes and Tori

- Hypercubes and Meshes

Genome Assembly

ANANTH KALYANARAMAN
Washington State University, Pullman, WA, USA

Synonyms

Genome sequencing

Definition

Genome assembly is the computational process of deciphering the sequence composition of the genetic material (DNA) within the cell of an organism, using numerous short sequences called *reads* derived from different portions of the target DNA as input. The term *genome* is a collective reference to all the DNA molecules in the cell of an organism. *Sequencing* generally refers to the experimental (wetlab) process of determining the sequence composition of biomolecules such as DNA, RNA, and protein. In the context of genome assembly, however, the term is more commonly used to refer to the experimental (wetlab) process of generating reads from the set of chromosomes that constitutes the genome of an organism. Genome *assembly* is the computational step that follows sequencing with the objective of reconstructing the genome from its reads.

Discussion

Introduction

Deoxyribonucleic acid (or DNA) is a double-stranded molecule which forms the genetic basis in most known organisms. The DNA along with other molecules, such as the ribonucleic acid (or RNA) and proteins, collectively constitute the subject of study in various

branches of biology such as molecular biology, genetics, genomics, proteomics, and systems biology. A DNA molecule is made up of two equal length strands with opposite directionality, with the ends labeled from 5' to 3' on one and 3' to 5' on the other. Each strand is a sequence of smaller molecules called *nucleotides*, and each nucleotide contains one of the four possible nitrogenous bases – adenine (a), cytosine (c), guanine (g), and thymine (t). Therefore for computational purposes, each strand can be represented in the form of a string over the alphabet {a, c, g, t}, expressed always in the direction from 5' to 3'. Furthermore, the base at a given position in one strand is related to the base at the corresponding position in the other strand by the following base-pairing rule (referred to as "base complementarity"): $a \leftrightarrow t, c \leftrightarrow g$. Therefore, the sequence of one strand can be directly deduced from the sequence of the other. For example, if one strand is 5' agaccaggtaac 3', then the other is 5' gtaactggct 3'.

The length of a genome is measured in the number of its base pairs ("bp"). Genomes range in length from being just a few million base pairs in microbes to several billions of base pairs in many eukaryotic organisms. However, all sequencing technologies available till date, since the invention of Sanger sequencing in late 1970s, have been limited to accurately sequencing DNA molecules no longer than ~1 Kbp. Consequently, scientists have had to deploy alternative sequencing strategies for extending the reach of technology to genome scale. The most popular strategy is the *whole genome shotgun* (or WGS) strategy, where multiple copies of a single long target genome are shredded randomly into numerous fragments of sequenceable length and the corresponding reads sequenced individually using any standard technology. Another popular albeit more expensive strategy is the hierarchical approach, where a library of smaller molecules called Bacterial Artificial Chromosomes (or BACs) is constructed. Each BAC is ~150 Kbp in length and they collectively provide a minimum tiling path over the entire length of the genome. Subsequently, the BACs are sequenced individually using the shotgun approach.

In both approaches, the information of the relative ordering among the sequenced reads is lost during sequencing either completely or almost completely. Therefore, the primary information that genome assemblers should rely upon is the end-to-end sequence

overlap preserved between reads that were sequenced from overlapping regions along the target genome. To increase the chance of overlap, the target genome is typically sequenced in a redundant fashion. This is referred to as *genome coverage*. A higher coverage typically tends to provide information for a more accurate assembly, although at increased costs of generation and analysis. The fact that reads could have originated from an arbitrary strand adds another dimension to the complexity of the reconstruction process. The process is further complicated by other factors such as errors introduced during read sequencing and the presence of genomic repeats that could ambiguously overlap detection and read placement. To partially aid the resolving of the genomic repeat regions, sequencing is sometimes performed in pairs from clonal insert libraries, where the genomic distance between the two reads of a pair can be estimated, typically in the 1–3 Kbp range. This technique is called *pair-end sequencing* and genome assemblers could take advantage of this information to resolve repeats that are smaller than the specified range.

Genome assembly is a classical computational problem in the field of bioinformatics and computational biology. More than two decades of research has yielded a number of approaches and algorithms and yet the problem continues to be actively pursued. This is because of several reasons. While there are different ways of formulating the genome assembly problem, all known formulations are NP-Hard. Therefore, researchers continue to work on developing improved, efficient approximation and heuristic algorithms. However, even the best serial algorithms take tens of thousands of CPU hours for eukaryotic genomes owing to their: large genomic sizes, which increase the number of reads to be analyzed; and high genomic repeat complexity, which adds substantial processing overheads. For instance, in the Celera's WGS version of the human genome assembly published in 2001, over 27 million reads representing 5.11x coverage over the genome were assembled in about 10,000 CPU h.

A paradigm shift in sequencing technologies has further aggravated the data-intensiveness of the problem and thereby the need for continued algorithmic development. Until the mid-2000s, the only available technology for use in genome assembly projects was Sanger sequencing, which produces reads of approximate length 1 Kbp each. Since then, a slew of

high-throughput sequencing technologies, collectively referred to as the *next-generation sequencing technologies*, have emerged, significantly revitalizing the sequencing community. Examples of next-generation technologies include Roche 454 Life Sciences system's pyrosequencing (read length ~400 bp), Illumina Genome Analyzer and HiSeq (read length ~35–125 bp); Life Technologies SOLiD (read length ~50 bp) and Helicos HeliScope (read length ~35 bp). While these instruments generate much shorter reads than Sanger, they do so at a much faster rate effectively producing several hundred millions of reads in a single experiment, and at significantly reduced costs (about 30–40 times cheaper). These attractive features are essentially democratizing the sequencing process and broadening community contribution to sequenced data. From a genome assembly perspective, a shorter read length could easily deteriorate the assembly quality because the reads are more likely to exhibit false or insufficient overlaps. To offset this shortcoming, sequencing is required at a much higher coverage (30x–200x) than for Sanger sequencing. The higher coverage has another desirable effect in that, because of its built-in redundancy, it could aid in a more reliable identification of real genomic variations and their differentiation from experimental artifacts. Detecting genomic variations such as single nucleotide polymorphisms (SNPs) is of prime importance in comparative and population genomics.

This combination of high coverage and short read lengths makes the short read genome assembly problem significantly more data-intensive than for Sanger reads. For instance, any project aiming to reconstruct a mammalian genome (e.g., human genome) *de novo* using any of the current next-generation technologies would have to contend with finding a way to assemble several billion short reads. This changing landscape in technology and application has led to the development of a new generation of short read assemblers. Although traditional developmental efforts have been targeting serial computers, the increasing data-intensiveness and the increasing complexity of genomes being sequenced have gradually pushed the community toward parallel processing, for what has now become an active branch within the area.

In what follows, an overview of the assembly problem, with its different formulations and algorithmic solutions is presented. This entry is not intended to be

a survey of tools for genome assembly. Rather, it will focus on the parallelism in the problem and related efforts in parallel algorithm development. In order to set the stage, the key ideas from the corresponding serial approach will also be presented as necessary. The bulk of the discussion will be on *de novo assembly*, which is typically the harder task of reconstructing a genome from its reads assuming no prior knowledge about the sequence of the genome except for its reads and as available, their pair-end sequencing information.

Algorithmic Formulations of Genome Assembly

The genome assembly problem: Let $R = \{r_1, r_2 \dots r_m\}$ denote a set of m reads sequenced from an unknown target genome G of length g . The problem of genome assembly is to reconstruct the sequence of genome G from R .

As a caveat, for nearly all input scenarios, the outcome expected is not a single assembled sequence but a *set of assembled sequences* for a couple of reasons. Typically, the genome of a species comprises of multiple chromosomes (e.g., 23 pairs in the human genome) and therefore each chromosome can be treated as an individual sequence target. Note that, however, the information about the source chromosome for a read is lost during a sequencing process such as WGS and it is left for the assembler to detect and sort these reads by chromosomes. Furthermore, any shotgun sequencing procedure tends to leave out “gaps” along the chromosomal DNA during sampling, and therefore it is possible to reconstruct the genome only for those sampled sections. It is expected that, through incorporation of pair-end information, at least a subset of these assembled products (called “contigs” in genome assembly parlance) can be partially ordered and oriented.

Notation and terminology: Let s denote a sequence over a fixed alphabet Σ . Unless otherwise specified, a DNA alphabet is assumed – i.e., $\Sigma = \{a, c, g, t\}$. Let $|s|$ denote the length of s ; and $s[i \dots j]$ denote the substring of s starting and ending at indices i and j respectively, with the convention that string indexing starts at 1. A *prefix* i (alternatively, *suffix* i) of s is $s[1 \dots i]$ (alternatively, $s[i \dots |s|]$). Let $n = \sum_{i=1}^m |r_i|$ and $\ell = \frac{n}{m}$. The sequencing coverage c is given by $\frac{g}{n}$. The terms *string* and *sequence* are used interchangeably. Let p denote the number of processors in a parallel computer.

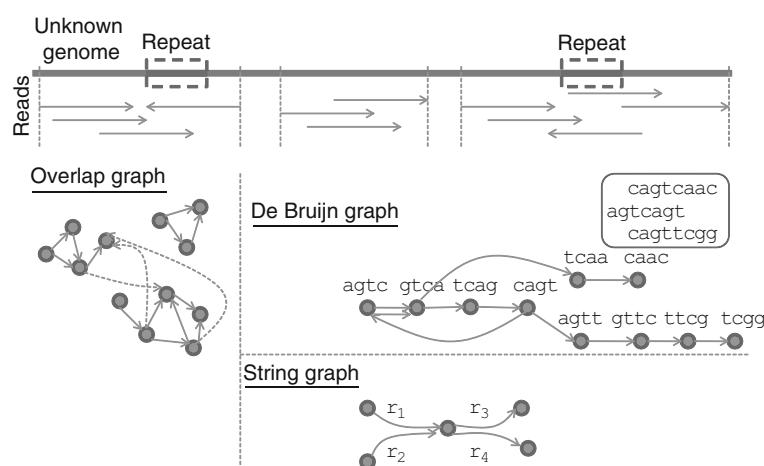
The most simplistic formulation of genome assembly is that of the *Shortest Superstring Problem* (SSP). A *superstring* is a string that contains each input read as a substring. The SSP formulation is NP-complete. Furthermore, its assumptions are not realistic in practice. Reads can contain sequencing errors and hence they need not always appear preserved as substrings in the genome; and genomes typically are longer than the shortest superstring due to presence of repeats and nearly identical genic regions (called paralogs).

Among the more realistic models for genome assembly, graph theoretic formulations have been in the forefront. In broad terms, there are three distinct ways to model the problem (refer to Fig. 1):

(1) *Overlap graph*: Construct graph $G(V, E)$ from R , where each read is represented by a unique vertex in V and an edge is drawn between (r_i, r_j) iff there is a significant suffix–prefix overlap between r_i and r_j . *Overlap* is typically defined using a pairwise sequence alignment model (e.g., semi-global alignment) and related metrics to assess the quality of an alignment. Given G , the genome assembly problem can be reduced to the problem of finding a Hamiltonian Path, if one exists, provided that there were no breaks (called sequencing gaps) along the genome during sequencing. If there

were gaps during sequencing, then one Hamiltonian Path is sought for every connected component in G (if it exists) and the genome can be recovered as an unordered set of pieces corresponding to the sequenced sections. This formulation using the overlap graph model has been shown to be NP-Hard.

- (2) *De Bruijn graph*: Let a k -mer denote a string of length k . Construct a De Bruijn graph, where the vertex set is the set of all k -mers contained inside the reads of R ; and a directed edge is drawn from vertices i to j , iff the $k - 1$ length suffix of the k -mer i is identical to the $k - 1$ length prefix of the k -mer j . Put another way, each edge in E represents a unique $k + 1$ -mer that is found in at least one of the reads in R . Given a De Bruijn graph G , genome assembly is the problem of finding a shortest Euler tour in which each read is represented by a sub-path in the tour. While finding an Euler tour is polynomially solvable, the optimization problem is still NP-Hard by reduction from SSP.
- (3) *String graph*: This model is a variant of the overlap graph in which the edges represent reads, and vertices represent branching of end-to-end overlaps of the adjoining reads. For example, if a read r_i overlaps with both r_j and r_k then it is represented by a vertex



Genome Assembly. Fig. 1 Illustration of the three different models for genome assembly. The top section of the figure shows a hypothetical unknown genome and the reads sequenced from it. The dotted box shows a repetitive region within the genome. In the overlap graph, solid lines indicate true overlaps and dotted lines show overlaps induced by the presence of the repeat. The De Bruijn graph shows the graph for an arbitrary section involving three reads. The string graph shows an example of a branching vertex with multiple possible entry and exit paths

which has an inbound edge corresponding to r_i and two outbound edges representing r_j and r_k . Furthermore, the graph is pruned off transitively inferable edges – e.g., if both r_i and r_j overlaps with r_k , whereas r_i also overlaps with r_j , then the overlap from r_j to r_k is inferable and is therefore removed. Given a string graph G , the assembly problem becomes one that is equivalent of finding a constrained least cost Chinese Postman tour of G . This formulation has also been shown to be NP-Hard.

All the three formulations incorporate features to handle sequencing errors. In the overlap graph model, the pairwise alignment formulation used to define edges automatically takes into account character substitutions, insertions, and deletions. In the other two models, an “error correction” procedure is run as a preprocessing step to mask off anomalous portions of the graph prior to performing the assembly tours.

Parallelization for the Overlap Graph Model

Algorithms that use the overlap graph model deploy a three-stage assembly approach of *overlap-layout-consensus*. In the first stage, the overlap graph is constructed by performing all-against-all pairwise comparison of the reads in R . As a result, a layout for the assembly is prepared using the pairwise overlaps and in the third stage, a multiple sequence alignment (MSA) of the reads is performed as dictated by the layout. Given the large number of reads generated in a typical sequencing project, the overlap computation phase dominates the run-time and hence is the primary target for parallelization and performance optimization.

A brute-force way of implementing this step will be to compare all $\binom{m}{2}$ possible pairs. This approach can also be parallelized easily, as individual alignment tasks can be distributed in a load-balanced fashion given the uniformity expected in the read lengths. However, an implementation may not be practically feasible for larger number of reads due to the quadratic increase in alignment workload. Each alignment task executed serially could take milliseconds of run-time. In fact, for the assembly problem, the nature of sampling done during sequencing guarantees that performing all-against-all comparisons is destined to be highly wasteful. This is because of the random shotgun approach to sequencing

which is expected to sample roughly equal number of reads covering each genomic base and it is typically only reads that originate from the same locus that tend to overlap with one another, with the exception of reads from repetitive regions of the genome.

A more scalable approach to detect pairwise overlaps is to first shortlist pairs of sequences based on the presence of sufficiently long exact matches (i.e., a filter built on a necessary-but-not-a-sufficient condition) and then compute pairwise comparisons only on them. Methods vary in the manner in which these “promising pairs” are shortlisted. One way of generating promising pairs is to use a string lookup table data structure that is built to record all k -mer occurrences within the reads in linear time. Sequence pairs that share one or more k -mers can be subsequently considered for alignment-based evaluation. While this approach supports a simple implementation, it poses a few notable limitations. The size complexity of the lookup table data structure contains an exponential term $O(|\Sigma|^k)$. This restricts the value of k in practice; for DNA alphabet, physical memory constraints demand that $k \leq 15$. Whereas, reads could share arbitrarily long matches, especially if the sequencing error rates are low. Another disadvantage of the lookup table is that it is only suited to detect pairs with fixed length matches. An exact match of an arbitrary length q will reveal itself as $q - k + 1$ smaller k -mer matches, thus increasing computation cost. Furthermore, the distribution of entries within the lookup table is highly data dependent and could scatter the same sequence pair in different locations of the lookup table, making it difficult for parallelization on distributed memory machines. Methods that use lookup tables simply use a high end shared memory machine for serial generation and storage, and focus on parallelizing only the pairwise sequence comparison step.

An alternative albeit more efficient way of enumerating promising pairs based on exact matches is to use a string indexing data structure such as suffix tree or suffix array that will allow detection of *variable* length matches. A match α between two reads r_i and r_j is called *left-maximal* (alternatively, *right-maximal*) if the characters preceding (alternatively, following) it in both strings, if any, are different. A match α between reads r_i and r_j is said to be a *maximal match* between the two reads if it is both left- and right-maximal. Pairs containing maximal matches of a minimum length cutoff, say

ψ , can be detected in optimal run-time and linear space using a suffix tree data structure. A generalized suffix tree of a set of strings is the compacted trie of all suffixes in the strings. The pair enumeration algorithm first builds a generalized suffix tree over all reads in R and then navigates the tree in a bottom-up order.

Parallel Generalized Suffix Tree Construction

For constructing a suffix tree, there are several linear time construction serial algorithms and optimal algorithms for the CRCW-PRAM model. However, optimal construction under the distributed memory machine model, where memory per processor is assumed to be too small to store the input sequences in R , is an open problem. Of the several approaches that have been studied, the following approach is notable as it has demonstrated linear scaling to thousands of processors on distributed memory supercomputers.

The major steps of the underlying algorithm are as follows. Each step is a parallel step. The steps marked as *comm* are communication-bound and the steps marked *comp* are computation-bound.

S1. (*comp*) Load R from I/O in a distributed fashion such that each processor receives $O\left(\frac{n}{p}\right)$ characters and no read is split across processor boundaries.

S2. (*comp*) Slide a window of length $k \leq \psi$ over the set of local reads and bucket sort locally all suffixes of reads based on their k length prefix. Note that for DNA alphabet, even a value as small as 10 for k is expected to create over a million buckets, sufficient to support parallel distribution. An alternative to local generation of buckets is to have a master-worker paradigm where the reads are scanned in small-sized batches and have a dedicated master distribute batches to workers in a load-balanced fashion.

S3. (*comm*) Parallel sort the buckets such that each processor receives approximately $\sim \frac{n}{p}$ suffixes and no bucket is split across processor boundaries. While the latter cannot be theoretically guaranteed, real-world genomic data sets typically distribute the suffixes uniformly across buckets thereby virtually ensuring that no bucket exceeds the $O\left(\frac{n}{p}\right)$ bound. If a bucket size becomes too big to fit in the local memory, then an alternative strategy can be deployed

whereby the prefix length for bucketing is iteratively extended until the size of the bucket fits in the local memory.

S4. (*comm, comp*) Each bucket in the output corresponds to a unique subtree in the generalized suffix tree rooted exactly k characters below the root. The subtree corresponding to each bucket is then locally constructed using a recursive bucket sort-based method that compares characters between suffixes. However, this step cannot be done strictly locally because it needs the sequences of reads whose suffixes are being compared. This can be achieved by building the local set of subtrees in small-sized batches and performing one round of communication using *Alltoallv()* to load the reads required to build each batch from other processors. In order to reduce the overhead due to read fetches, observe that the aggregate memory on a relatively small group of processors is typically sufficient to accommodate the read set for most inputs in practice. For example, even a set of 100 million reads each of length 1Kbp needs only of the order of 100 GB. Even assuming 1 GB per processor, this means the read set will fit within 100 processors. Higher number of processors is required only to speedup computation. One could take advantage of this observation by partitioning the processor space into subgroups of fixed, smaller size, determined by the input size, and then have the on-the-fly read fetches to seek data from processors from within each subgroup. This improved scheme could distribute the load of any potential hot spots and also the overall communication could be faster as the collective transportation primitive now operates on a smaller number of processors.

The above approach can be implemented to run in $O\left(\frac{n\ell}{p}\right)$ time and $O\left(\frac{n}{p}\right)$ space, where ℓ is the mean read length. The output of the algorithm is a distributed representation of the generalized suffix tree, with each processor storing roughly $\frac{n}{p}$ suffixes (or leaves in the tree).

The *cluster-then-assemble* Approach

A potential stumbling block in the overlap-layout-consensus approach is the large amount of memory required to store and retrieve the pairwise overlaps so that they can be used for generating an assembly layout. For genomes which have been sequenced at a uniform

coverage c , the expectation is that each base along the genome is covered by c reads on average, thereby implying $\binom{c}{2}$ overlapping read pairs for every genomic base. However, this theoretical linear expectation holds for only a fraction of the target genome, whereas factors such as genomic repeats and oversampled genic regions could increase the number of overlapping pairs arbitrarily beyond the expected level. An example case in point is the gene-enriched sequencing of the highly repetitive ~2.5 billion bp maize genome.

One way to overcome this scalability bottleneck is to use clustering as a preprocessing step to assembly. This approach, referred to as *cluster-then-assemble*, builds upon the assumption that any genome-scale sequencing effort is likely to undersample the genome, leaving out sequencing gaps along the genome length and thereby allowing the assemblers to reconstruct the genome in pieces – one piece for every contiguous stretch of the genome (aka “contig” or “genomic island”) that is fully sampled through sequencing. This assumption is not unrealistic as tens of thousands of sequencing gaps have always resulted in almost all eukaryotic genomes sequenced so far using the WGS approach. The cluster-then-assemble takes advantage of this assumption as follows: The set of m reads is first partitioned into groups using a single-linkage transitive closure clustering method, such that there exists a sequence of overlapping pairs connecting every read to every other read in the same cluster. Ideally, each cluster

should correspond to one genomic island, although the presence of genomic repeats could collapse reads belonging to different islands together. Once clustered, each cluster represents an independent subproblem for assembly, thereby breaking a large problem with m reads into numerous, disjoint subproblems of significantly reduced size small enough to fit in a serial computer. In practice, tens of thousands of clusters are generated making the approach highly suited for trivial parallelization after clustering.

The primary challenge is to implement the clustering step in parallel, in a time- and space-efficient manner. In graph-theoretic terms, the output produced by a transitive closure clustering is representative of connected components in the overlap graph, thereby allowing the problem to be reduced to one of connected component detection. However, generating the entire graph G prior to detection would contradict the purpose of clustering from a space complexity standpoint.

Figure 2 outlines an algorithm to perform sequence clustering. The algorithm can be described as follows: Initialize each read in a cluster of its own. In an iterative process, generate promising pairs based on maximal matches in a non-increasing order of their lengths using suffix trees (as explained in the previous section). Before assigning a promising pair for further alignment processing, a check is made to see whether the constituent reads are in the same cluster. If they are part of the same cluster already, then the pair is discarded; otherwise it

Algorithm 1 Read Clustering

```

Input: Read set  $R = \{r_1, r_2, \dots, r_m\}$ 
Output: A partition  $C = \{C_1, C_2, \dots, C_k\}$  of  $R$ ,  $1 \leq k \leq m$ 

1. Initialize Clusters:
    $C \leftarrow \{\{r_i\} \mid 1 \leq i \leq m\}$   $\Rightarrow (\text{master})$ 
2. FOR each pair  $(r_i, r_j)$  with a maximal match of length  $\geq \psi$ 
   generated in non-increasing order of maximal match length  $\Rightarrow (\text{worker})$ 
    $C_p \leftarrow \text{Find}(r_i)$   $\Rightarrow (\text{master})$ 
    $C_q \leftarrow \text{Find}(r_j)$   $\Rightarrow (\text{master})$ 
   IF  $C_p \neq C_q$  THEN  $\Rightarrow (\text{master})$ 
     overlap quality  $\leftarrow \text{Align}(r_i, r_j)$   $\Rightarrow (\text{worker})$ 
     IF overlap quality is significant THEN  $\Rightarrow (\text{master})$ 
        $\text{Union}(C_p, C_q)$   $\Rightarrow (\text{worker})$ 
3. Output  $C$   $\Rightarrow (\text{master})$ 
```

Genome Assembly. Fig. 2 Pseudocode for a read clustering algorithm suited for parallelization based on the master-worker model. The site of computation is shown in brackets. Operations on the set of clusters are performed using the Union-Find data structure

is aligned. If the alignment results show a satisfactory overlap (based on user-defined alignment parameters), then merge the clusters containing both reads into one larger cluster. The process is repeated until all promising pairs are exhausted or until no more merges are possible. Using the union–find data structure would ensure the *Find* and *Union* calls to run in amortized time proportional to the Inverse Ackerman function – a small constant for all practical inputs.

There are several advantages to this clustering strategy. Clustering can be achieved in at most $m - 1$ merging steps. Checking if a read pair is already clustered before alignment is aimed at reducing the number of pairs aligned. Generating promising pairs in a non-increasing order of their maximal match lengths is a heuristic that could help identify pairs that are more likely to succeed the alignment test sooner during execution. Furthermore, promising pairs are processed as they are generated, obviating the need to store them. This coupled with the use of the suffix tree data structure implies an $O(n)$ serial space complexity for the clustering algorithm.

The parallel algorithm can be implemented using a master–worker paradigm. A dedicated master processor can be responsible for initializing and maintaining the clusters and also for distributing alignment workload to the workers in a load-balanced fashion. The workers at first can generate a distributed representation of the generalized suffix tree in parallel (as explained in the previous section). Subsequently, they can generate promising pairs from their local portion of the tree and send them to the master. To reduce communication overheads, pairs can be sent in arbitrary-sized batches as demanded by the situation of the work queue buffer at the master. The master processor can check the pairs against the current set of clusters, filter out pairs that do not need alignment, and add only those pairs that need alignment to its work queue buffer. The pairs in the work queue buffer can be redistributed to workers in fixed size batches, to have the workers compute alignments and send back the results. Communication overheads can be masked by overlapping alignment computation with communication waits using non-blocking calls. The PaCE software suite implements the above parallel algorithm, and it has demonstrated linear scaling to thousands of processors on a distributed memory supercomputer.

Short Read Assembly

The landscape of genome assembly tools has significantly transformed itself over the last few years with the advent of next-generation sequencing technologies. A plethora of new-generation assemblers that collectively operate under the banner of “short read assemblers” is continuing to emerge. Broadly speaking, these tools can be grouped into two categories: (1) those that follow or extend from the classical overlap graph model; and (2) those that deploy either the De Bruijn graph or the string graph formulation. The former category includes programs such as Edena, Newbler, PE-Assembler, QSRA, SHARCGS, SSAKE, and VCAKE. The programs that fall under the latter category are largely inspired by two approaches – the De Bruijn graph formulation used in the EULER assembler; and the string graph formulation proposed by Myers. As of this writing, these programs include EULER-SR, ALLPATHS, Velvet, SOAPdenovo, ABySS, and YAGA. Either of these lists is likely to expand in the coming years, as new implementations of short read assemblers are continuing to emerge as are new technologies.

In principle, the techniques developed for assembling Sanger reads do not suit direct application for short read assembly due to a combination of factors that include shorter read length, higher sequencing coverage, an increased reliance on pair-end libraries, and idiosyncrasies in sequencing errors.

A shorter read length implies that the method has to be more sensitive to differences to avoid potential mis-assemblies. This also means providing a robust support for incorporating the knowledge available from pair-end libraries. In case of next-generation sequencing, pair-end libraries are typically available for different clone insert sizes, which translates to a need to implement multiple sets of distance constraints.

A high sequencing coverage introduces complexity at two levels. The number of short reads to assemble increases linearly with increased coverage, and this number can easily reach hundreds of millions even for modest-sized genomes because of shorter read length. For instance, a mid-size genome such as that of *Arabidopsis* (~115 Mbp) or fruit fly (~120 Mbp) sequenced at 100x coverage either using Illumina or SOLiD would generate a couple of hundred million reads. Secondly, the average number of overlapping read

pairs at every given genomic location is expected to grow quadratically ($\propto \binom{c}{2}$) with the coverage depth (c). This particularly affects the time and memory scalability of methods that use the overlap graph model not only because they operate at the level of pairwise read overlaps, but also because many tools assume that such overlaps can be stored and retrieved from local memory. Consequently, such methods take between 6–10 h and several gigabytes of memory even for assembling small bacterial genomes. From a parallelism perspective, a high coverage in sampling could also potentially mean fewer sequencing gaps, as the probability of a genomic base being captured by a read improves with coverage by the Lander–Waterman model. Therefore, divide-and-conquer-based techniques such as the cluster-then-assemble may not be as effective in breaking the initial problem size down.

Short read assemblers also have to deal with technology specific errors. The error rates associated with next-generation technologies are typically in the 1–5% range, and cannot be ignored as differentiating them from real differences could be key to capturing natural variations such as near identical paralogs.

In the current suite of tools specifically built for short read assembly, only a handful of tools support some degree of parallelism. These tools include PE-Assembler, ABySS, and YAGA. Others are serial tools that work on desktop computers and rely on high-memory nodes for larger inputs. Even the tools that support parallelism do so to varying degrees. PE-Assembler, which implements a variant of the overlap graph model, limits parallelism to node level and assumes that the number of parallel threads is small (< 10). It deploys a “seed-extend” strategy in which a subset of “reliable” reads are selected as seeds and other reads that overlap with each seed are incrementally added to extend and build into a consensus sequence in the 3' direction. Parallelism is supported by selecting multiple seeds and launching multiple threads that assume responsibility of extending these different reads in parallel. While the algorithm has the advantage of not having to build and manage large graphs, it performs a conservative extension primarily relying on pair-end data to collapse reads into contigs. From a parallelism perspective, the algorithm relies on shared memory access for managing the read set, which works well if the number of threads is very small. Furthermore, not

all steps in the method are parallel and some steps are disk-bound. Experimental results show that the parallel efficiency drops drastically beyond three threads.

ABySS and YAGA are two parallel methods that implement the De Bruijn graph model and work for distributed memory computers. The De Bruijn and string graph formulations are conceptually better suited for short read assembly. These formulations allow the algorithm to operate at a read or read's subunit (i.e., k -mer) level rather than the pairwise overlap level. Repetitive regions manifest themselves in the form of special graph patterns, and error correction mechanism could detect and reconcile anomalous graph substructures prior to performing assembly tours, thereby reducing the chance of misassemblies. The assembly itself manifests in the form of graph traversals. Constructing these graphs and traversing them to produce assembly tours (although not guaranteed to be optimal due to intractability of the problem) are problems with efficient heuristic solutions on a serial computer. The methods ABySS and YAGA provide two different approaches to implement the different stages in parallel using De Bruijn graphs. While these methods differ in their underlying algorithmic details and in the degree offered for parallelism, the structural layout of their algorithms is similar consisting of these four major steps: (1) parallel graph construction; (2) error correction; (3) incorporation of distance constraints due to pair-end reads; and (4) assembly tour and output.

In what follows, approaches to parallelize each of these major steps are outlined, with the bulk of exposition closely mirroring the algorithm in YAGA because its algorithm more rigorously addresses parallelization for all the steps, and takes advantage of techniques that are more standard and well understood among distributed memory processing codes (e.g., sorting, list ranking). By contrast, parallelization is described only for the initial phase of graph construction in ABySS. Therefore, where appropriate, variations in the ABySS algorithmic approach will be highlighted in the text. Other than differences in their parallelization strategies, the two methods also differ particularly in the type of De Bruijn graph they construct (directed vs. bidirected) and their ability to handle multiple read lengths. Such details are omitted in an attempt to keep the text focused on the parallel aspects of the problem. An interested reader can refer to the individual papers for details

pertaining to the nuances of the assembly procedure and the output quality of these assemblies.

Parallel De Bruijn Graph Construction and Compaction

Given m reads in set R , the goal is to construct in parallel a distributed representation of the corresponding De Bruijn graph built out of k -mers, for a user-specified value of k . Note that, once a De Bruijn graph representation is generated, it can be transformed into a corresponding string graph representation by compressing paths whose label when concatenated spells out the characters in a read, and by accordingly introducing branch nodes that capture overlap continuation between adjoining reads. This can be done in a successive stage of graph compaction, a minor variant of which is described in the later part of this section. The computing model assumed is p processors (or equivalently, processes), each with access to a local RAM, and connected through a network interconnect and with access to a shared file system where the input is made available.

To construct the De Bruijn graph, the reads in R are initially partitioned and loaded in a distributed manner such that each processor receives $O\left(\frac{n}{p}\right)$ input characters. Let R_i refer to the subset of reads in processor p_i . Through a linear scan of the reads in R_i , each p_i enumerates the set of k -mers present in R_i . However, instead of storing each such k -mer as a vertex of the De Bruijn graph, the processor equivalently generates and stores the corresponding edges connecting those vertices in the graph. In other words, there is a bijection between the set of edges and the set of distinct $k + 1$ -mers in R_i . In this edge-centric representation, the vertex information connecting each edge is implicit and the count of reads containing a given $k + 1$ -mer is stored internally at that edge. Note that after this generation process, the same edge could be potentially generated at multiple processor locations. To detect and merge such duplicates, the algorithm simply performs a *parallel sort* of the edges using the $k + 1$ -mers as the key. Therefore, in one sorting step, a distributed representation of the De Bruijn graph is constructed. Standard parallel sort routines such as sample sort can be used here to ensure even redistribution of the edges across processors due to sorting.

An alternative to this edge-centric representation is a vertex-centric representation, which is followed in ABySS and in an earlier version of YAGA. Here, the k -mer set corresponding to each R_i is generated by the corresponding p_i . The vertices adjacent to a given vertex could be generated remotely by one or more processors. Therefore, this approach necessitates processors to communicate with one another in order to check the validity of all edges that could be theoretically drawn from its local set of vertices. While the number of such edge validation queries is bounded by 8 per vertex ($\{a, c, g, t\}$ on either strand orientation), the method runs the risk of generating false edges, e.g., if a $k - 1$ -mer, say α , occurs in exactly two places along the genome as aac and gat , then the vertex-centric approach will erroneously generate edges for aat and gac , which are $k + 1$ -mers nonexistent in the input. Besides this risk, care must be taken to guarantee an even redistribution of vertices among processors by the end of the construction process. For instance, a static allocation scheme in which a hash function is used to map each k -mer to a destination processor, as it is done in ABySS, runs the risk of producing unbalanced distribution as the k -mer concentration within reads is input dependent.

Graph compaction: Once a distributed edge-centric representation of the De Bruijn graph is generated, the next step is to simplify it by identifying maximal segments of simple paths (or “chains”) and compact each of them into a single longer edge. Edges that belong to a given chain could be distributed and therefore this problem of removing chains becomes a two-step process: First, to detect the edges (or equivalently, the vertices) which are part of chains and then perform compaction on the individual chains. To mark the vertices that are part of some chain, assume without loss of generality that each edge is stored twice as $< u, v >$ and $< v, u >$. *Sorting* the edges in parallel by their first vertex it would bring all edges incident on a vertex together on some processor. Only those vertices that have a degree of two can be part of chains and such vertices can be easily marked with a special label. In the next step, the problem of compacting the vertices along each chain can be treated as a variant of *segmented parallel list ranking*. A distributed version of the list ranking algorithm can be used to calculate the relative distances of each marked vertex from the start and end of its respective chain. The same procedure can also be used to determine the

vertex identifiers of those two boundary vertices for each marked vertex. Compaction then follows through the operations of concatenating the edge labels along the chain at one of the terminal edges, removing all internal edges, and aggregating an average $k + 1$ -mer frequency of the constituent edges along the chain. All of these operators are binary associative to allow being implemented using calls to a segmented parallel prefix routine. The output of this step is a distributed representation of the compacted graph.

Error Correction and Variation Detection

In the De Bruijn graph representation, errors due to sequencing manifest themselves as different subgraph motifs which could be detected and pruned. For ease of exposition, let us informally call an edge in the compacted De Bruijn graph as being “strongly supported” (alternatively, “weakly supported”) if its average $k + 1$ -mer frequency is relatively high (alternatively, low). Examples of motifs are follows: (1) *Tips* are weakly supported dead ends in the graph created due to a base miscall occurring in a read at one of its end positions. Because of compaction such tips occur as single edges branching out of a strongly supported path, and can be easily removed; (2) *Bubbles* are detours that provide alternative paths between two terminal vertices. Due to compaction, these detours will also be single edges. There are two types of bubbles – weakly supported bubbles are manifestations of single base miscall occurring internal to a read and need to be removed; whereas, bubbles that originate from the same vertex and are supported roughly to the same degree could be the result of natural variations such as near identical paralogs (i.e., copies of the same gene occurring at different genomic loci). Such bubbles need to be retained. (3) *Spurious links* connect two otherwise disparate paths in the graph. Weakly supported links are manifestations of erroneous $k + 1$ -mers that happen to match the $k + 1$ -mer present at a valid genomic locus, and they can be severed by examining the supports of the other two paths.

The first pass of error correction on the compacted graph could reveal new instances of motifs that can be pruned through iterative passes subsequently until no new instances are observed.

Incorporation of Distance Constraints Using Pair-End Information

The goal of this step is to map the information provided by read pairs that are linked by the pair-end library onto the compacted and error corrected De Bruijn graph. Once mapped, this information can be used to guide the assembly tour of the graph consistent (to the extent possible) with the distance constraints imposed by the pair-end information. To appreciate the value added by this step to the assembly procedure, recall that the pair-end information consists of a list of read pairs of the form $\langle r_i, r_j \rangle$ that have originated from the same clonal insert during sequencing. In genomic distance parlance, this implies that the number of bases separating the two reads is bounded by a minimum and maximum. Also recall that an assembly from a De Bruijn graph corresponds to a tour of the graph (possibly multiple tours if there were gaps in the original sequencing). Now consider a scenario where a path in the De Bruijn graph branches into multiple separate subpaths. A correct assembly tour would have to decide which of those branches reflect the sequence of characters along the unknown genome. It should be easy to see how pair-end information can be used to resolve such situations.

To incorporate the information provided in the form of read pairs by such pair-end libraries, the YAGA algorithm uses a *cluster summarization* procedure, which can be outlined as follows: First, the list of read pairs provided as input by the pair-end information is delineated into a corresponding list of constituent $k + 1$ -mer pairs. Note that two $k + 1$ -mers from two reads of a pair can map to two different edges on the distributed graph. Furthermore, different positions along the same edge could be paired with positions emanating from different edges. To this effect, the algorithm attempts to compute a grouping of edge pairs based on their best alignment with the imposed distance constraints. To achieve this, an observed distance interval is computed between every edge pair on the graph linked by a read pair, and then overlapping intervals that capture roughly the same distances along the same orientation are incrementally clustered using a greedy heuristic. This last step is achieved using a two-phase clustering step that primarily relies on several rounds of *parallel sorting* tuples containing edge pair and interval distance information. The formal details pertaining to this algorithmic step has been omitted here for brevity.

Completing the Assembly Tour

An important offshoot of the above summarization procedure is that redundant distance information captured by edge pairs in the same cluster can be removed, thereby allowing significant compression in the level of information needed to store relative to the original graph. This compression, in most practical cases, would allow for the overall tour to be performed sequentially on a single node. The serial assembly touring procedure begins by using edges that have significantly longer edge labels than the original read length as “seeds,” and then by extending them in both directions as guided by the pair-end summarization traversal constraints.

The YAGA assembler has demonstrated scaling on 512 IBM BlueGene/L processors and performs assembly of over a billion synthetically generated reads in under 2 h. The run-time is dominated by the pair-end cluster summarization phase. As of this writing, performance-related information is not available for ABySS.

Future Trends

Genome sequencing and assembly is an actively pursued, constantly evolving branch of bioinformatics. Technological advancements in high-throughput sequencing have fueled algorithmic innovation and along with a compelling need to harness new computing paradigms. With the adoption of ever faster, cheaper, and massively parallel sequencing machines, this application domain is becoming increasingly data- and compute-intensive. Consequently, parallel processing is destined to play a critical role in genomic discovery.

Numerous large-scale sequencing projects including the 2001 assembly of the human genome to the most recent 2009 assembly of the maize genome have benefited from the use of parallel processing, although in different ad hoc ways. Heterogeneous clusters comprising of a mixture of a few high-end shared memory machines along with numerous compute nodes have been used to “farm” out tasks and accelerate the overlap computation phase in particular. This is justified because nearly all of these large-scale projects used the more traditional Sanger sequencing. A few special-purpose projects such as the 2005 maize gene-enriched sequencing used more strongly coupled parallel codes such as PaCE. However, with an aggressive adoption of next-generation sequencing for genome sequencing

and more complex genomes in the pipeline for sequencing (e.g., wheat, pine, metagenomic communities), this scenario is about to change, and more strongly coupled parallel codes are expected to become part of the mainstream computing in genome assembly.

In addition to *de novo* sequencing, next-generation sequencing is also increasingly being used in *genome resequencing* projects, where the goal is to assemble a genome of a particular variant strain (or subspecies) of an already sequenced genome. The type of computation that originates is significantly different from that of *de novo* assembly. For resequencing, the reads generated from a new strain are compared against a fully assembled sequence of a reference strain. This process, sometimes called *read mapping*, only requires comparison of reads against a much larger reference. Approaches that capitalize on advanced string data structures such as suffix trees are likely to play an active role in these algorithms. Also, as this branch of science becomes more data-intensive, reaching the petascale range, different parallel paradigms such as MapReduce need to be explored, in addition to distributed memory and shared memory models. The developments in genome sequencing can be carried over to other related applications that also involve large-scale sequence analysis, e.g., in transcriptomics, metagenomics, and proteomics.

Fine-grain parallelism in the area of string matching and sequence alignment has been an active pursued topic over the last decade and there are numerous hardware accelerators for performing sequence alignment on various platforms including General Purpose Graphical Processing Units, Cell Broadband Engine, Field-Programmable Gate Arrays, and Multicores. These advances are yet to take their place in mainstream sequencing projects.

Genome sequencing is at the cusp of revolutionary possibilities. With a rapidly advancing technology base, the possibility of realizing landmark goals such as personalized medicine and “\$1,000 genome” do not look distant or far-fetched. The well-advertised \$10 million Archon Genomics X PRIZE will be awarded to the first team that sequences 100 human genomes in 10 days, at a recurring cost of no more than \$10,000 per genome. In 2010, the cost for sequencing a human genome plummeted below \$10,000 using technologies from Illumina and SOLiD. Going past these next-generation technologies, however, companies such as

Pacific Biosciences are now releasing a third-generation (“gen-3”) sequencer that uses an impressive approach called single-molecule sequencing (or “SMS”) [9], and have proclaimed grand goals such as sequencing a human genome in 15 min for less than \$100 by 2014. These are exciting times for genomics, and the field is likely to continue serving as a rich reservoir for new problems that pose interesting compute- and data-intensive challenges which can be addressed only through a comprehensive embrace of parallel computing.

Related Entries

- ▶ [Homology to Sequence Alignment, From](#)
- ▶ [Suffix Trees](#)

Bibliographic Notes and Further Reading

Even though DNA sequencing technologies have been available since the late 1970s, it was not until the 1990s that they were applied at a genome scale. The first genome to be fully sequenced and assembled was the ~1.8 Mbp bacterial genome of *H. influenzae* in 1995 [8]. The sequencing of the more complex, ~3 billion bp long human genome followed [27]. Several other notable large-scale sequencing initiatives followed in the new millennium including that of chimpanzee, rice, and maize (to cite a few examples). All of these used either the WGS strategy or hierarchical strategy coupled with Sanger sequencing, and their assemblies were performed using programs that followed the overlap–layout–consensus model. The National Center for Biotechnology Information (NCBI) (<http://www.ncbi.nlm.nih.gov>) maintains a comprehensive database of all sequenced genomes. More than a dozen programs exist to perform genome assembly under the overlap–layout–consensus model. Notable examples include PCAP [12], Phrap (<http://www.phrap.org/>), Arachne [2], Celera [22], and TIGR assembler [26]. For a detailed review of fragment assembly algorithms, refer to [7, 24]. The parallel algorithm that uses the cluster-then-assemble approach along with the suffix tree data structure for assembly was implemented in a program called PaCE and was first described in [16] in the context of clustering Expressed Sequence Tag data and then later adapted for genome assembly [17]. Experiments

conducted as part of the maize genome sequencing consortium demonstrated scaling of this method to over a million reads generated from gene-enriched fractions of the maize genome on a 1,024 node BlueGene/L supercomputer [17]. The parallel suffix tree construction algorithm described in this entry was first described in [16] and a variant of this method was later presented in [11]. An optimal algorithm to detect maximal matching pairs of reads in parallel using the suffix tree data structure is presented in [16].

The NP-completeness of the Shortest Superstring Problem (SSP) was shown by Gallant et al. [10]. The De Bruijn graph formulation for genome assembly was first introduced by Idury and Waterman [13] in the context of a sequencing technique called sequencing-by-hybridization, and later extended to WGS based approaches in the EULER program by Pevzner et al. [23]. The string graph formulation was developed by Myers [21]. The proof of NP-Hardness for the overlap–layout–consensus is due to Kececioglu and Myers [18]. The proofs of NP-Hardness for the De Bruijn and string graphs models of genome assembly are due to Medvedev et al. [20].

Since the later part of 2000s, various next-generation sequencing technologies such as Roche 454 (<http://www.genome-sequencing.com/>), SOLiD (<http://www.appliedbiosystems.com/>), Illumina (<http://www.illumina.com/>), and HeliScope (<http://www.helicosbio.com/>) have emerged along side serial assemblers. A “third” generation of machines that promise a brand new way of sequencing (by single-molecule sequencing) are also on their way (e.g., Pacific Biosciences (<http://www.pacificbiosciences.com/>)). Consequently, the development of short read assemblers continue to be in hot pursuit. Edena, Newbler (<http://www.454.com>), PE-Assembler [1], QSRA [3], SHARCGS [6], SSAKE [28], and VCAKE [15] are all examples of programs that operate using the overlap graph model. EULER-SR [5], ALLPATHS [4], Velvet [29], SOAPdenovo [19], ABySS [25], and YAGA [14] are programs that use the De Bruijn graph formulation. Of these tools, PE-Assembler, ABySS, and YAGA are parallel implementations, although to varying degrees as described in the main text.

Acknowledgment

Study supported by NSF grant IIS-0916463.

Bibliography

1. Ariyaratne P, Sung W (2011) PE-assembler: de novo assembler using short paired-end reads. *Bioinformatics* 27(2):167–174
2. Batzoglou S, Jaffe DB, Stanley K, Butler J et al (2002) ARACHNE: a whole-genome shotgun assembler. *Genome Res* 12(1):177–189
3. Bryant D, Wong W, Mockler T (2009) QSRA – a quality-value guided de novo short read assembler. *BMC Bioinform* 10(1):69
4. Butler J, MacCallum L, Kleber M, Shlyakhter IA et al (2008) ALL-PATHS: de novo assembly of whole-genome shotgun microreads. *Genome Res* 18:810–820
5. Chaisson MJ, Pevzner PA (2008) Short read fragment assembly of bacterial genomes. *Genome Res* 18:324–330
6. Dohm J, Lottaz C, Borodina T, Himmelbauer H (2007) SHARCGS, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing. *Genome Res* 17(11):1679–1706
7. Emrich S, Kalyanaraman A, Aluru S (2005) Chapter 13: algorithms for large-scale clustering and assembly of biological sequence data. In: *Handbook of computational molecular biology*. CRC Press, Boca Raton
8. Fleischmann R, Adams M, White O, Clayton R et al (1995) Whole-genome random sequencing and assembly of *Haemophilus influenzae* rd. *Science* 269(5223):496–512
9. Flusberg BA, Webster DR, Lee JH, Travers KJ et al (2010) Direct detection of DNA methylation during single-molecule, real-time sequencing. *Nat Methods* 7:461–465
10. Gallant J, Maier D, Storer J (1980) On finding minimal length superstrings. *J Comput Syst Sci* 20:50–58
11. Ghosh A, Makarychev K (2009) Indexing genomic sequences on the IBM blue gene. In: *Proceedings ACM/IEEE conference on supercomputing*. Portland
12. Huang X, Wang J, Aluru S, Yang S, Hiller L (2003) PCAP: a whole-genome assembly program. *Genome Res* 13:2164–2170
13. Idury RM, Waterman MS (1995) A new algorithm for DNA sequence assembly. *J Comput Biol* 2(2):291–306
14. Jackson BG, Regenitter M, Yang X, Schnable PS, Aluru S (2010) Parallel de novo assembly of large genomes from high-throughput short reads. In: *IEEE international symposium on parallel distributed processing*, pp 1–10
15. Jeck W, Reinhardt J, Baltrus D, Hickenbotham M et al (2007) Extending assembly of short DNA sequences to handle error. *Bioinformatics* 23:2942–2944
16. Kalyanaraman A, Aluru S, Brendel V, Kothari S (2003) Space and time efficient parallel algorithms and software for EST clustering. *IEEE Trans Parallel Distrib Syst* 14(12):1209–1221
17. Kalyanaraman A, Emrich SJ, Schnable PS, Aluru S (2007) Assembling genomes on large-scale parallel computers. *J Parallel Distrib Comput* 67(12):1240–1255
18. Kececioglu J, Myers E (1995) Combinatorial algorithms for DNA sequence assembly. *Algorithmica* 13(1–2):7–51
19. Li R, Zhu H, Ruan J, Qian W et al (2010) De novo assembly of human genomes with massively parallel short read sequencing. *Genome Res* 20(2):265–272
20. Medvedev P, Georgiou K, Myers G, Brudno M (2007) Computability of models for sequence assembly. Lecture notes in computer science, vol 4645. Springer, Heidelberg, pp 289–301
21. Myers EW (2005) The fragment assembly string graph. *Bioinformatics*, 21(Suppl 2):ii79–ii85
22. Myers EW, Sutton GG, Delcher AL, Dew IM et al (2000) A Whole-Genome assembly of *drosophila*. *Science* 287(5461):2196–2204
23. Pevzner PA, Tang H, Waterman M (2001) An eulerian path approach to DNA fragment assembly. In: *Proceedings of the national academy of sciences of the United States of America*, vol 98, pp 9748–9753
24. Pop M (2009) Genome assembly reborn: recent computational challenges. *Briefings in Bioinformatics* 10(4):354–366
25. Simpson J, Wong K, Jackman S, Schein J et al (2009) ABySS: a parallel assembler for short read sequence data. *Genome Res* 19:1117–1123
26. Sutton GG, White O, Adams MD, Kerlavage AR (2011) TIGR assembler: a new tool for assembling large shotgun sequencing projects. *Genome Sci Technol* 1(1):9–19
27. Venter C, Adams MD, Myers EW, Li P et al (2001) The sequence of the human genome. *Science* 291(5507):1304–1351
28. Warren P, Sutton G, Holt R (2006) Assembling millions of short DNA sequences using SSAKE. *Bioinformatics* 23:500–501
29. Zerbino DR, Velvet BE (2008) Algorithms for de novo short read assembly using de bruijn graphs. *Genome Res* 18:821–829

Genome Sequencing

► Genome Assembly

3GIO

► PCI Express

Glasgow Parallel Haskell (GpH)

KEVIN HAMMOND

University of St. Andrews, St. Andrews, UK

Synonyms

GpH (Glasgow Parallel Haskell)

Definition

Glasgow Parallel Haskell (GpH) is a simple parallel dialect of the purely functional programming language, Haskell. It uses a *semi-explicit* model of parallelism, where possible parallel threads are marked by the programmer, and a sophisticated runtime system then

decides on the timing of thread creation, allocation to processing elements, migration. There have been several implementations of GpH, covering platforms ranging from single multicore machines through to computational grids or clouds. The best known of these is the GUM implementation that targets both shared-memory and distributed-memory systems using a sophisticated virtual shared-memory abstraction built over a common message-passing layer, but there is a new implementation, GHC-SMP, which directly exploits shared-memory systems, and which targets multicore architectures.

Discussion

Purely Functional Languages and Parallelism

Because of the absence of side effects in purely functional languages, it is relatively straightforward to identify computations that can be run in parallel: any sub-expression can be evaluated by a dedicated parallel task. For example in the following very simple function definition, each of the two arguments to the addition operation can be evaluated in parallel.

```
f x = fibonacci x + factorial x
```

This property was already realized by the mid-1980s, when there was a surge of interest both in parallel evaluation in general, and in the novel architectural designs that it was believed could overcome the sequential “von-Neumann bottleneck.” In fact, the main issue in a purely functional language is not extracting enough parallelism – it is not unusual for even a short-running program to produce many tens of thousands of parallel threads – but is rather one of identifying sufficiently large-grained parallel tasks. If this is not done, then thread creation and communication overheads quickly eliminate any benefit that can be obtained from parallel execution. This is especially important on conventional processor architectures, which historically provided little, if anything, in the way of hardware support for parallel execution. The response of some parallel functional language designers has therefore been to provide very explicit parallelism mechanisms. While this usually avoids the problem of excessive parallelism, it places a significant burden on the programmer, who must understand the details of parallel execution as well as

the application domain, and it can lead to code that is specialized to a specific parallel architecture, or class of architectures. It also violates a key design principle for most functional languages, which is to provide as much isolation as possible from the underlying implementation. GpH is therefore designed to allow programmers to provide information about parallel execution while delegating issues of placement, communication, etc. to the runtime system.

History and Development of GpH

Glasgow Parallel Haskell (GpH) was first defined in 1990. GpH was designed to be a simple parallel extension to the then-new nonstrict, purely functional language Haskell [10], adding only two constructs to sequential Haskell: *par* and *seq*. Unlike most earlier *lazy* functional languages, Haskell was always intended to be parallelizable. The use of the term “non-strict” rather than *lazy* reflects this: while *lazy* evaluation is inherently sequential, since it fixes a specific evaluation order for sub-expressions, under Haskell’s non-strict evaluation model, any number of sub-expressions can be evaluated in parallel provided that they are needed by the result of the program.

The original GpH implementation targeted the GRIP novel parallel architecture, using direct calls to low-level GRIP communications primitives. GRIP was a shared-memory machine, using custom microcoded “intelligent memory units” to share global program data between off-the-shelf processing elements (Motorola 68020 processors, each with a private 4 MB memory), developed in an Alvey research project which ran from 1985 to 1988. Initially, GpH built on the prototype **Haskell** compiler developed by Hammond and Peyton Jones in 1989. It was subsequently ported to the Glasgow Haskell Compiler, GHC [9], that was developed at Glasgow University from 1991 onward, and which is now the de-facto standard compiler for Haskell (since the focus of the maintenance effort was moved to Microsoft Research in 1998, GHC has also become known as the Glorious Haskell Compiler). In 1994, the communications library was redesigned to give what became the highly portable GUM implementation [15]. This allowed a single parallel implementation to target both commercial shared-memory systems and the then-emerging class of cost-effective loosely coupled networks of workstations. Initially, GUM targeted

system-specific communication libraries, but it was subsequently ported to PVM. There are now UDP, PVM, MPI, and MPICH-G2 instances of GUM, as well as system-specific implementations, and the same GUM implementation runs on multicore machines, shared-memory systems, workstation clusters, computational grids, and is being ported to large-scale high-performance systems, such as the 22,656-core HECToR system at the Edinburgh Parallel Computer Centre. Key parts of the GUM implementation have also been used to implement the *Eden* [5] parallel dialect of Haskell, and GpH is being incorporated into the latest mainstream version of GHC.

Although the two parallelism primitives that GpH uses are very simple, it became clear that they could be packaged using higher-order functions to give much higher-level parallel abstractions, such as parallel pipelines, data-parallelism, etc. This led ultimately to the development of *evaluation strategies* [14]: high-level parallel structures that are built from the basic *par* and *seq* primitives using standard higher-order functions. Because they are built from simple components and standard language technology, evaluation strategies are highly flexible: they can be easily composed or nested; the parallelism structure can change dynamically; and the applications programmer can define new strategies on an as-needed basis, while still using standard strategies.

The GpH Model of Parallelism

GpH is unusual in using only two parallelism primitives: *par* and *seq*. The design of the *par* primitive dates back to the late 1980s [8], where it was used in a parallel implementation of the Lazy ML (LML) compiler. The primitive, higher-order, function, *par* is used by the programmer to mark a sub-expression as being suitable for parallel evaluation. For example, a variant of the function *f* above can be parallelized using two instances of *par*.

```
f x =
  let r1 = fibonacci x;
  r2 = factorial x in
  let result = (r1, r2) in
    r1 `par` (r2 `par` result)
```

r1 and *r2* can now both be evaluated in parallel with the construction of the result pair (*r1*, *r2*). There is no need to specify any explicit communication, since

the results of each computation are shared through the variables *r1* and *r2*. The runtime system also decides on issues such as when a thread is created, where it is placed, how much data is communicated, etc. Very importantly, it can also decide *whether* a thread is created. The parallelism model is thus *semi-explicit*: the programmer marks possible sites of parallelism, but the runtime system takes responsibility for the underlying parallel control based on information about system load, etc. This approach therefore eliminates significant difficulties that are commonly experienced with more explicit parallel approaches, such as raw MPI. The programmer needs to make sure there is enough scope for parallelism, but does not need to worry about issues of deadlock, communication, throttling, load-balancing, etc. In the example above, it is likely that only one of *r1* or *r2* (or neither) will actually be evaluated in parallel, since the current thread will probably need both their values. Which of *r1* or *r2* is evaluated first by the original thread will, however, depend on the context in which it is called. It is therefore left unspecified to avoid unnecessary sequentialization.

Lazy Thread Creation

Several different versions of the *par* function have been described in the literature. The version used in GpH is asymmetric in that it marks its first argument as being suitable for possible execution (the expression is *sparked*), while continuing sequential execution of its second argument, which forms the result of the expression. For example, in *par s e*, the expression *s* will be sparked for possible parallel evaluation, and the value of *e* will be returned as the result of the current thread. Since the result expression is always evaluated by the spawning thread, and since there can be no side effects, it follows that it is completely safe to ignore any spark. That is, unlike many parallel systems, the creation of threads from sparks is entirely optional. This fact can be used to throttle the creation of threads from sparks in order to avoid swamping the parallel machine. This is a *lazy thread creation* approach (the term “*lazy task creation*” was coined later by Mohr et al. [7] to describe a similar mechanism in MultiLisp). A corollary is that, since sparks do not carry any execution state, they can be very lightweight. Generally, a single pointer is adequate to record a sparked expression in most implementations of GpH.

Sparks may be chosen for conversion to threads using a number of different strategies. One common approach is to use the oldest spark first. If the application is a divide-and-conquer program, this means that the spark representing the largest amount of work will be chosen. Alternatively, an approach may be used where the youngest (smallest) spark is executed locally and the oldest (largest) spark is offloaded for remote execution. This will improve locality, but may increase thread creation costs, since many of the locally created threads might otherwise be subsumed into their parent thread.

Parallel Graph Reduction

A second key issue that must be dealt with is that of thread synchronization. Efficient sequential non-strict functional language implementations generally use an evaluation technique called *graph reduction*, where a program builds a graph data structure representing the work that is needed to give the result of a program and gradually rewrites this using the functional rules defined in the program until the graph is sufficiently complete to yield the required result. This rewriting process is known as *reducing the graph to some normal form* (in fact *weak head normal form*). Each node in the graph represents an expression in the original program. Initially, this will be a single unevaluated expression corresponding to the result of the program (a “thunk”). As execution proceeds, thunks are evaluated, and each graph node is overwritten with its result. In this way, results are automatically shared between several consumers. Only graph nodes that actually contribute to the final result need to be rewritten. This means that unnecessary work can be avoided, a process that, in the sequential world, allows *lazy evaluation*.

The same mechanism naturally lends itself to parallel evaluation. Each shared node in the graph represents a possible synchronization point between two parallel threads. The first thread to evaluate a graph node will lock it. Any thread that evaluates the node in parallel with the thread that is evaluating it will then block when it attempts to read the value of the node. When the evaluating thread produces the result, the graph node will be updated and any blocked threads will be awoken. In this way, threads will automatically synchronize through the graph representing the computation, not

just at the root of each thread, but whenever they share any sub-expressions.

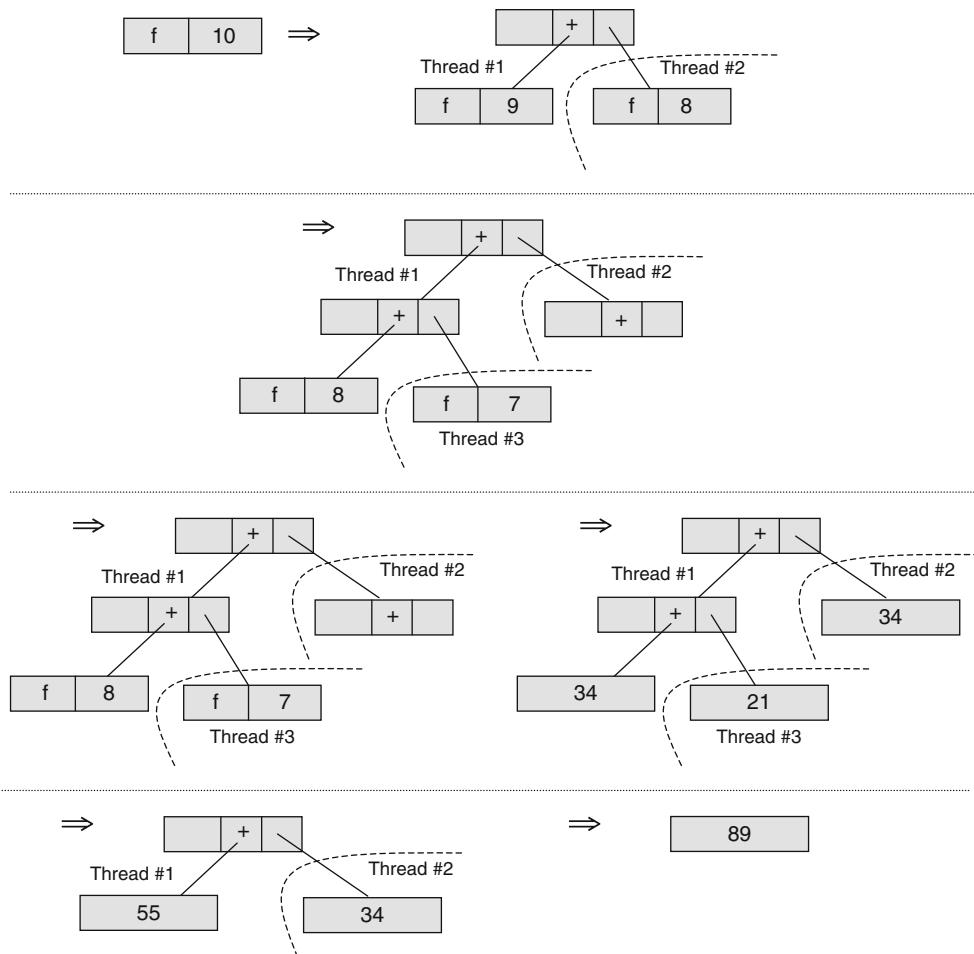
[Figure 1](#) shows a simple example of a divide-and-conquer parallel program, where the root of the computation, $f\ 9$, is rewritten using three threads: one for the main computation and two sub-threads, one to evaluate $f\ 8$ and one to evaluate $f\ 7$. These thunks are linked into the addition nodes in the main computation. Having evaluated the sparked thunks, the second and third threads update their root nodes with the corresponding result. Once the main thread has evaluated the remaining thunk (another call to $f\ 8$, which we have assumed is not shared with Thread #2), it will incorporate these results into its own result. The final stage of the computation is to rewrite the root of the graph (the result of the program) with the value 89.

Evaluate-and-Die

As a thread evaluates its program graph, it may encounter a node that has been sparked. There are three possible cases, depending on the evaluation status of the sparked node. If the thunk has already been evaluated, then its value can be used as normal. If the thunk has not yet been evaluated, then the thread will evaluate it as normal, first annotating it to indicate that it is under evaluation. Once a result is produced, the node will be overwritten with the result value, and any blocked threads will be reawakened. Finally, if the node is actually under evaluation, then the thread must be blocked until the result is produced and the node updated with this value. This is the *evaluate-and-die* execution model [8]. The advantage of this approach is that it automatically absorbs sparks into already executing threads, so increasing their granularity and avoiding thread creation overheads. If a spark refers to a thunk that has already been evaluated then it may be discarded without a thread ever being created.

The *seq* Primitive

While *par* is entirely adequate for identifying many forms of parallelism, Roe and Peyton Jones discovered [13] that by adding a sequential combining function, called *seq*, much tighter control could be obtained over parallel execution. This could be used both to reduce the need for detailed knowledge of the underlying parallel implementation and to encode more sophisticated patterns of parallel execution. For example, one



Glasgow Parallel Haskell (GpH). Fig. 1 Parallel graph reduction

of the *par* constructs in the example above can be replaced by a *seq* construct, as shown below.

```
f x =
  let r1 = fibonacci x;
  r2 = factorial x in
  let result = (r1, r2) in
  r1 `par` (r2 `seq` result)
  -- was r1 `par` r2 `par` result
```

The *seq* construct acts like ; in a conventional language such as C: it first evaluates its first argument (here *r2*), and then returns its second argument (here *result*). So in the example above, rather than creating two sparks as before, now only one is created. Previously, depending on the order in which threads were scheduled, either the parent thread would have blocked when it evaluated *r1* or *r2*, because these

were already under evaluation, or one or both of the sparked threads would have blocked, because they were being evaluated (or had been evaluated) by the parent thread. Now, however, the only possible synchronization is between the thread evaluating *r1* and the parent thread. Note that it is not possible to use either of the simpler forms of *par r1 (r1, r2)* or *par r2 (r1, r2)* to achieve the same effect, as might be expected. Because the implementation is free to evaluate the pair *(r1, r2)* in whichever order it prefers, there is a 50% probability that the spark will block on the parent thread.

This is not the only use of *seq*. For example, evaluation strategies (discussed below) also make heavy use of this primitive to give precise control over evaluation order. While not absolutely essential for parallel evaluation, it is thus very useful to provide a finer

degree of control than can be achieved using *seq* alone. However, there are two major costs to the use of *seq*. The first is that the strictness properties of the program may be changed – this means that some thunks may be evaluated that were previously not needed, and that the termination properties of the program may therefore be changed. The second is that parallel programs may become *speculative*.

Speculative Evaluation

As described above, in GpH it is safe to *spark* any sub-expression that is needed by the result of the parallel computation. However, nothing prevents the programmer from sparking a sub-expression that is not known to be needed. This therefore allows speculative evaluation. Although there was some early experimentation with mechanisms to control speculation, these were found to be very difficult to implement, and current implementations of GpH do not provide facilities to kill created threads, change priorities dynamically, etc., as is necessary to support safe speculation. For example,

```
spec x y = x `par` y
```

defines a new function *spec* that sparks *x* and continues execution of *y*. If *x* is not definitely needed by *y*, or is a completely independent expression, this will spark *x* speculatively. If a thread is created to evaluate *x*, it will terminate either when it has completed evaluation of *x*, or when the main program terminates, having produced its own result, or if it evaluates an undefined value. In the latter case, it may cause the entire program to fail. Moreover, it is theoretically possible to prevent any progress on the main computation by flooding the runtime system with useless speculative threads. For these reasons, any speculative evaluation has to be treated carefully: if the program is to terminate, speculative sub-expressions should terminate in finite time without an error, and there should not be too many speculative sparks.

Evaluation Strategies

As discussed above, it is possible to construct higher-level parallel abstractions from basic *seq/par* constructs using standard Haskell programming constructs. These parallel abstractions can be associated with computation functions using higher-order definitions. This is the *evaluation strategy* approach. The key idea of evaluation

strategies to separate what is to be evaluated from how it could be evaluated in parallel [14]. For example, a simple sequential ray tracing function could be defined as follows:

```
raytracer :: Int -> Int -> Scene ->
[Light] -> [[Vector]]
raytracer xlim ylim scene lights =
map traceline [0..ylim-1]
  where traceline y = [tracepixel scene
    lights x y | x <- [0..xlim-1]]
```

Given a visible image with maximum *x* and *y* dimensions of *xlim* and *ylim*, the *raytracer* function applies the *traceline* function to each line in the image (by mapping it across each value of *y* in the range *0..ylim-1*), and hence applies the *tracepixel* function to each pixel using the specified *scene* and lighting model. The *raytracer* function may be parallelized, for example, by adding the *parMap* strategy to parallelize each line as follows:

```
raytracer :: Int -> Int -> Scene ->
[Light] -> [[Vector]]
raytracer xlim ylim scene lights = map
  traceline [0..ylim-1] `using` parMap rnf
```

Here, *parMap* is a strategy that specifies that each element of its value argument should be evaluated in parallel. It is parameterized on another strategy that specifies what to do with each element. Here, *rnf* indicates that each element should be evaluated as far as possible (*rnf* stands for “reduce to normal form”). The *using* function simply applies its second argument (an evaluation strategy) to its first argument (a functional value), and returns the value. It can be easily defined using higher-order functions and the *seq* primitive as follows:

```
using :: a -> Strategy a -> a
x `using` s = s x `seq` x
```

So, when a strategy is applied to an expression by the *using* operation, the strategy is first applied to the value of the expression, and once this has completed, the value is returned. This allows the use of both parallel and sequential strategies.

In fact, any list strategy may be used to parallelize the *raytracer* function instead of *parMap*. For example, a task farm could be used, or the list could be divided into equally sized chunks as shown below.

```

raytracer :: Int -> Int -> Scene ->
[Light] -> [[Vector]]
raytracer xlim ylim scene lights = ...
`using' parListChunk chunkSize rnf

```

Evaluation strategies have proved to be very powerful and flexible, having been successfully applied to several large programs, including symbolic programs with irregular patterns of parallelism [4]. Typically, only a few lines need to be changed at key points by adding appropriate `using` clauses. In abstracting parallel patterns from sequential computations, evaluation strategies have some similarities with *algorithmic skeletons*. The key differences between evaluation strategies and typical skeleton approaches is that evaluation strategies do not mandate a specific parallel implementation (since they rely on semi-explicit parallelism, they provide hints to the runtime system rather than directives); and that they are completely user-programmable – everything above the `par` and `seq` primitives is programmed using standard Haskell code. Unlike most skeleton approaches, they may also be easily composed to give irregular nested parallel structures or phased parallel computations, for example.

The GUM Implementation of GpH

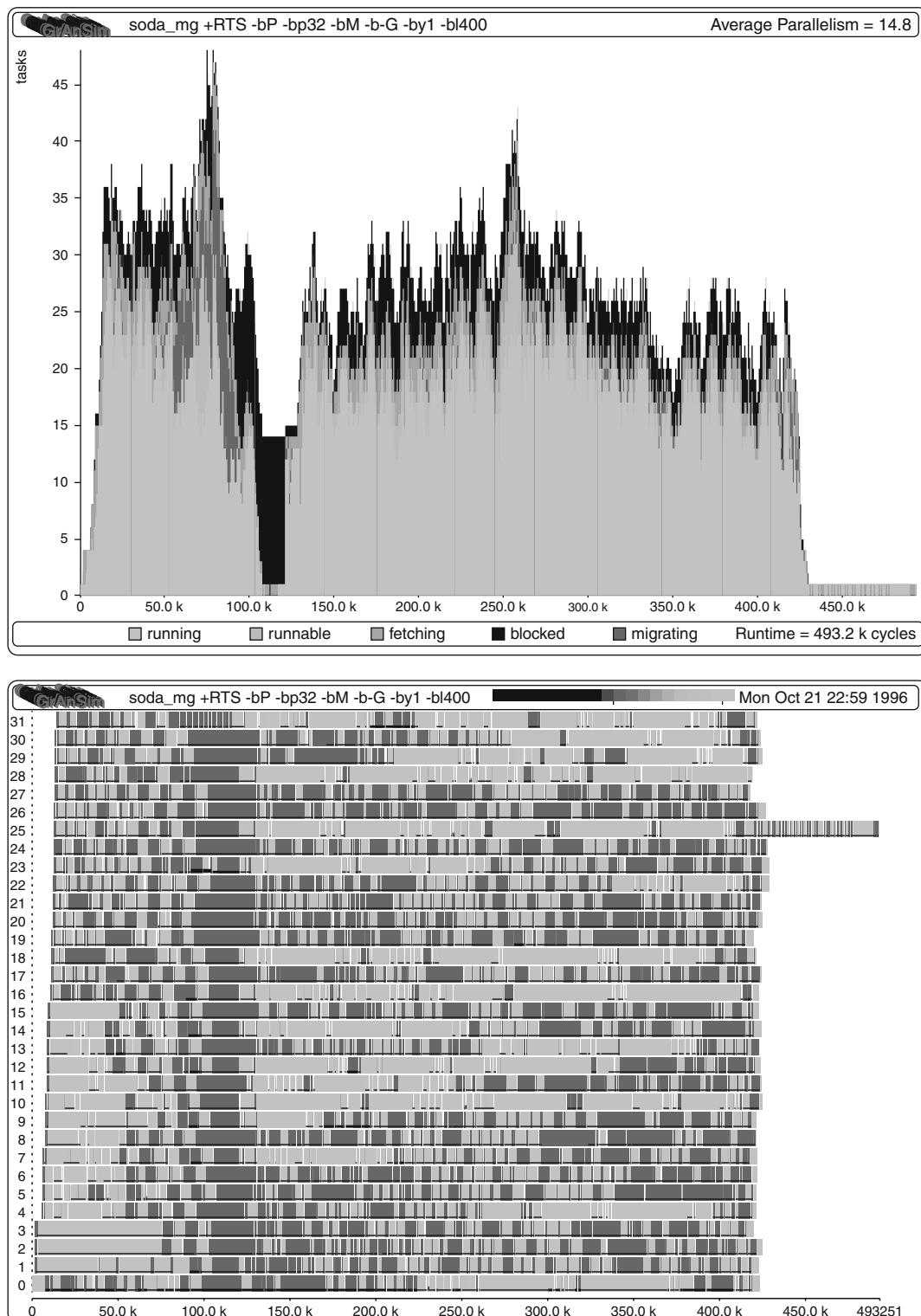
GUM is the original and most general implementation of GpH. It provides a virtual shared-memory abstraction for parallel graph reduction, using a message-passing implementation to target both physically shared-memory and distributed-memory systems. In the GUM model, the graph representing the program that is being evaluated is distributed among the set of processor elements (PEs) that are available to execute the program. These PEs usually correspond to the cores or processors that are available to execute the parallel program, but it is possible to map PEs onto multiple processes on the same core/processor, if required. Each PE manages its own execution environment, with its own local pools of sparks and threads, which it schedules as required. Sparks and threads are offloaded on demand to maintain good work balance.

A key feature of GUM is that it uses a two-level memory model: each PE has its own private heap that contains unshared program graph created by local threads. Within this heap, some graph nodes may be

shared with other PEs. These nodes are given *global addresses*, which identify the owner of the graph node. The advantage of this model is that it allows completely independent memory management: by keeping tables of global in-pointers, which are used as garbage collection roots into the local heap, each PE can garbage-collect its own local heap independently of all other PEs. This local garbage collection is conservative, as required, since even if a global in-pointer is no longer referenced by any other PE, it will still be treated as a garbage collection root. In order to collect global garbage, a separate scheme is used, based on distributed reference counting. Since the majority of the program graph never needs to be shared, this approach brings major efficiency gains over typical single-level memory management. In addition to the reduced need for synchronization during garbage collection, there is also no need to maintain global locks across purely local graph. Within each PE, GUM uses the same efficient (and sequential) garbage collector as the standard GHC implementation. This is currently a stop-and-copy generational collector based on Appel's collector.

Visualizing the Behavior of GpH Programs

Good visualization is an essential part of understanding parallel behavior. Runtime information can be visualized at a variety of levels to give progressively more detailed information. For example, Fig. 2 visualizes the overall and per-PE parallel activity for a 32-PE system running the soda application (a simple crossword-puzzle solver). Apart from the clear phase-transition about 25% into the execution, very few threads are blocked; and there are very few runnable, but not running threads. Those that are runnable are migrated to balance overall system load. The profile also reveals that relatively little time is spent fetching nonlocal data. The per-PE profile gives a more detailed view. It is obvious that the workload is well balanced and that the work-stealing mechanism is effective in distributing work. The example also shows where PEs are idle for repeated periods, and that only the main PE is active at the end of the computation (collating results to be written as the output of the program). It thus clearly identifies places where improvements could be made to the parallel program.



Glasgow Parallel Haskell (GpH). Fig. 2 Sample overall and per-PE Activity Profiles for a 32-PE machine

The GHC-SMP Implementation of GpH

A recent development is the GHC-SMP [6] implementation of GpH from Microsoft Research Labs, Cambridge, UK, which is integrated into the standard GHC distribution. This provides an implementation of **GpH** that specifically targets shared-memory and multicore systems. It uses a similar model of PEs and threads to the **GUM** implementation, including spark pools and runnable thread pools, and implements a similar spark-stealing model. The key difference from GUM is that GHC-SMP uses a physically shared heap, rather than a virtual shared heap with an underlying message-passing implementation. This heap is garbage-collected using a global stop-and-copy collector that requires the synchronization of all PEs, but which may itself be executed in parallel using the available processor cores. Also, unlike GUM, GHC-SMP exports threads to remote PEs based on the load of the local core, rather than on demand.

GRID-GUM and the SymGrid System

By replacing the low-level communications library with MPICH-G2, it is possible to execute GUM (or Eden – see below) not only on standard clusters of workstations, but also on wide-area computational grids, coordinated by the standard Globus grid middleware. The GRID-GUM and grid-enabled Eden implementations form the basis for the SymGrid-Par middleware [3], which aims to provide high-level support for computational grids for a variety of symbolic computing systems. The middleware coordinates symbolic computing engines into a coherent parallel system, providing high-level skeletons. The system uses a high-level data-exchange protocol (SCSCP) based on the standard OpenMath XML format for mathematical data. Results have been very promising to date, with superlinear performance being achievable for some mathematical applications, without changing any of the sequential symbolic computing engines.

The GranSim Simulator

The GranSim parallel simulator was developed in 1995 [1] as an efficient and accurate simulator for GpH running on a sequential machine, specifically to expose granularity issues. It is unusual in modifying the sequential GHC runtime system so that evaluating graph nodes also triggers the addition of sparks,

and simulates inter-PE communication. It thus follows the actual GUM implementation very precisely. It allows a range of communication costs to be simulated for a specific parallel application, ranging from zero (ideal communication) to costs that are similar to those Program execution times are also simulated, using a cost model that takes architectural characteristics into account. One of the key uses of GranSim is as the core of a parallel program development methodology, where a GpH program is first simulated under ideal parallel conditions, and then under communication cost settings for specific parallel machines: shared-memory, distributed-memory, etc. This allows the program to be gradually tuned for a specific parallel setting without needing access to the actual parallel machine, and in a way that is repeatable, can be easily debugged, and which provides detailed metrics.

GdH and Mobile Haskell

GdH [11] is a distributed Haskell dialect that builds on GpH. It adds explicit constructs for task creation on specific processors, with inter-processor communication through explicit mutable shared variables, that can be written on one processor and read on another. Internally, each processor runs a number of threads. The **GdH** implementation extends the GUM implementation with a number of explicit constructs. The main construct is `revalIO`, which constructs a new task on a specific processor. In conjunction with mutable variables, this can be used to construct higher-level constructs. For example, the `showSystem` definition below prints the names of all available PEs. The `getAllHostNames` function maps the `hostName` function over the list of all processor identifiers, returning an IO action that obtains the environment variable `HOST` on the specified PE. This is used in the `showSystem` IO operation which first obtains a list of all host names, then outputs them as a sorted list, with duplicates eliminated using the standard `nub` function.

```
getAllHostNames :: IO [String]
getAllHostNames = mapM hostName allPEId

hostName :: PEId -> IO String
hostName pe = evalIO (getEnv "HOST") pe

showSystem = do { hostnames <=
```

```
getAllHostNames; showHostNames hostnames}
  where
    showHostNames names = putStrLn
      (show (sort (nub names)))
```

At the language level, GdH is similar to Concurrent Haskell, which is designed to execute explicit concurrent threads on a single PE. The key language difference is the inclusion of an explicit PEID to allocate tasks to PEs, supported by a distributed runtime environment. GdH is also broadly similar to the industrial Erlang language, targeting a similar area of distributed programming, but is non-strict rather than strict and, since it is a research language, does not have the range of telecommunications-specific libraries that Erlang supports in the form of the Erlang/OTP development platform. Mobile Haskell [12] similarly extends Concurrent Haskell, adding higher-order communication channels (known as Mobile Channels), to support mobile computations across dynamically evolving distributed systems. The system uses a bytecode implementation to ensure portability, and serializes arbitrary Haskell values including (higher-order) functions, IO actions, and even mobile channels.

Eden

Eden [5] (described in a separate encyclopedia entry) is closely related to GpH. Like GpH, it uses an implicit communication mechanism, sharing values through program graph nodes. However, it uses an explicit process construct to create new processes and explicit process application to pass streams of values to each process. Unlike GpH, all data that is passed to an Eden process must be evaluated before it is communicated. There is also no automatic work-stealing mechanism, task migration, granularity agglomeration, or virtual shared graph mechanism. Eden, does, however, provide additional threading support: each output from the process is evaluated using its own parallel thread. It also provides mechanisms to allow explicit communication channels to be passed as first-class objects between processes.

Eden has been used to implement both algorithmic skeletons and evaluation strategies, where it has the advantage of providing more controllable parallelism but the disadvantage of losing adaptivity. One

particularly useful technique is to use Eden to implement a master-worker evaluation strategy, where a set of dynamically generated worker functions is allocated to a fixed set of worker processors using an explicitly programmed scheduler. This allows Eden to deal with varying thread granularities, without using a lazy thread creation mechanism, as in GpH.

While it does not need the advanced adaptivity mechanisms that GUM provides, the Eden implementation shares several basic components, in particular, the communication library and scheduler have very similar implementations.

G

Current Status

GpH has been adopted as a significant part of the Haskell community effort on parallelism. The evaluation strategies library has just been released as part of the mainstream GHC compiler release, building on the GHC-SMP implementation, and there has also been significant recent effort on visualization with the release of the EdenTV and ThreadScope visualizers for Eden and GHC-SMP, respectively. Work is currently underway to integrate GUM and GHC-SMP to give a wide-spectrum implementation for GpH, and Eden will also form part of this effort. The SymGrid-Par system is being developed as part of a major UK project to provide support for high-performance computing on massively parallel machines, such as HECToR.

Future Directions

The advent of multicore computers and the promise of manycore computers have changed the nature of parallel computing. The GpH model of lightweight multithreaded parallelism is well suited to this new order, offering the ability to easily generate large amounts of parallelism. The adaptivity mechanisms built into the GUM implementation mean that the parallel program can dynamically, and automatically, change its behavior to increase parallelism, improve locality, or throttle back parallelism as required.

Future parallel systems are likely to be built more hierarchically than present ones, with processors built from heterogeneous combinations of general-purpose cores, graphics processing units, and other specialist units, using several communication networks and a complex memory hierarchy. Proponents of manycore architectures anticipate that a single processor

will involve hundreds or even thousands of such units. Because of the cost of maintaining a uniform memory model across large numbers of systems, when deployed on a large scale these processors are likely to be combined hierarchically into multiple levels of clusters. These systems may then be used to form high performance “clouds.” Future parallel languages and implementations must therefore be highly flexible and adaptable, capable of dealing with multiple levels of communication latency and internal parallel structure, and perhaps fault tolerance. The GpH model with evaluation strategies, supported by adaptive implementations such as GUM forms a good basis for this, but it will be necessary to extend the existing models and implementations to cover more heterogeneous processor types, to deal with multiple levels of parallelism and additional program structure, and to focus more directly on locality issues.

Related Entries

- [Eden](#)
- [Fortress \(Sun HPCS Language\)](#)
- [Functional Languages](#)
- [Futures](#)
- [MPI \(Message Passing Interface\)](#)
- [MultiLisp](#)
- [NESL](#)
- [Parallel Skeletons](#)
- [Processes, Tasks, and Threads](#)
- [Profiling](#)
- [PVM \(Parallel Virtual Machine\)](#)
- [Shared-Memory Multiprocessors](#)
- [Sisal](#)
- [Speculation, Thread-Level](#)

Bibliographic Notes and Further Reading

The main venues for publication on GpH and other parallel functional languages are the International Conference on Functional Programming (ICFP) and its satellite events including the Haskell Symposium; the International Symposium on Implementation and Application of Functional Languages (IFL); the International Conference on Programming Language Design and Implementation (PLDI); and the Symposium on Trends in Functional Programming

(TFP). Papers also frequently appear in the Journal of Functional Programming (JFP). A survey of Haskell-based parallel languages and implementations, as of 2002, can be found in [16], and a general introduction to research in parallel functional programming, as of 1998, can be found in [2]. Some examples of the use of GpH in larger applications can be found in [4]. Much of this entry is based on private notes, e-mails, and final reports on the various research projects that have used GpH. The main paper on evaluation strategies is [14]. The main paper on the GUM implementation is [15], and the main paper on the GranSim simulator is [1]. Many subsequent papers have used these systems and ideas. For example, one recent paper describes the SymGrid-Par system [3]. Further material may be found on the GpH web page at <http://www.macs.hw.ac.uk/~dsg/gph/>, on the GdH web page at <http://www.macs.hw.ac.uk/~dsg/gdh/>, on the GHC web page at <http://www.haskell.org/ghc>, on the SCIEnce project web page at <http://www.symbolic-computation.org>, on the Eden web page at <http://www.mathematik.uni-marburg.de/~eden>, and on the contributor’s web page at <http://www-fp.cs.st-andrews.ac.uk/~kh>.

Bibliography

1. Hammond K, Loidl H-W, Partridge A (1995) Visualising granularity in parallel programs: a graphical winnowing system for Haskell. Proceedings of the HPFC’95 – Conference on High Performance Functional Computing, Denver, pp 208–221, 10–12 April 1995
2. Hammond K, Michaelson G (eds) (1999) Research directions in parallel functional programming. Springer, Heidelberg
3. Hammond K, Zain AA, Cooperman G, Petcu D, Trinder PW (2007) SymGrid: a framework for symbolic computation on the grid. Proceedings of the EuroPar ’07: 13th International EuroPar Conference, Springer LNCS 4641, Rennes, France, pp 457–466
4. Loidl H-W, Trinder P, Hammond K, Junaidu SB, Morgan RG, Peyton Jones SL (1999) Engineering parallel symbolic programs in GpH. Concurrency Pract Exp 11(12):701–752
5. Loogen R, Ortega-Mallén Y, Peña Mar R (2005) Parallel functional programming in Eden. J Functional Prog 15(3):431–475
6. Marlow S, Jones SP, Singh S (2009) Runtime support for multicore Haskell. Proceedings of the ICFP ’09: 14th ACM SIGPLAN International Conference on Functional Programming, ACM Press, New York
7. Mohr E, Kranz DA, Halstead RH Jr (1991) Lazy task creation: a technique for increasing the granularity of parallel programs. IEEE Trans Parallel Distrib Syst 2(3):264–280
8. Peyton Jones S, Clack C, Salkild J (1989) High-performance parallel graph reduction. Proceedings of the PARLE’89 – Conference

- on Parallel Architectures and Languages Europe, Springer LNCS 365, pp 193–206
9. Peyton Jones S, Hall C, Hammond K, Partain W, Wadler P (1993) The Glasgow Haskell compiler: a technical overview. Proceedings of the JFIT (Joint Framework for Information Technology) Technical Conference, Keele, UK, pp 249–257
 10. Peyton Jones S (ed), Hughes J, Augustsson L, Barton D, Boutel B, Burton W, Fasel J, Hammond K, Hinze R, Hudak P, Johnsson T, Jones M, Launchbury J, Meijer E, Peterson J, Reid A, Runciman C, Wadler P (2003) Haskell 98 language and libraries. The revised report. Cambridge University Press, Cambridge
 11. Pointon R, Trinder P, Loidl H-W (2000) The design and implementation of Glasgow distributed Haskell. Proceedings of the IFL'00 – 12th International Workshop on the Implementation of Functional Languages, Springer LNCS 2011, Aachen, Germany, pp 53–70
 12. Rauber Du Bois A, Trinder P, Loidl H (2005) mHaskell: mobile computation in a purely functional language. J Univ Computer Sci 11(7):1234–1254
 13. Roe P (1991) Parallel programming using functional languages. PhD thesis, Department of Computing Science, University of Glasgow
 14. Trinder P, Hammond K, Loidl H-W, Peyton Jones S (1998) Algorithm + strategy = parallelism. J Funct Prog 8(1):23–60
 15. Trinder P, Hammond K, Mattson J Jr, Partridge A, Peyton Jones S (1996) GUM: a portable parallel implementation of Haskell. Proceedings of the PLDI'96 – ACM Conference on Programming Language Design and Implementation, Philadelphia, pp 79–88
 16. Trinder P, Loidl H-W, Pointon R (2002) Parallel and distributed Haskell. J Funct Prog 12(4&5):469–510. Special Issue on Haskell

Global Arrays

► Global Arrays Parallel Programming Toolkit

Global Arrays Parallel Programming Toolkit

JAREK NIEPLOCHA^{1,†}, MANOJKUMAR KRISHNAN¹,
 BRUCE PALMER¹, VINOD TIPPARAJU²,
 ROBERT HARRISON², DANIEL CHAVARRÍA-MIRANDA¹
¹Pacific Northwest National Laboratory, Richland,
 WA, USA
²Oak Ridge National Laboratory, Oak Ridge, TN, USA

Synonyms

Global arrays; GA

[†]deceased.

Definition

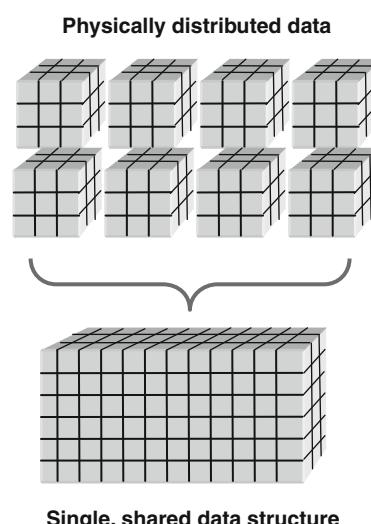
Global Arrays is a high-performance programming model for scalable, distributed-memory, parallel computer systems. Global Arrays is based on the concept of globally accessible dense arrays that are logically shared, yet physically distributed onto the memories of a parallel distributed computer system (Fig. 1 illustrates this concept).

Discussion

Introduction

Global Arrays (GA) is a high-performance programming model for scalable, distributed-memory, parallel computer systems. GA is a library-based Partitioned Global Address Space (PGAS) programming model. The underlying supported sequential languages are Fortran, C, C++, and Python. GA provides *global view access* to very large dense arrays through API functions implemented for those languages, under a Single Program Multiple Data (SPMD) execution environment.

GA was originally developed as part of the underlying software infrastructure for the US Department of Energy's NWChem computational chemistry software package. Over time, it has been developed into a standalone package with a rich set of API functions (200+) that cater to many needs in scientific application development. GA has been used to enable scalable



Global Arrays Parallel Programming Toolkit. Fig. 1 Dual view of Global Arrays data structures

parallel execution for several major scientific applications including NWChem (computational chemistry, specifically electronic structure calculation), STOMP (Subsurface Transport Over Multiple Phases, a subsurface flow and transport simulator), ScalaBLAST (a more scalable, higher-performance version of BLAST), Molpro (quantum chemistry), TE²THYS (unstructured, implicit CFD and coupled fluid/solid mechanics finite volume code), Pagoda (Parallel Analysis of Geodesic Data), COLUMBUS (computational chemistry), and GAMESS-UK (computational chemistry).

GA's development has occurred over the last two decades. For this reason, the number of people involved and their contributions is large. GA's original development occurred as a co-design effort between the NWChem team and the computer science team focused on GA. The main designer and original developer of GA was Jarek Nieplocha. Robert Harrison led the main effort in the development of NWChem.

Basic Global Arrays

There are three classes of operations in Global Arrays: core operations, task parallel operations, and data-parallel operations. These operations have multiple language bindings, but provide the same functionality independent of the language. The current GA library contains approximately 200 operations that provide a rich set of functionality related to data management and computations involving distributed arrays. GA is interoperable with MPI, enabling the development of hybrid programs that use both programming models.

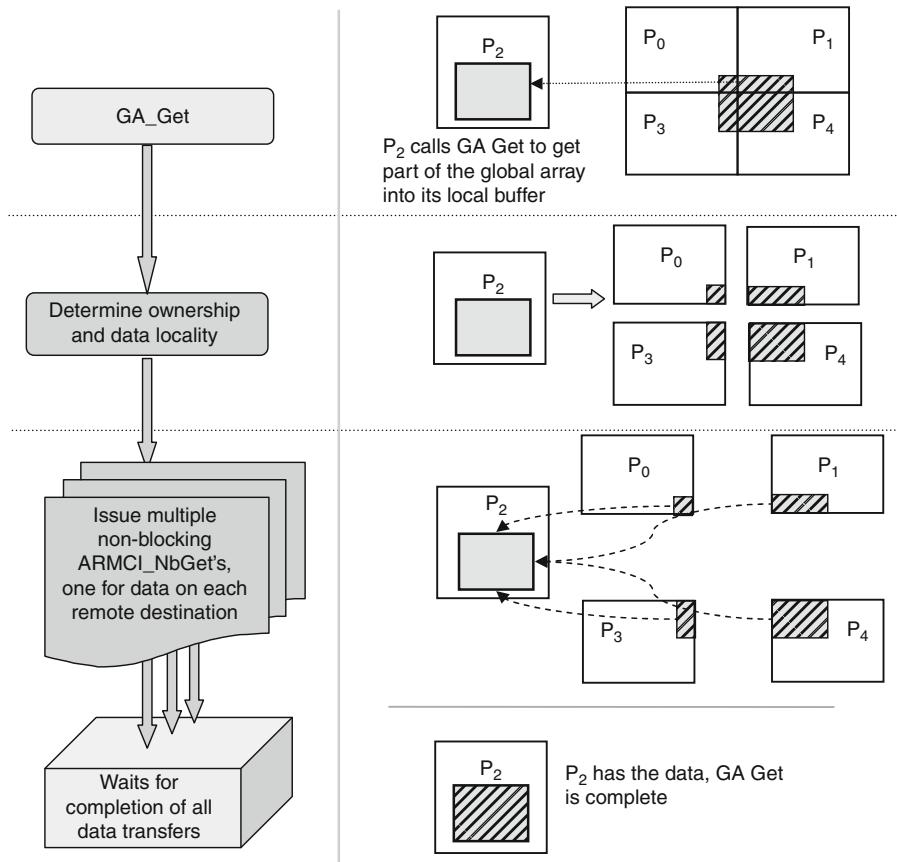
The basic components of the Global Arrays toolkit are function calls to create global arrays, copy data to, from, and between global arrays, and identify and access the portions of the global array data that are held locally. There are also functions to destroy arrays and free up the memory originally allocated to them. The basic function call for creating new global arrays is **nga_create**. The arguments to this function include the dimension of the array, the number of indices along each of the coordinate axes, and the type of data (integer, float, double, etc.) that each array element represents. The function returns an integer handle that can be used to reference the array in all subsequent operations. The allocation of data can be left completely to the toolkit, but if it is desirable to control the distribution

of data for load balancing or other reasons, additional versions of the **nga_create** function are available that allow the user to specify in detail how data is distributed between processors. The basic **nga_create** call provides a simple mechanism to control data distribution via the specification of an array that indicates the minimum dimensions of a block of data on each processor.

One of the most important features of Global Arrays is the ability to easily move blocks of data between global arrays and local buffers. The data in the global array can be referred to using a global indexing scheme and data can be moved in a single function call, even if it represents data distributed over several processors. The **nga_get** function can be used to move a block of distributed data from a global array to a local buffer. The arguments consist of the array handle for the array that data is being taken from, two integer arrays representing the lower and upper indices that bound the block of distributed data that is going to be moved, a pointer to the local buffer or a location in the local buffer that is to receive the data, and an array of strides for the local data. The **nga_put** call is similar and can be used to move data in the opposite direction.

The number of basic GA operations is fairly small and many parallel programs can be written with just the following ten routines:

- **GA_Initialize ()**: Initialize the GA library.
- **GA_Terminate ()**: Release internal resources and finalize execution of a GA program.
- **GA_Nnodes ()**: Return the number of GA compute processes (corresponds to the SPMD execution environment).
- **GA_Nodeid ()**: Return the GA process ID of the calling compute process, this is a number between 0 and **GA_Nnodes () - 1**.
- **NGA_Create ()**: Create an *n*-dimensional globally accessible dense array (global array instance).
- **NGA_Destroy ()**: Deallocate memory and resources associated with a global array instance.
- **NGA_Put ()**: Copy data from a local buffer to an array section within a global array instance in a one-sided manner.
- **NGA_Get ()**: Copy data from an array section within a global array instance to a local buffer in a



Global Arrays Parallel Programming Toolkit. Fig. 2 Left: `GA_Get` flow chart. Right: An example: Process P_2 issues `GA_Get` to get a chunk of data, which is distributed (partially) among P_0, P_1, P_3 , and P_4 (owners of the chunk)

one-sided manner (see Fig. 2 for a detailed description of how `get()` operates).

- `GA_Sync()`: Synchronize compute processes via a barrier and ensure that all pending GA operations are complete (in accordance to the GA *consistency model*).
- `NGA_Distribution()`: Returns the array section owned by a specified compute process.

Example GA Program

We present a parallel matrix multiplication program written in Global Arrays using the Fortran language interface. It uses most of the basic GA calls described before, in addition to some more advanced calls to create global array instances with specified data distributions. The program computes the result of $C = A \times B$. Some variable declarations have been omitted for brevity.

Discussion on the Example Program

The program is (mostly) a fully functional GA code, except for omitted variable declarations. It creates global array instances with specific data distributions, illustrates the use of the `nga_put()` and `nga_get()` primitives, as well as locality information through the `nga_distribution()` call. The code includes calls to initialize and terminate the MPI library, which are needed to provide the SPMD execution environment to the GA application. (It is possible to write a GA application that does not call the MPI library through the use of the TCGMSG simple message-passing environment included with GA.) The code includes the creation and use of local buffers to be used as sources and targets for put and get operations (1A, 1B, 1C), which in this case were allocated as Fortran 90 dynamic arrays.

Lines 43–69 contain the principal part of the example code and illustrate several of the features of

```
1 program matmul
2 integer :: sz
3 integer :: i, j, k, pos, g_a, g_b, g_c
4 integer :: nproc, me, ierr
5 integer, dimension(2) :: dims, nblock, chunks
6 integer, dimension(1) :: lead
7 double precision, dimension(:,:,:), pointer :: lA, lB
8 double precision, dimension(:,:,:), pointer :: lC
9 call mpi_init(ierr)

10 call ga_initialize()
11 nproc = ga_nnodes()
12 me = ga_nodeid()

13 if (me .eq. 0) then
14    write (*, *) 'Running on: ', nproc, ' processors'
15 end if

16 dims(:) = sz
17 chunks(:) = sz/sqrt(dble(nproc)) ! only runs on a perfect square number
   of processors
18 nblock(1) = sz/chunks(1)
19 nblock(2) = sz/chunks(2)
20 allocate(dmap(nblock(1) + nblock(2)))

21 pos = 1
22 do i = 1, sz - 1, chunks(1) ! compute beginning coordinate of each
   partition in the 1st dimension
23    dmap(pos) = i
24    pos = pos + 1
25 end do

26 do j = 1, sz - 1, chunks(2) ! compute beginning coordinate of each
   partition in the 2nd dimension
27    dmap(pos) = j
28    pos = pos + 1
29 end do

30 ret = nga_create_irreg(MT_DBL, ubound(dims), dims, 'A', dmap, nblock, g_a) !
   create a global array instance with specified data distribution
31 ret = ga_duplicate(g_a, g_b, 'B') ! duplicate same data distribution for
   array B
32 ret = ga_duplicate(g_a, g_c, 'C') ! and C

33 allocate(lA(chunks(1), chunks(2)), lB(chunks(1), chunks(2)),
   lC(chunks(1), chunks(2)))
34 lA(:, :) = 1.0
```

```
35  lB(:, :) = 2.0
36  lC(:, :) = 0.0
37  lead(1) = chunks(2)

38 call nga_distribution(g_a, me, tcoords1, tcoordsh)

39 ! initialize global array instances to respective values
40 call nga_put(g_a, tcoords1, tcoordsh, lA(1, 1), lead)
41 call nga_put(g_b, tcoords1, tcoordsh, lB(1, 1), lead)
42 call nga_put(g_c, tcoords1, tcoordsh, lC(1, 1), lead)

43 ! obtain all blocks in the row of the A matrix
44 tcoords1(1) = 1
45 tcoords1(2) = tcoords1(2)
46 tcoordsh1(1) = chunks(1)
47 tcoordsh1(2) = tcoordsh(2)

48 ! obtain all blocks in the column of the B matrix
49 tcoords1(1) = tcoords1(1)
50 tcoords1(2) = 1
51 tcoordsh2(1) = tcoordsh(1)
52 tcoordsh2(2) = chunks(2)

53 do pos = 1, nblock(1) ! matrix is square
54   call nga_get(g_a, tcoords1, tcoordsh1, lA(1, 1), lead)
55   call nga_get(g_b, tcoords1, tcoordsh2, lB(1, 1), lead)

56   do j = 1, n
57     do k = 1, n
58       do i = 1, n
59         lC(i, j) = lC(i, j) + lA(i, k) * lB(k, j)
60       end do
61     end do

62   ! advance coordinates for blocks
63   tcoords1(1) = tcoords1(1) + chunks(1)
64   tcoordsh1(1) = tcoordsh1(1) + chunks(1)

65   tcoords1(2) = tcoords1(2) + chunks(2)
66   tcoordsh2(2) = tcoordsh2(2) + chunks(2)
67 end do
68 ! lC contains the final result for the block owned by the process
69 call nga_put(g_c, tcoords1, tcoordsh, lC(1, 1), lead)

70 ! do something with the result
71 call ga_print(g_c)
72 deallocate(dmap, lA, lB, lC)
```

```

73 ret = ga_destroy(g_a)
74 ret = ga_destroy(g_b)
75 ret = ga_destroy(g_c)
76 call ga_terminate()
77 call mpi_finalize(ierr)

78 end program matmul

```

GA: data is being accessed using *global coordinates* (lines 54–55 and 63–66) that correspond to the global size of the global array instances that were created previously; in addition, the access to the global array data for arrays *g_a* and *g_b* in lines 54 and 55 is done *without requiring the participation* of the process where that data is allocated (one-sided access). [Figure 3](#) illustrates the concept of one-sided access.

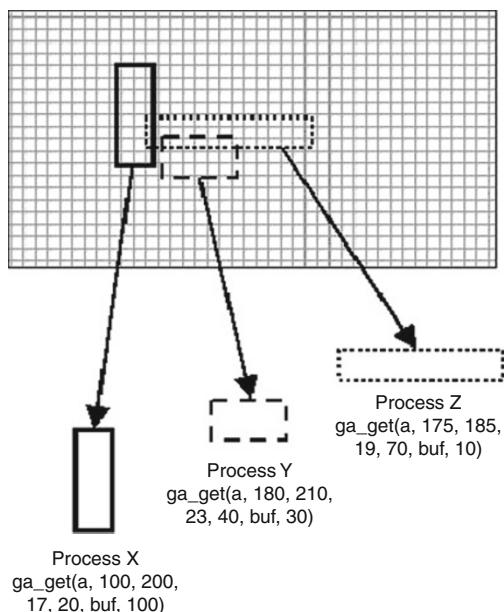
Global Arrays Concepts

GA allows the programmer to control data distribution and makes the locality information readily available to be exploited for performance optimization. For example, global arrays can be created by: (1) allowing the

library to determine the array distribution, (2) specifying the decomposition for only one array dimension and allowing the library to determine the others, (3) specifying the distribution block size for all dimensions, or (4) specifying an irregular distribution as a Cartesian product of irregular distributions for each axis. The distribution and locality information is always available through interfaces that allow the application developer to query: (1) which data portion is held by a given process, (2) which process owns a particular array element, and (3) a list of processes and the blocks of data owned by each process corresponding to a given section of an array.

The primary mechanisms provided by GA for accessing data are block copy operations that transfer data between layers of memory hierarchy, namely, global memory (distributed array) and local memory. Further, extending the benefits of using blocked data accesses and copying remote locations into contiguous local memory can improve cache performance by reducing both conflict and capacity misses [1]. In addition, each process is able to access directly the data held in a section of a Global Array that is locally assigned to that process. Data representing sections of the Global Array owned by other processes on SMP clusters can also be accessed directly using the GA interface, if desired. Atomic operations are provided that can be used to implement synchronization and assure correctness of an accumulate operation (floating-point sum reduction that combines local and remote data) executed concurrently by multiple processes and targeting overlapping array sections.

GA is extensible as well. New operations can be defined exploiting the low-level interfaces dealing with distribution, locality, and providing direct memory access (`nga_distribution`, `nga_locate_region`, `nga_access`, `nga_release`, `nga_release_update`). These, e.g., were used to provide



Global Arrays Parallel Programming Toolkit. Fig. 3 Any part of GA data can be accessed independently by any process at any time

additional linear algebra capabilities by interfacing with third-party libraries, e.g., ScaLAPACK [2].

Global Arrays Memory Consistency Model

In shared-memory programming, one of the issues central to performance and scalability is memory consistency. Although the sequential consistency model [3] is straightforward to use, weaker consistency models [4] can offer higher performance on modern architectures and they have been implemented on actual hardware. GA's nature as a one-sided, global-view programming model requires similar attention to memory consistency issues. The GA approach is to use a weaker-than-sequential consistency model that is still relatively straightforward to understand by an application programmer. The main characteristics of the GA approach include:

- GA distinguishes two types of completion of the store operations (i.e., put, scatter) targeting global shared memory: local and remote. The blocking store operation returns after the operation is completed locally, i.e., the user buffer containing the source of the data can be reused. The operation completes remotely after either a memory fence operation or a barrier synchronization is called. The fence operation is required in critical sections of the user code, if the globally visible data is modified.
- The blocking operations (get/put) are ordered only if they target overlapping sections of global arrays. Operations that do not overlap or access different arrays can complete in arbitrary order.
- The nonblocking get/put operations complete in arbitrary order. The programmer uses wait/test operations to order completion of these operations, if desired.

Global Arrays Extensions

To allow the user to exploit data locality, the toolkit provides functions identifying the data from the global array that is held locally on a given processor. Two functions are used to identify local data. The first is the **nga_distribution** function, which takes a processor ID and an array handle as its arguments and returns a set of lower and upper indices in the global address space representing the local data block. The second is the **nga_access** function, which returns an array index and an array of strides to the locally held

data. In Fortran, this can be converted to an array by passing it through a subroutine call. The C interface provides a function call that directly returns a pointer to the local data.

In addition to the communication operations that support task parallelism, the GA toolkit includes a set of interfaces that operate on either entire arrays or sections of arrays in the data-parallel style. These are collective data-parallel operations that are called by all processes in the parallel job. For example, movement of data between different arrays can be accomplished using a single function call. The **nga_copy_patch** function can be used to move a patch, identified by a set of lower and upper indices in the global index space, from one global array to a patch located within another global array. The only constraints on the two patches are that they contain equal numbers of elements. In particular, the array distributions do not have to be identical, and the implementation can perform, as needed, the necessary data reorganization (the so-called MxN problem [5]). In addition, this interface supports an optional transpose operation for the transferred data. If the copy is from one patch to another on the same global array, there is an additional constraint that the patches do not overlap.

Historical Development and Comparison with Other Programming Models

The original GA package [6–8] offered basic one-sided communication operations, along with a limited set of collective operations on arrays in the style of BLAS [9]. Only two-dimensional arrays and two data types were supported. The underlying communication mechanisms were implemented on top of vendor-specific interfaces. In the course of 10 years, the package evolved substantially and the underlying code was completely rewritten. This included separation of the GA internal one-sided communication engine from the high-level data structure. A new portable, general, and GA-independent communication library called ARMCI was created [10]. New capabilities were later added to GA without the need to modify the ARMCI interfaces. The GA toolkit evolved in multiple directions:

- Adding support for a wide range of data types and virtually arbitrary array ranks.
- Adding advanced or specialized capabilities that address the needs of some new application areas,

e.g., ghost cells or operations for sparse data structures.

- Expansion and generalization of the existing basic functionality. For example, mutex and lock operations were added to better support the development of shared-memory-style application codes. They have proven useful for applications that perform complex transformations of shared data in task parallel algorithms, such as compressed data storage in the multireference configuration interaction calculation in the COLUMBUS package [11].
- Increased language interoperability and interfaces. In addition to the original Fortran interface, C, Python, and a C++ class library were developed.
- Developing additional interfaces to third-party libraries that expand the capabilities of GA, especially in the parallel linear algebra area: ScaLAPACK [2] and SUMMA [12]. Interfaces to the TAO optimization toolkit have also been developed [13].
- Developed support for multilevel parallelism based on processor groups in the context of a shared-memory programming model, as implemented in GA [14, 15].

These advances generalized the capabilities of the GA toolkit and expanded its appeal to a broader set of applications. At the same time, the programming model, with its emphasis on a shared-memory view of the data structures in the context of distributed memory systems with a hierarchical memory, is as relevant today as it was in 1993 when the project started.

Comparison with Other Programming Models

The two predominant classes of programming models for parallel computers are distributed-memory, shared-nothing, and Uniform Memory Access (UMA) shared-everything models. Both the shared-everything and fully distributed models have advantages and shortcomings. The UMA shared-memory model is easier to use but it ignores data locality/placement. Given the hierarchical nature of the memory subsystems in modern computers, this characteristic can have a negative impact on performance and scalability. Careful code restructuring to increase data reuse and replacing fine-grained load/stores with block access to shared data can address the problem and yield performance for shared memory that is competitive with message passing [16].

However, this performance comes at the cost of compromising the ease of use that the UMA shared-memory model posits. Distributed, shared-nothing memory models, such as message-passing or one-sided communication, offer performance and scalability but they are more difficult to program. The classic message-passing paradigm not only transfers data but also synchronizes the sender and receiver. Asynchronous (nonblocking) send/receive operations can be used to diffuse the synchronization point, but cooperation between sender and receiver is still required. The synchronization effect is beneficial in certain classes of algorithms, such as parallel linear algebra, where data transfer usually indicates completion of some computational phase; in these algorithms, the synchronizing messages can often carry both the results and a required dependency. For other algorithms, this synchronization can be unnecessary and undesirable, and a source of performance degradation and programming complexity.

The Global Arrays toolkit [6–8] attempts to offer the best features of both models. It implements a *global-view* programming model, based on one-sided communication, in which data locality is managed by the programmer. This management is achieved by calls to functions that transfer data between a global address space (a distributed array) and local storage. In this respect, the GA model has similarities to the distributed shared-memory models that provide, e.g., an explicit acquire/release protocol [17]. However, the GA model acknowledges that remote data is slower to access than local data and allows data locality to be specified by the programmer and hence managed. GA is related to the global address space languages such as UPC [18], Titanium [19], and, to a lesser extent, Co-Array Fortran [20]. In addition, by providing a set of data-parallel operations, GA is also related to data-parallel languages such as HPF [21], ZPL [22], and Data Parallel C [23]. However, the Global Array programming model is implemented as a library that works with most languages used for technical computing and does not rely on compiler technology for achieving parallel efficiency. It also supports a combination of task and data parallelism and is fully interoperable with the message-passing (MPI) model. The GA model exposes to the programmer the hierarchical memory of modern high-performance computer systems [24], and by recognizing the communication overhead for remote data

transfers, it promotes data reuse and locality of reference. Virtually all scalable architectures possess nonuniform memory access characteristics that reflect their multilevel memory hierarchies. These hierarchies typically comprise processor registers, multiple levels of cache, local memory, and remote memory. Over time, both the number of levels and the cost (in processor cycles) of accessing deeper levels have been increasing. Scalable programming models must address memory hierarchy since it is critical to the efficient execution of applications.

Related Entries

- [Coarray Fortran](#)
- [MPI \(Message Passing Interface\)](#)
- [PGAS \(Partitioned Global Address Space\) Languages](#)
- [UPC](#)

Bibliographic Notes and Further Reading

A more detailed version of this entry has been published in the *International Journal of High Performance Computing Applications*, vol. 20, no. 2, May 2006 by SAGE Publications, Inc., All rights reserved. © 2006.

Bibliography

1. Lam MS, Rothberg EE, Wolf ME (1991) Cache performance and optimizations of blocked algorithms. In: Proceedings of the 4th international conference on architectural support for programming languages and operating systems, Santa Clara, 8–11 Apr 1991
2. Blackford LS, Choi J, Cleary A, D'Azevedo E, Demmel J, Dhillon I, Dongarra J, Hammarling S, Henry G, Petitet A, Stanley K, Walker D, Whaley RC (1997) ScaLAPACK: a linear algebra library for message-passing computers. In: Proceedings of eighth SIAM conference on parallel processing for scientific computing, Minneapolis
3. Scheurich C, Dubois M (1987) Correct memory operation of cache-based multiprocessors. In: Proceedings of 14th annual international symposium on computer architecture, Pittsburgh
4. Dubois M, Scheurich C, Briggs F (1986) Memory access buffering in multiprocessors. In: Proceedings of 13th annual international symposium on Computer architecture, Tokyo, Japan
5. CCA-Forum. Common component architecture forum. <http://www.cca-forum.org>
6. Nieplocha J, Harrison RJ, Littlefield RJ (1994) Global arrays: a portable shared memory programming model for distributed memory computers. In: Proceedings of Supercomputing, Washington, DC, pp 340–349
7. Nieplocha J, Harrison RJ, Littlefield RJ (1996) Global arrays: A nonuniform memory access programming model for high-performance computers. *J Supercomput* 10:169–189
8. Nieplocha J, Harrison RJ, Krishnan M, Palmer B, Tipparaju V (2002) Combining shared and distributed memory models: Evolution and recent advancements of the Global Array Toolkit. In: Proceedings of POHLL' 2002 workshop of ICS-2002, New York
9. Dongarra JJ, Croz JD, Hammarling S, Duff I (1990) Set of level 3 basic linear algebra subprograms. *ACM Trans Math Softw* 16:1–17
10. Nieplocha J, Carpenter B (1999) ARMCI: a portable remote memory copy library for distributed array libraries and compiler runtime systems. In: Proceedings of RTSPP of IPPS/SDP'99, San Juan, Puerto Rico
11. Dachsel H, Nieplocha J, Harrison RJ (1998) An out-of-core implementation of the COLUMBUS massively-parallel multireference configuration interaction program. In: Proceedings of high performance networking and computing conference, SC'98, Orlando
12. VanDeGeijn RA, Watts J (1997) SUMMA: Scalable universal matrix multiplication algorithm. *Concurr Pract Exp* 9:255–274
13. Benson S, McInnes L, Moré JJ Toolkit for Advanced Optimization (TAO). <http://www.mcs.anl.gov/tao>
14. Nieplocha J, Krishnan M, Palmer B, Tipparaju V, Zhang Y (2005) Exploiting processor groups to extend scalability of the GA shared memory programming model. In: Proceedings of ACM computing frontiers, Italy
15. Krishnan M, Alexeev Y, Windus TL, Nieplocha J (2005) Multilevel parallelism in computational chemistry using common component architecture and global arrays. In: Proceedings of Supercomputing, Seattle
16. Shan H, Singh JP (2000) A comparison of three programming models for adaptive applications on the origin2000. In: Proceedings of supercomputing, Dallas
17. Zhou Y, Iftode L, Li K (1996) Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In: Proceedings of operating systems design and implementation symposium, Seattle, pp 75–88
18. Carlson WW, Draper JM, Culler DE, Yelick K, Brooks E, Warren K (1999) Introduction to UPC and language specification. Center for Computing Sciences CCS-TR-99-157, IDA Center for Computing Sciences, Bowie
19. Yelick K, Semenzato L, Pike G, Miyamoto C, Liblit B, Krishnamurthy A, Hilfinger P, Graham S, Gay D, Colella P, Aiken A (1998) Titanium: A high-performance Java dialect. *Concurr Pract Exp* 10:825–836
20. Numrich RW, Reid JK (1998) Co-array Fortran for parallel programming. *ACM Fortran Forum* 17:1–31
21. High Performance Fortran Forum (1993) High Performance Fortran Language Specification, version 1.0. *Sci Program* 2(1):1–170
22. Snyder L (1999) A programmer's guide to ZPL. MIT Press, Cambridge
23. Hatcher PJ, Quinn MJ (1991) Data-parallel programming on MIMD computers. MIT Press, Cambridge
24. Nieplocha J, Harrison RJ, Foster I (1996) Explicit management of memory hierarchy. *Adv High Perform Comput* 185–200

Gossiping

► [Allgather](#)

GpH (Glasgow Parallel Haskell)

► [Glasgow Parallel Haskell \(GpH\)](#)

GRAPE

JUNICHIRO MAKINO

National Astronomical Observatory of Japan, Tokyo,
Japan

Definition

GRAPE (GRAvity PipE) is the name of a series of special-purpose computers designed for the numerical simulation of gravitational many-body systems. Most of GRAPE machines consist of hardwired pipeline processors to calculate the gravitational interaction between particles and programmable computers to handle all other works. GRAPE-DR (Greatly Reduced Array of Processor Elements with Data Reduction) replaced the hardwired pipeline by simple SIMD programmable processors.

Discussion

Introduction

The improvement in the speed of computers has been a factor of 100 in every decade, for the last 60 years. In these 60 years, however, the computer architecture has become more and more complex. Pipelined architecture were introduced in 1960s, and vector architectures became the mainstream in 1970s. In 1980s, a number of parallel architectures appeared, but in the 1990s and 2000s, distributed memory parallel computers built from microprocessors have taken over.

The technological driving force of this evolution of computer architecture has been the increase of the number of available transistors in integrated circuits, at least after the invention of integrated circuits in 1960s. In the case of CMOS LSIs, the number of transistors in

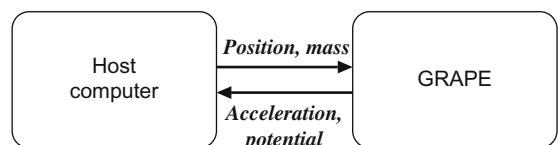
a chip doubles in every 18 months. Under the assumption of so-called CMOS scaling, this means that the switching speed doubles in every 36 months. In the last 30 years, the number of transistors available on an LSI chip increased by roughly a factor of one million.

One way to make use of this huge number of transistors is to implement application-specific pipeline processors into a chip. The GRAPE series of special-purpose computers is one of such efforts to make efficient use of large number of transistors available on LSIs.

In many scientific simulations, it is necessary to solve N -body problems numerically. The gravitational N -body problem is one such example, which describes the evolution of many astronomical objects from the solar system to the entire universe. In some cases, it is important to treat non-gravitational effects such as the hydrodynamical interaction, radiation, and magnetic fields, but the gravity is the primary driving force that shapes the universe.

To solve the gravitational N -body problem, one needs to calculate the gravitational force on each body (particle) in the system from all other particles in the system. There are many ways to do so, and if relatively low accuracy is sufficient, one can use the Barnes-Hut tree algorithm [3] or FMM [5]. Even with these schemes, the calculation of the gravitational interaction between particles (or particles and multipole expansions of groups of particles) is the most time-consuming part of the calculation. Thus, one can greatly improve the speed of the entire simulation, just by accelerating the speed of the calculation of particle-particle interaction. This is the basic idea behind GRAPE computers.

The basic idea is shown in Fig. 1. The system consists of a host computer and special-purpose hardware, and the special-purpose hardware handles the calculation of gravitational interaction between particles. The host computer performs other calculations such as the time integration of particles, I/O, and diagnostics.



GRAPE. Fig. 1 Basic structure of a GRAPE system

This architecture accelerates not only the simple algorithm in which the force on a particle is calculated by taking the summation of forces from all other particles in the system, but also the Barnes–Hut tree algorithms and FMM. Moreover, it can be used with individual timestep algorithms [1], in which particles have their own times and timesteps and integrated in an event-driven fashion. The use of individual timestep is critical in simulations of many systems including star clusters and planetary systems, where close encounters and physical collisions of two particles require very small timesteps for a small number of particles.

History

The GRAPE project started in 1988. The first machine completed, the GRAPE-1 [7], was a single-board unit on which around 100 IC and LSI chips were mounted and wire-wrapped. The pipeline processor of GRAPE-1 was implemented using commercially available IC and LSI chips. It was a natural consequence of the fact that project members lacked both money and experience to design custom LSI chips. In fact, none of the original design and development team of GRAPE-1 had the knowledge of electronic circuit more than what was learned in basic undergraduate course for physics students.

For GRAPE-1, an unusually short word format was used, to make the hardware as simple as possible. The input coordinates are expressed in 15-bit fixed point format. After subtraction, the result is converted to 8-bit logarithmic format, in which 3 bit are used for the “fractional” part. This format is used for all following operations except for the final accumulation. The final accumulation was done in 48-bit fixed point, to avoid overflow and underflow. The advantage of the short word format is that ROM chips can be used to implement complex functions that require two inputs. Any function of two 8-bit words can be implemented by one ROM chip with 16-bit address input. Thus, all operations other than the initial subtraction of the coordinates and final accumulation of the force were implemented by ROM chips.

The use of extremely short word format in GRAPE-1 was based on the detailed theoretical analysis of error propagation and numerical experiment [13]. There are three dominant sources of error in numerical simulations of gravitational many-body systems. The first one

is the error in the numerical integration of orbits of particles. The second one is the error in the calculated accelerations themselves. The third one comes from the fact that in many cases, the number of particles used is much smaller than the number of stars in the real systems such as a galaxy.

Whether or not the third one should be regarded as the source of error depends on the problem one wants to study. If the problem is, for example, merging of two galaxies, which takes place in relatively short timescale (compared to the orbital period of typical stars in galaxies), the effect of small number of particles can be, and should be, regarded as the numerical error.

On the other hand, if we want to study long-term evolution of a star cluster, which takes place in the timescale much longer than the orbital timescale, the evolution of orbits of individual stars is driven by close encounters with other stars. In this case, the effect of small (or finite) number of particles is not an numerical error but what is there in real systems.

Thus, the required accuracy of pairwise force calculation depends on the nature of the problem. In the case of the study of the merging of two galaxies, the average error can be as large as 100% of the pairwise force, if the error is guaranteed to be random. The average error of interaction calculated by GRAPE-1 is less than 10%, which was good enough for many problems. In the number format used in GRAPE-1, the positions of particles are expressed in the 16-bit fixed-point format, so that the force between two nearby particles, both far from the origin of coordinates, is still expressed with sufficient accuracy. Also, the accumulation of the forces from different particles is done in the 48-bit fixed-point format, so that there is no loss of effective bits during the accumulation. Thus, the primary source of error with GRAPE-1 is the low-accuracy pairwise force calculation. It can be regarded as random error.

Strictly speaking, the error due to the short word format cannot always be regarded as random, since it introduces correlation both in space and time. One could eliminate this correlation by applying random coordinate transformation, but the quantitative study of such transformation has not done yet.

GRAPE-1 used the GPIB (IEEE-488) interface for the communication with the host computer. It was fast enough for the use with simple direct summation. However, when combined with the tree algorithm, faster

communication was necessary. GRAPE-1A used VME bus for communication, to improve the performance. The speed of GRAPE-1 and 1A was around 300 Mflops, which is around 1/10 of the speed of fastest vector supercomputers of the time. Hardware cost of them was around 1/1,000 of supercomputers, or roughly equal to the cost of low-end workstations. Thus, these first-generation GRAPEs offered price-performance two orders of magnitude better than that of general-purpose computers.

GRAPE-2 is similar to GRAPE-1A, but with much higher numerical accuracy. In order to achieve higher accuracy, commercial LSI chips for floating-point arithmetic operations such as TI SN74ACT8847 and Analog Devices ADSP3201/3202 were used. The pipeline of GRAPE-2 processes the three components of the interaction sequentially. So it accumulates one interaction in every three clock cycles. This approach was adopted to reduce the circuit size. Its speed was around 40 Mflops, but it is still much faster than workstations or minicomputers at that time.

GRAPE-3 was the first GRAPE computer with a custom LSI chip. The number format was the combination of the fixed point and logarithmic format similar to what were used in GRAPE-1. The chip was fabricated using 1 μm design rule by National Semiconductor. The number of transistors on chip was 110 K. The chip operated at 20 MHz clock speed, offering the speed of about 0.8 Gflops. Printed-circuit boards with eight chips were mass-produced, for the speed of 6.4 Gflops per board. Thus, GRAPE-3 was also the first GRAPE computer to integrate multiple pipelines into a system. Also, GRAPE-3 was the first GRAPE computer to be manufactured and sold by a commercial company. Nearly 100 copies of GRAPE-3 have been sold to more than 30 institutes (more than 20 outside Japan).

With GRAPE-4, a high-accuracy pipeline was integrated into one chip. This chip calculates the first time derivative of the force, so that fourth-order Hermite scheme [10] can be used. Here, again, the serialized pipeline similar to that of GRAPE-2 was used. The chip was fabricated using 1 μm design rule by LSI Logic. Total transistor count was about 400 K.

The completed GRAPE-4 system consisted of 1,728 pipeline chips (36 PCB boards each with 48 pipeline chips). It operated on 32 MHz clock, delivering the speed of 1.1 Tflops. Completed in 1995, GRAPE-4 was

the first computer for scientific calculation to achieve the peak speed higher than 1 Tflops. Also, in 1995 and 1996, it was awarded the Gordon Bell Prize for peak performance, which is given to a real scientific calculation on a parallel computer with the highest performance. Technical details of machines from GRAPE-1 through GRAPE-4 can be found in [14] and references therein.

GRAPE-5 [9] was an improvement over GRAPE-3. It integrated two full pipelines which operate on 80 MHz clock. Thus, a single GRAPE-5 chip offered the speed eight times more than that of the GRAPE-3 chip, or the same speed as that of an eight-chip GRAPE-3 board. GRAPE-5 was awarded the 1999 Gordon Bell Prize for price-performance. The GRAPE-5 chip was fabricated with 0.35 μm design rule by NEC.

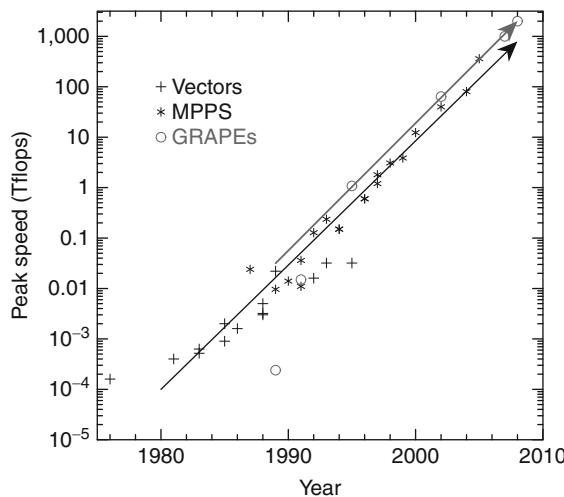
[Table 1](#) summarizes the history of GRAPE project. [Figure 2](#) shows the evolution of GRAPE systems and general-purpose parallel computers. One can see that evolution of GRAPE is faster than that of general-purpose computers.

The GRAPE-6 was essentially a scaled-up version of GRAPE-4 [15], with the peak speed of around 64 Tflops. The peak speed of a single pipeline chip was 31 Gflops. In comparison, GRAPE-4 consists of 1,728 pipeline chips, each with 600 Mflops. The increase of a factor of 50 in speed was achieved by integrating six pipelines into one chip (GRAPE-4 chip has one pipeline which needs three cycles to calculate the force from one particle) and using three times higher clock frequency. The advance of the device technology (from 1 μm to 0.25 μm) made these improvements possible. [Figure 3](#) shows the processor chip delivered in early 1999. The six pipeline units are visible.

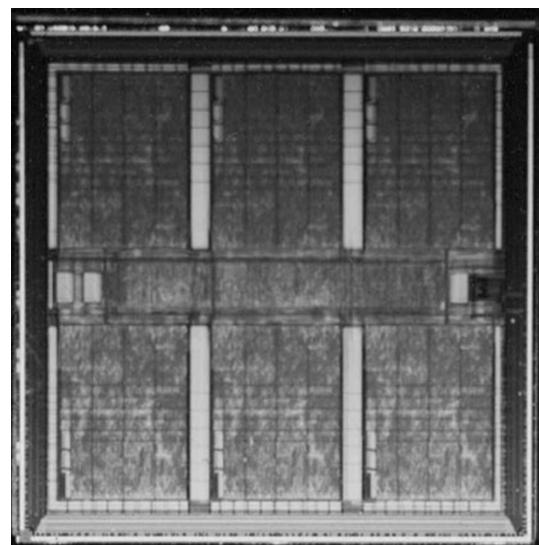
Starting with GRAPE-4, the concept of virtual multiple pipeline (VMP) is used. VMP is similar to simultaneous multithreading (SMT), in the sense that a single pipeline processor behaves as multiple processors. However, what is achieved is quite different. In the case of SMT, the primary gain is in the latency tolerance, since one can execute independent instructions with different threads. In the case of hardwired pipeline processors, there is no need to reduce the latency. With VMP, the bandwidth to the external memory is reduced, since the data of one particle which exerts force are shared by multiple virtual pipelines, each of which calculates the force on its own particle. This sharing of the

GRAPE. Table 1 History of GRAPE project

GRAPE-1	(89/4–89/10)	310 Mflops, low accuracy
GRAPE-2	(89/8–90/5)	50 Mflops, high accuracy(32 bit/64 bit)
GRAPE-1A	(90/4–90/10)	310 Mflops, low accuracy
GRAPE-3	(90/9–91/9)	18 Gflops, high accuracy
GRAPE-2A	(91/7–92/5)	230 Mflops, high accuracy
HARP-1	(92/7–93/3)	180 Mflops, high accuracy Hermite scheme
GRAPE-3A	(92/1–93/7)	8 Gflops/board some 80 copies are used all over the world
GRAPE-4	(92/7–95/7)	1 Tflops, high accuracy Some 10 copies of small machines
MD-GRAPE	(94/7–95/4)	1 Gflops/chip, high accuracy programmable interaction
GRAPE-5	(96/4–99/8)	5 Gflops/chip, low accuracy
GRAPE-6	(97/8–02/3)	64 Tflops, high accuracy

**GRAPE. Fig. 2** The evolution of GRAPE and general-purpose parallel computers. The peak speed is plotted against the year of delivery. Open circles, crosses, and stars denote GRAPEs, vector processors, and parallel processors, respectively

particle can be extended to physical multiple pipelines, as far as the total number of pipeline is not too large. Thus, special-purpose computers based on GRAPE-like pipeline have a unique advantage that their requirement of external memory bandwidth is much smaller than that of general-purpose computers with similar peak performance.

**GRAPE. Fig. 3** The GRAPE-6 processor chip

In the case of GRAPE-6, each of six physical pipelines is implemented as eight virtual pipelines. Thus, one GRAPE-6 chip calculates forces on 48 particles in parallel. The required memory bandwidth was 720 MB/s, for the peak speed of 31 Gflops. A traditional vector processor with the peak speed of 31 Gflops would require the memory bandwidth of 125

GB/s. Thus, GRAPE-6 requires the memory bandwidth around 1/200 of that of a traditional vector processor.

One processor board of GRAPE-6 housed 32 GRAPE-6 chips. Each GRAPE-6 chip has its own memory to store particles which exert the force. Thus, different processor chips on one processor board calculate the forces on the same 48 particles from different particles, and the partial results are summed up by an hardwired adder tree when the result is sent back to the host. Thus, the summation over 32 chips added only a small startup overhead (less than 1 μ s) per one force calculation, which typically requires several milliseconds.

The completed GRAPE-6 system consisted of 64 processor boards, grouped into 4 clusters with 16 boards each. Within a cluster, 16 boards are organized in a 4×4 matrix, with 4 host computers. They are organized so that the effective communication speed is proportional to the number of host computers. In a simple configuration, the effective communication speed becomes independent of the number of host computers. The details of the network used in GRAPE-6 are given in [11].

Machines for Molecular Dynamics

Classical MD calculation is quite similar to astrophysical N-body simulations since, in both cases, we integrate the orbit of particles (atoms or stars) which interact with other particles with simple pairwise force. In the case of Coulomb force, the force law itself is the same as that of the gravitational force, and the calculation of Coulomb force can be accelerated by GRAPE hardware.

However, in MD calculations, the calculation cost of van der Waals force is not negligible, though van der Waals force decays much faster than the Coulomb force (r^{-7} compared to r^{-2}).

It is straightforward to design a pipelined processor which can handle particle-particle force given by some arbitrary function of the distance between particles. In GRAPE-2A and its successors, a combination of table lookup and polynomial approximation is used.

GRAPE-2A and MD-GRAPE were developed in the University of Tokyo, following these lines of idea. GRAPE-2A was built using commercial chips and MD-GRAPE used a custom-designed pipeline chip.

Another difference between astrophysical simulations and MD calculations is that in MD calculations, usually the periodic boundary condition is applied. Thus, we need some way to calculate Coulomb forces

from image particles. The direct Ewald method is rather well suited for the implementation in hardware. In 1991, WINE-1 was developed. It is a pipeline to calculate the wave-space part of the direct Ewald method. The real-space part can be handled by GRAPE-2A or MD-GRAPE hardware.

In 1995, a group led by Toshikazu Ebisuzaki in RIKEN started to develop MDM [16], a massively parallel machine for large-scale MD simulations. Their primary goal was the simulation of protein molecules.

MDM consists of two special-purpose hardware, massively parallel version of MD-GRAPE (MDGRAPE-2) and that of WINE (WINE-2). The MDGRAPE-2 part consisted of 1,536 custom chips with four pipelines, for the theoretical peak speed of 25 Tflops. The WINE-2 part consists of 2,304 custom pipeline chips, for the peak speed of 46 Tflops.

The MDM effort was followed up by the development of MDGRAPE-3 [18], led by Makoto Taiji of RIKEN. MDGRAPE-3 achieved the peak speed of 1 Pflops in 2006.

Related Projects

The GRAPE project is not the first project to implement the calculation of pairwise force in particle-based simulations in hardware.

Delft Molecular Dynamics Processor [2] (DMDP) is one of the earliest efforts. It was completed in early 1980s. For the calculation of interaction between particles, it used the hardwired pipeline similar to that of GRAPE systems. However, in DMDP, time integration of orbits and other calculations are all done in the hardwired processors. Thus, in addition to the force calculation pipeline, DMDP had pipelines to update position, select particles for interaction calculation, and calculate diagnostics such as correlation function. FASTRUN [4] has the architecture similar to that of DMDP, but designed to handle more complex systems such as protein molecule.

To some extent, this difference in the designs of GRAPE computers and that of machines for molecular dynamics comes from the difference in the nature of the problem. In astronomy, the wide ranges in the number density of particles and timescale make it necessary to use adaptive schemes such as treecode and individual timesteps. With these schemes, the calculation cost per timestep per particle is generally higher than

that of fastest scheme optimized to shared timestep and near-uniform distribution of particles. The approach in which only the force calculation is done in hardware is more advantageous for astronomical N -body problems than for molecular dynamics.

Anton [17] is the latest effort to speed up the molecular dynamics simulation of proteins by specialized hardware. It is essentially the revival of the basic idea of DMDP, except that pipeline processors for operations other than force calculation were replaced by programmable parallel processors. It achieved the speed almost two orders of magnitude faster than that of general-purpose parallel computers for the simulation of protein molecules in water.

LSI Economics and GRAPE

GRAPE has achieved the cost performance much better than that of general-purpose computers. One reason for this success is simply that with GRAPE architecture, one can use practically all transistors for arithmetic units, without being limited by the memory wall problem. Another reason is the fact that arithmetic units can be optimized to their specific uses in the pipeline. For example, in the case of GRAPE-6, the subtraction of two positions is performed in 64-bit fixed point format, not in the floating-point format. Final accumulation is also done in fixed point. In addition, most of arithmetic operations to calculate the pairwise interactions are done in single precision. These optimizations made it possible to pack more than 200 arithmetic units into a single chip with less than 10 M transistors. The first microprocessor with fully pipelined double-precision floating-point unit, Intel 80860, required 1.2 M transistors for two (actually one and half) operations. Thus, the number of transistors per arithmetic unit of GRAPE is smaller by more than a factor of 10. When compared with more recent processors, the difference becomes even larger. The Fermi processor from NVIDIA integrates 512 arithmetic unit (adder and multiplier) with 3G transistors. Thus, it is five times less efficient than Intel 80860, and nearly 100 times less efficient than GRAPE-6. The difference in the power efficiency is even larger, because the requirement for the memory bandwidth is lower for GRAPE computers. As a result, performance per watt of GRAPE-6 chip, fabricated with the 250 nm design rule, is comparable to that of GPGPU chips fabricated with 40 nm design rule. Thus, as silicon

technology advances, the relative advantage of special-purpose architecture such as GRAPE becomes bigger.

However, there is another economical factor. As the silicon semiconductor technology advances, the initial cost to design and fabricate custom chip increases. In 1990, the initial cost for a custom chip was around 100 K USD. By 2000, it has become higher than 1 M USD. By 2010, the initial cost of a 45 nm chip is around 10 M USD. Roughly speaking, initial cost has been increasing as $n^{0.7}$, where n is the number of transistors one can fit into a chip.

The total budget for GRAPE-4 and GRAPE-6 projects is 3 and 4 M USD, respectively. Thus, a similar budget had become insufficient by early 2000s. The whole point of special-purpose computer is to be able to outperform “expensive” supercomputers, with the price of 10–100 M USD. Even if a special-purpose computer is 100–1,000 times faster, it is not practical to spend the cost of a supercomputer for a special-purpose computer which can solve only a narrow range of problems.

There are several possible solutions. One is to reduce the initial cost by using FPGA (Field-Programmable Gate Array) chips. An FPGA chip consists of a number of “programmable” logic blocks (LBs) and also “programmable” interconnections. A LB is essentially a small lookup table with multiple inputs, augmented with one flip-flop and sometimes full-adder or more additional circuits. The lookup table can express any combinatorial logic for input data, and with flip-flop, it can be part of a sequential logic. Interconnection network is used to make larger and more complex logic, by connecting LBs. The design of recent FPGA chips has become much more complex, with large functional units like memory blocks and multiplier (typically 18×18 bit) blocks.

Because of the need for the programmability, the size of the circuit that can be fit into an FPGA chip is much smaller than that for a custom LSI, and the speed of the circuit is also slower. Roughly speaking, the price of an FPGA chip per logic gate is around 50 times higher than that of a custom chip with the same design rule. If the relative advantage of a specialized architecture is much larger than this factor of 50, its implementation based on FPGA chips can outperform general-purpose computers.

In reality, there are quite a number of projects to use FPGAs for scientific computing, but most of them

turned out to be not competitive with general-purpose computers. The primary reason for this result is the relative cost of FPGA discussed above. Since the logic gate of FPGAs is much more expensive than that of general-purpose computers, the design of a special-purpose computer with FPGA must be very efficient in gate usage. FPGA-based systems which use standard double- or single-precision arithmetic are generally not competitive with general-purpose computers. In order to be competitive, it is necessary to use much shorter word length. GRAPE architecture with reduced accuracy is thus an ideal target for FPGA-based approach. Several successful approaches have been reported [6, 8].

GRAPE-DR

Another solution for the problem of the high initial cost is to widen the application range by some way to justify the high cost. GRAPE-DR project [12] followed that approach.

With GRAPE-DR, the hardwired pipeline processor of previous GRAPE systems was replaced by a collection of simple SIMD programmable processors. The internal network and external memory interface was designed so that it could emulate GRAPE processor efficiently and could be used for several other important applications, including the multiplication of dense matrices.

GRAPE-DR is an acronym of “Greatly Reduced Array of Processor Elements with Data Reduction.” The last part, “Data Reduction,” means that it has an on-chip tree network which can do various reduction operations such as summation, max/min, and logical and/or.

The GRAPE-DR project was started in FY 2004, and finished in FY 2008. The GRAPE-DR processor chip consists of 512 simple processors, which can operate at the clock cycle of 500 MHz, for the 512 Gflops of single precision peak performance (256 Gflops double precision). It was fabricated with TSMC 90 nm process and the size is around 300 mm². The peak power consumption is around 60 W. The GRAPE-DR processor board houses four GRAPE-DR chips, each with its own local DRAM chips. It communicates with the host computer through Gen1 16-lane PCI-Express interface.

To some extent, the difference between GRAPE and GRAPE-DR is similar to that between traditional GPUs and GPGPUs. In both cases, hardwired pipelines are replaced by simple programmable processors. The main

differences between GRAPE-DR and GPGPUs are (a) processor element of GRAPE-DR is much simpler, (b) external memory bandwidth of GRAPE-DR is much smaller, and (c) GRAPE-DR is designed to achieve near-peak performance in real scientific applications such as gravitational N-body simulation and molecular dynamics simulation, and also dense matrix multiplication. These differences made GRAPE-DR significantly more efficient in both transistor usage and power usage. GRAPE-DR chip, which was fabricated with 90 nm design rule and has 300 mm² area, integrates 512 processing elements. The NVIDIA Fermi chip, which is fabricated with 40 nm design rule and has > 500 mm² area, integrates the same 512 processing elements. Thus, there is about a factor of 10 difference in the transistor efficiency. This difference resulted in more than a factor of 2 difference in the power efficiency.

Whether or not the approach like GRAPE-DR will be competitive with other approaches, in particular GPGPUs, is at the time of writing rather unclear. The reason is simply that the advantage of a factor of 10 is not quite enough, because of the difference in other factors, among which the most important is the development cycle. New GPUs are announced roughly every year, while it is somewhat unlikely that one develops the special-purpose computers every year, even if there is sufficient budget. In 5 years, general-purpose computers become ten times faster, and GPGPUs will also become faster by a similar factor. Thus, a factor of 10 advantage will disappear while the machine is being developed. On the other hand, the transistor efficiency of general-purpose computers, and that of GPUs, has been decreasing for the last 20 years and probably will continue to do so for the next 10 years or so. GRAPE-DR can retain its efficiency when it is implemented with more advanced semiconductor technology, since, as in the case of GRAPE, one can use the increased number of transistors to increase the number of processor element. Thus, it might remain competitive.

Future Directions

Future of Special-Purpose Processors

In hindsight, 1990s was a very good period for the development of special-purpose architecture such as GRAPE, because of two reasons. First, the semiconductor technology reached the point where many

floating-point arithmetic units can be integrated into a chip. Second, the initial design cost of a chip was still within the reach of fairly small research projects in basic science.

By now, semiconductor technology reached to the point that one could integrate thousands of arithmetic units into a chip. On the other hand, the initial design cost of a chip has become too high.

The use of FPGAs and the GRAPE-DR approach are two examples of the way to tackle the problem of increasing initial cost. However, unless one can keep increasing the budget, GRAPE-DR approach is not viable, simply because it still means exponential increase in the initial, and therefore total, cost of the project.

On the other hand, such increase in the budget might not be impossible, since the field of computational science as a whole is becoming more and more important. Even though a supercomputer is expensive, it is still much less expensive compared to, for example, particle accelerators or space telescopes. Of course, computer simulation cannot replace the real experiments of observations, but computer simulations have become essential in many fields of science and technology.

In addition, there are several technologies available in between FPGAs and custom chips. One is what is called “structured ASIC.” It requires customization of typically just one metal layer, resulting in large reduction in the initial cost. The number of gates one can fit into the given silicon area falls between those of FPGAs and custom chips. Another possibility is just to use the technology one or two generations older.

Application Area of Special-Purpose Computers

Primary application area of GRAPE and GRAPE-DR has been the particle-based simulation, in particular that requires the evaluation of long-range interaction. It is suited to special-purpose computers because they are compute-intensive. In other words, the necessary bandwidth to the external memory is relatively small. Grid-based simulations based on schemes like finite-difference or finite-element methods are less compute-intensive and thus not suited to special-purpose computers.

However, the efficiency of large-scale parallel computers based on general-purpose microprocessors for these grid-based simulation has been decreasing rather quickly. There are two reasons for this decrease. One is the lack of the memory bandwidth. Currently, the memory bandwidth of microprocessors normalized by the calculation speed is around 0.3 bytes/flops, which is not enough for most of grid-based simulations. Even so, this ratio will become smaller and smaller in the future. The other reason is the latency of the communication between processors.

One possible solution for these two problems is to integrate the main memory, processor cores, and communication interface into a single chip. This integration gives practically unlimited bandwidth to the memory, and the communication latency is reduced by one or two orders of magnitude.

Obvious disadvantage of this approach is that the total amount of memory would be severely limited. However, in many application of grid-based calculation, very long time integrations of relatively small systems are necessary. Many of such applications requires memory much less than one TB, which can be achieved by using several thousand custom processors each with around 1 GB of embedded DRAM.

Bibliography

1. Aarseth SJ (1963) Dynamical evolution of clusters of galaxies, I. MN 126:223
2. Bakker AF, Gilmer GH, Grabow MH, Thompson K (1990) A special purpose computer for molecular dynamics calculations. J Comput Phys 90:313
3. Barnes J, Hut P (1986) A hierarchical O(NlogN) force calculation algorithm. Nature 324:446
4. Fine R, Dimmler G, Levinthal C (1991) FASTRUN: a special purpose, hardwired computer for molecular simulation. Proteins Struct Funct Genet 11:242
5. Greengard L, Rokhlin V (1987) A fast algorithm for particle simulations. J Comput Phys 73:325
6. Hamada T, Fukushige T, Kawai A, Makino J (1999) PROGRAPE-1: a programmable, multi-purpose computer for many-body simulations. PASJ 52:943–995
7. Ito T, Makino J, Ebisuzaki T, Sugimoto D (1990) A special-purpose N-body machine GRAPE-1. Comput Phys Commun 60:187
8. Kawai A, Fukushige T (2006) \$158/GFLOP Astrophysical N-body simulation with a reconfigurable add-in card and a hierarchical tree algorithm. In: Proceedings of SC06, ACM (Online)
9. Kawai A, Fukushige T, Makino J, Taiji M (2000) GRAPE-5: a special-purpose computer for N-body simulations. PASJ 52:659

10. Makino J, Aarseth SJ (1992) On a Hermite integrator with Ahmad-Cohen scheme for gravitational many-body problems. *PASJ* 44:141
11. Makino J, Fukushige T, Koga M, Namura K (2003) GRAPE-6: massively-parallel special-purpose computer for astrophysical particle simulations. *PASJ* 55:1163
12. Makino J, Hiraki K, Inaba M (2007) GRAPE-DR: 2-Pflops massively-parallel computer with 512-core, 512-Gflops processor chips for scientific computing. In: Proceedings of SC07, ACM (Online)
13. Makino J, Ito T, Ebisuzaki T (1990) Error analysis of the GRAPE-1 special-purpose N-body machine. *PASJ* 42:717
14. Makino J, Taiji M (1998) Scientific simulations with special-purpose computers – The GRAPE systems. Wiley, Chichester
15. Makino J, Taiji M, Ebisuzaki T, Sugimoto D (1997) GRAPE-4: a massively parallel special-purpose computer for collisional N-body simulations. *ApJ* 480:432
16. Narumi T, Susukita R, Ebisuzaki T, McNiven G, Elmegreen B (1999) *Mol Simul* 21:401
17. Shaw DE, Denero MM, Dror RO, Kuskin JS, Larson RH, Salmon JK, Young C, Batson B, Bowers KJ, Chao JC, Eastwood MP, Gagliardo J, Grossman JP, Ho CR, Ierardi DJ, Kolossváry I, Klepeis JL, Layman T, McLeavey C, Moraes MA, Mueller R, Priest EC, Shan Y, Spengler J, Theobald M, Towles B, Wang SC (2007) Anton: a special-purpose machine for molecular dynamics simulation. In: Proceedings of the 34th annual international symposium on computer architecture (ISCA '07), ACM, San Diego, pp 1–12
18. Taiji M, Narumi T, Ohno Y, Futatsugi N, Suenaga A, Takada N, Konagaya A (2003) Protein explorer: a petaflops special-purpose computer system for molecular dynamics simulations. In: The SC2003 Proceedings, IEEE, Los Alamitos, CD-ROM

Graph Algorithms

DAVID A. BADER¹, GUOJING CONG²

¹Georgia Institute of Technology, Atlanta, GA, USA

²IBM, Yorktown Heights, NY, USA

Discussion

Parallel Graph Algorithms

Relationships in real-world situations can often be represented as graphs. Efficient parallel processing of graph problems has been a focus of many algorithm researchers. A rich collection of parallel graph algorithms have been developed for various problems on different models. The majority of them are based on the parallel random access machine (PRAM). PRAM is a shared-memory model where data stored in the global memory can be accessed by any processor. PRAM is

synchronous, and in each unit of time, each processor either executes one instruction or stays idle.

Techniques

Graph problems are diverse. Given a graph $G = (V, E)$, where $|V| = n$ and $|E| = m$, several techniques are frequently used in designing parallel graph algorithms. The basic techniques are described as follows (Detailed descriptions can be found in [24]).

Prefix Sum

Given a sequence of n elements s_1, s_2, \dots, s_n with a binary associative operator denoted by \oplus , the prefix sums of the sequence are the partial sums defined by

$$S_i = s_1 \oplus s_2 \oplus \dots \oplus s_i, 1 \leq i \leq n$$

Using the balanced tree technique, the prefix sums of n elements can be computed in $O(\log n)$ time with $O(n/\log n)$ processors. Fast prefix sum is of fundamental importance to the design of parallel graph algorithms as it is frequently used in algorithms for more complex problems.

Pointer Jumping

Pointer jumping, also sometimes called path doubling, is useful for handling computation on rooted forests. For a rooted forest, there is a parent function P defined on the set of vertices. $P(r)$ is set to r when r does not have a parent. When finding the root of each tree, the pointer jumping technique updates the parent of each node by that node's grandparent, that is, set $P(r) = P(P(r))$. The algorithm runs in $O(\log n)$ time with $O(n)$ processors. Pointer jumping can also be used to compute the distance between each node to its root. Pointer jumping is used in several connectivity and tree algorithms (e.g., see [29, 31]).

Divide and Conquer

The divide-and-conquer strategy is recognized as a fundamental technique in algorithm design (not limited to parallel graph algorithms). The frequently used quicksort algorithm is based on divide and conquer. The strategy partitions the input into partitions of roughly equal size and recursively works on each partition concurrently. There is usually a final step to combine the solutions of the subproblems into a solution for the original problem.

Pipelining

Like divide-and-conquer, the use of pipelining is not limited to parallel graph algorithm design. It is of critical importance in computer hardware and software design. Pipelining breaks up a task into a sequence of smaller tasks (subtasks). Once a subtask is complete and moves to the next stage, the ones following it can be processed in parallel. Insertion and deletion with 2-3 trees demonstrate the pipelining technique [27].

Deterministic Coin Tossing

Deterministic coin tossing was proposed by Cole and Vishkin [9] to break the symmetry of a directed cycle without using randomization. Consider the problem of finding a three-coloring of graph G . A k -coloring of G is a mapping $c : V \rightarrow \{0, 1, \dots, k - 1\}$ such that $c(i) \neq c(j)$ if $\langle i, j \rangle \in E$. Initially, the cycle has a trivial coloring with $n = |V|$ colors. The apparent symmetry in the problem (i.e., the vertices cannot be easily distinguished) presents the major difficulty to reducing the number of colors needed for the coloring. Deterministic coin tossing uses the binary representation of an integer $i = \cdots i_k \cdots i_1 i_0$ to break the symmetry. For example, suppose t is the least significant bit position in which $c(i)$ and $c(S(i))$ differ ($S(i)$ is the successor of i in the cycle), then the new coloring for i can be set as $2t + c(i)_t$.

Accelerating Cascades

Accelerating cascades was presented together with deterministic coin tossing in [9] for the design of faster algorithms for list ranking and other problems. This technique combines two algorithms, one optimal and the other super fast, for the same problem to get a optimal and very fast algorithm. The general strategy is to start with the optimal algorithm to reduce the problem size and then apply the fast but nonoptimal algorithm.

Other Building Blocks for Parallel Graph Algorithms

List ranking determines the position, or rank, of the list items in a linked list. A generalized list-ranking problem is defined as follows. Let X be an array of n elements stored in arbitrary order. For each element i , let $X(i).value$ be its value and $X(i).next$ be the index of its successor. Then, for any binary associative operator \oplus , compute $X(i).prefix$ such that $X(head).prefix = X(head).value$ and $X(i).prefix =$

$X(i).value \oplus X(predecessor).prefix$, where $head$ is the first element of the list, i is not equal to $head$, and $predecessor$ is the node preceding i in the list. Pointer jumping can be applied to solve the list-ranking problem. An optimal list-ranking algorithm is given in [10].

The Euler tour technique is another of the basic building blocks for designing parallel algorithms, especially for tree computations. For example, postorder/preorder numbering, computing the vertex level, computing the number of descendants, etc., can be done work-time optimally on EREW PRAM by applying the Euler tour technique. As suggested by its name, the power of the Euler tour technique comes from defining a Eulerian circuit on the tree. In Tarjan and Vishkin's biconnected components paper [31] that originally introduced the Euler tour technique, the input to their algorithm is an edge list with the cross-pointers between twin edges $\langle u, v \rangle$ and $\langle v, u \rangle$ established. With these cross-pointers it is easy to derive an Eulerian circuit. The Eulerian circuit can be treated as a linked list, and by assigning different values to each edge in the list, list ranking can be used for many tree computation. For example, when rooting a tree, the value 1 is associated with each edge. After list ranking, simply inspecting the list rank for $\langle u, v \rangle$ and $\langle v, u \rangle$ can set the correct parent relationship for u and v .

Tree contraction systematically shrinks a tree into a single vertex by successively shrinking parts of the tree. It can be used to solve the expression evaluation problem. It is also used in other algorithm, for example, computing the biconnected components [31].

Classical Algorithms

The use of the basic techniques are demonstrated in several classical graph algorithms for spanning tree, minimum spanning tree, and biconnected components.

Various deterministic and randomized techniques have been given for solving the spanning tree problem (and the closely related connected components problem) on PRAM models. A brief survey of these algorithms can be found in [3]. The Shiloach–Vishkin algorithm (SV) algorithm is representative of several connectivity algorithms in that it adapts the widely used graft-and-shortcut approach. Through carefully designed grafting schemes, the algorithm achieves complexities of $O(\log n)$ time and $O((m+n)\log n)$

work under the arbitrary CRCW PRAM model. The algorithm takes an edge list as input and starts with n isolated vertices and m processors. Each processor P_i ($1 \leq i \leq m$) inspects edge $e_i = \langle v_{i_1}, v_{i_2} \rangle$ and tries to graft vertex v_{i_1} to v_{i_2} under the constraint that $v_{i_1} < v_{i_2}$. Grafting creates $k \geq 1$ connected components in the graph, and each of the k components is then shortcut to a single super-vertex. Grafting and short-cutting are iteratively applied to the reduced graphs $G' = (V', E')$ (where V' is the set of super-vertices and E' is the set of edges among super-vertices) until only one super-vertex is left.

Minimum spanning tree (MST) is one of the most studied combinatorial problems with practical applications. While several theoretic results are known for solving MST in parallel, many are considered impractical because they are too complicated and have large constant factors hidden in the asymptotic complexity. See for a survey of MST algorithms. Many parallel algorithms are based on the Boruvka algorithm. Boruvka's algorithm is comprised of Boruvka iterations that are used in many parallel MST algorithms. A Boruvka iteration is characterized by three steps: *find-min*, *connected-components*, and *compact-graph*. In *find-min*, for each vertex v the incident edge with the smallest weight is labeled to be in the MST; *connect-components* identifies connected components of the induced graph with the labeled MST edges; *compact-graph* compacts each connected component into a single supervertex, removes self-loops and multiple edges, and relabels the vertices for consistency.

A connected graph is said to be *separable* if there exists a vertex v such that removal of v results in two or more connected components of the graph. Given a connected, undirected graph G , the biconnected components problem finds the maximal-induced subgraphs of G that are not *separable*. Tarjan [30] presents an optimal $O(n + m)$ algorithm that finds the biconnected components of a graph based on depth-first search (DFS). Eckstein [17] gave the first parallel algorithm that takes $O(d \log^2 n)$ time with $O((n + m)/d)$ processors on CREW PRAM, where d is the diameter of the graph. Tarjan and Vishkin [31] present an $O(\log n)$ time algorithm on CRCW PRAM that uses $O(n + m)$ processors. This algorithm utilizes many of the fundamental primitives including prefix sum, list ranking, sorting, connectivity, spanning tree, and tree computations.

Communication Efficient Graph Algorithms

Communication-efficient parallel algorithms were proposed to address the “bottleneck of processor-to-processor communication” (e.g., see [15]). Goodrich [19] presented a communication-efficient sorting algorithm on weak-CREW BSP that runs in $O(\log n / \log(h+1))$ communication rounds (with at most h data transported by each processor in each round) and $O((n \log n)/p)$ local computation time, for $h = \Theta(n/p)$. Goodrich's sorting algorithm is frequently used in communication-efficient graph algorithms. Dehne et al. designed an efficient list-ranking algorithm for coarse-grained multicomputers (CGM) and BSP that takes $O(\log p)$ communication rounds with $O(n/p)$ local computation. In the same study, a series of communication-efficient graph algorithms such as connected components, ear decomposition, and biconnected components are presented using the list-ranking algorithm as a building block. On the BSP model, Adler et al. [1] presented a communication-optimal MST algorithm. The list-ranking algorithm and the MST algorithm take similar approaches to reduce the number of communication rounds. They both start by simulating several (e.g., $O(\log p)$ or $O(\log \log p)$) steps of the PRAM algorithms on the target model to reduce the input size so that it fits in the memory of a single node. A sequential algorithm is then invoked to process the reduced input of size $O(n/p)$, and finally the result is broadcast to all processors for computing the final solution.

Practical Implementation

Many real-world graphs are large and sparse. These instances are especially hard to process due to the characteristics of the workload. Although fast theoretic algorithms exist in the literature, large and sparse graph problems are still challenging to solve in practice. There remains a significant gap between algorithmic model and architecture. The mismatch between memory access pattern and cache organization is the most outstanding barrier to high-performance graph analysis on current systems.

Graph Workload

Compared with traditional scientific applications, graph analysis is more memory intensive. Graph algorithms put tremendous pressure on the memory subsystem

to deliver data to the processor. [Table 1](#) shows the percentages of memory instructions executed in the SPLASH2 benchmarks [34] and several graph algorithms. SPLASH2 represents typical scientific applications for shared-memory environments. On IBM Power5, on average, about 10% more memory instructions are executed in the graph algorithms. For biconnected components, the number is over 59%.

For memory-intensive applications, the locality behavior is especially crucial to the performance. Unfortunately, most graph algorithms exhibit erratic memory access patterns that result in poor performance, as shown in [Table 2](#). [Table 2](#) is the cycles per instruction (CPI) construction [13] for three graph algorithms on IBM Power5. The algorithms studied are betweenness centrality (BC), biconnected components (BiCC), and minimum spanning tree (MST). CPI construction attributes in percentage cycles to categories such as completion, instruction cache miss penalty, and stall. In [Table 2](#), for all algorithms, significant amount of cycles are spent on pipeline stalls. About 60–70% of the cycles are wasted on the load-store unit stalls, and more than 50% of the cycles are spent on stalls due to data cache misses. [Table 2](#) clearly shows that graph algorithms perform poorly on current cache-based architectures, and the culprit is the memory access pattern. The floating-point stalls (FPU STL) column is revealing about one other prominent feature of graph algorithms.

FPU STL is the percentage of cycles wasted on floating-point unit stalls. In [Table 2](#), BiCC and MST do not incur any FPU stalls. Unfortunately, this does not mean that the workload fully utilizes the FPU. Instead, as there are no floating-point operations in these algorithms, the elaborately designed floating point units lay idle. CPI construction shows that graph workloads spent most of the time waiting for data to be delivered, and there are not many other operations to hide the long latency to main memory. In fact, of the three algorithms, only BC incurs execution of a few floating-point instructions.

G

Implementation on Shared-Memory Machines

It is relatively straightforward to map PRAM algorithms on to shared-memory machines such as symmetric multiprocessors (SMPs), multi-core and many core systems. While these systems are of shared-memory architecture, they are by no means the PRAM used in theoretical work – synchronization cannot be taken for granted, memory bandwidth is limited, and performance requires a high degree of locality. Practical design choices need to be made to achieve high performance on such systems.

Adapting to the Available Parallelism

Nick's Class (\mathcal{NC}) is defined as the set of all problems that run in polylog-time with a polynomial number of processors. Whether a problem P is in \mathcal{NC} is a fundamental question. The PRAM model assumes an

Graph Algorithms. Table 1 Percentages of load-store instructions for the SPLASH2 benchmark and the graph problems. ST, MST, BiCC, and CC stand for spanning tree, minimum spanning tree, biconnected components, and connected components, respectively

Benchmark	SPLASH2				Graph problems			
	Barnes	Cholesky	Ocean	Raytrace	ST	MST	BiCC	CC
Load	20.3%	20.6%	21.4%	25.1%	45.6%	28.6%	40.4%	44.5%
Store	15.6%	5.2%	17.8%	9.5%	2%	15.2%	19.2%	1.4%
Load+store	35.9%	25.8%	39.2%	34.6%	47.6%	43.8%	59.6%	45.9%

Graph Algorithms. Table 2 CPI construction for three graph algorithms. Base cycles are for “useful” work. The “Stall” columns show the percentages of cycles on pipeline stalls followed by stalls due to load-store unit, data cache miss, floating-point unit, fix point unit, respectively

Algorithm	Base	GCT	Stall	LSU STL	DCache STL	FPU STL	FXU STL
BC	0.099	0.011	0.888	0.797	0.558	0.030	0.016
BiCC	0.170	0.066	0.762	0.600	0.445	0.000	0.060
MST	0.106	0.037	0.855	0.713	0.600	0.000	0.015

unlimited number of processors and explores the maximum inherent parallelism of P . Acknowledging the practical restriction of limited parallelism provided by real computers, Kruskal et al. [26] argued that non-polylogarithmic time algorithms (e.g., sublinear time algorithms) could be more suitable than polylog algorithms for implementation with practically large input size. \mathcal{EP} (short for *efficient parallel*) algorithms, by the definition in [26], is the class of algorithms that achieve a polynomial reduction in running time with a polylogarithmic inefficiency. With \mathcal{EP} algorithms, the design focus is shifted from reducing the complexity factors to solving problems of realistic sizes efficiently with a limited number of processors.

Reducing Synchronization

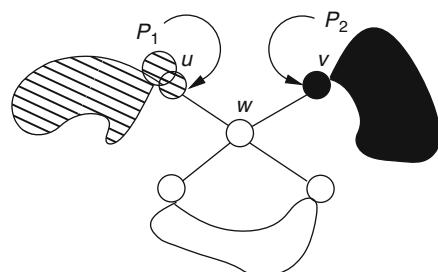
When adapting a PRAM algorithm to shared-memory machines, a thorough understanding of the algorithm usually suffices to eliminate unnecessary synchronization such as barriers. Reducing synchronization thus is an implementation issue. Asynchronous algorithm design, however, is more aggressive in reducing synchronization. It sometimes allows nondeterministic intermediate results but deterministic solutions.

In a parallel environment, to ensure correct final results oftentimes a total ordering on all the events is not necessary, and a partial ordering in general suffices. Relaxed constraints on ordering reduce the number of synchronization primitives in the algorithm.

Bader and Cong presented a largely asynchronous spanning tree algorithm in [3] that employs a constant number of barriers. The spanning tree algorithm for shared-memory machines has two main steps: (1) stub spanning tree and (2) work-stealing graph traversal. In the first step, one processor generates a stub spanning tree, that is, a small portion of the spanning tree by randomly walking the graph for $O(p)$ steps. The vertices of the stub spanning tree are evenly distributed into each processor's queue, and each processor in the next step will traverse from the first element in its queue. After the traversals in step 2, the spanning subtrees are connected to each other by this stub spanning tree. In the graph traversal step, each processor traverses the graph (by coloring the nodes) similar to the sequential algorithm in such a way that each processor finds a subgraph of the final spanning tree. Work-stealing is used to balance the load for graph traversal.

One problem related to synchronization is that there could be portions of the graph traversed by multiple processors and be in different subgraphs of the spanning tree. The immediate remedy is to synchronize using either locks or barriers. With locks, coloring the vertex becomes a critical section, and a processor can only enter the critical section when it gets the lock. Although the nondeterministic behavior is now prevented, it does not perform well on large graphs due to an excessive number of locking and unlocking operations.

In the proposed algorithm, no barriers are introduced in graph traversal. The algorithm runs correctly without barriers even when two or more processors color the same vertex. In this situation, each processor will color the vertex and set as its parent the vertex it has just colored. Only one processor succeeds at setting the vertex's parent to a final value. For example, using Fig. 1, processor P_1 colored vertex u , and processor P_2 colored vertex v , and at a certain time they both find w unvisited and are now in a race to color vertex w . It makes no difference which processor colored w last because w 's parent will be set to either u or v (and it is legal to set w 's parent to either of them; this will not change the validity of the spanning tree, only its shape). Further, this event does not create cycles in the spanning tree under sequential consistency model. Both P_1 and P_2 record that w is connected to each processor's own tree. When each of w 's unvisited children are visited by various processors, its parent will be set to w , independent of w 's parent.



Graph Algorithms. Fig. 1 Two processors P_1 and P_2 see vertex w as unvisited, so each is in a race to color w and set w 's parent pointer. The shaded area represents vertices colored by P_1 , the black area represents those marked by P_2 , and the white area contains unvisited vertices

Choosing Between Barriers and Locks

Locks and barriers are two major types of synchronization primitives. In practice, the choice of using locks or barriers may not be very clear. Take the “graft and shortcut” spanning tree algorithm for example. For graph $G = (V, E)$ represented as an edge list, the algorithm starts with n isolated vertices and $2m$ processors. For edge $e_i = \langle u, v \rangle$, processor P_i ($1 \leq i \leq m$) inspects u and v , and if $v < u$, it grafts vertex u to v and labels e_i to be a spanning tree edge. The problem here is that for a certain vertex v , its multiple incident edges could cause grafting v to different neighbors, and the resulting tree may not be valid. To ensure that v is only grafted to one of the neighbors, locks can be used. Associated with each vertex v is a flag variable protected by a lock that shows whether v has been grafted. In order to graft v a processor has to obtain the lock and check the flag, thus race conditions are prevented. A different solution uses barriers [31] in a two-phase election. No checking is needed when a processor grafts a vertex, but after all processors are done (ensured with barriers), a check is performed to determine which one succeeds and the corresponding edge is labeled as a tree edge. Whether to use a barrier or lock is dependent on the algorithm design as well as the barrier and lock implementations. Locking typically introduces large memory overhead. When contention among processors is intense, the performance degrades significantly.

Cache Friendly Design

The increasing speed difference between processor and main memory makes cache and memory access patterns important factors for performance. The fact that modern processors have multiple levels of memory hierarchy is generally not reflected by most of the parallel models. As a result, few parallel algorithm studies have touched on the cache performance issue. The SMP model proposed by Helman and JáJa is the first effort to model the impact of memory access and cache over an algorithm’s performance [21]. The model forces an algorithm designer to reduce the number of noncontiguous memory accesses. However, it does not give hints to the design of cache-friendly parallel algorithms.

Chiang et al. [7] presented a PRAM simulation technique for designing and analyzing efficient external-memory (sequential) algorithms for graph problems. This technique simulates the PRAM memory by

keeping a task array of $O(N)$ on disk. For each PRAM step, the simulation sorts a copy of the contents of the PRAM memory based on the indices of the processors for which they will be operands, and then scans this copy and performs the computation for each processor being simulated. The following can be easily shown:

Theorem 1 *Let A be a PRAM algorithm that uses N processors and $O(N)$ space and runs in time T . Then A can be simulated in $O(T \cdot \text{sort}(N))$ I/Os [7].*

Here, $\text{sort}(N)$ represents the optimal number of I/Os needed to sort N items striped across the disks, and $\text{scan}(N)$ represents the number of I/Os needed to read N items striped across the disks. Specifically,

$$\text{sort}(x) = \frac{x}{DB} \log_{\frac{M}{B}} \frac{x}{B}$$

$$\text{scan}(x) = \frac{x}{DB}$$

where $M = \#$ of items that can fit into main memory, $B = \#$ of items per disk block, and $D = \#$ of disks in the system.

A similar technique can be applied to the cache-friendly parallel implementation of PRAM algorithms for large inputs. I/O efficient algorithms exhibit good spatial locality behavior that is critical to good cache performance. Instead of having one processor simulate the PRAM step, $p \ll n$ processors may perform the simulation concurrently. The simulated PRAM implementation is expected to incur few cache block transfers between different levels. For small input sizes, it would not be worthwhile to apply this technique as most of the data structures can fit into cache. As the input size increases, the cost to access memory becomes more significant, and applying the technique becomes beneficial.

Algorithmic Optimizations

For most problems, parallel algorithms are inherently more complicated than the sequential counterparts, incurring large overheads with many algorithm steps. Instead of lowering the asymptotic complexities, in many cases, reducing the constant factors improves performance. Cong and Bader demonstrates the benefit of such optimizations with their biconnected components algorithm [11].

The algorithm eliminates edges that are not essential in computing the biconnected components. For any

input graph, edges are first eliminated before the computation of biconnected components is done so that at most $\min(m, 2n)$ edges are considered. Although applying the filtering algorithm does not improve the asymptotic complexity, in practice, the performance of the biconnected components algorithm can be significantly improved.

An edge e is considered as *nonessential* for biconnectivity if removing e does not change the biconnectivity of the component to which it belongs. Filtering out *nonessential* edges when computing biconnected components (these edges are put back in later) yields performance advantages. The Tarjan–Vishkin algorithm (TV) is all about finding the equivalence relation R'_c^* [24, 31]. Of the three conditions for R'_c , it is trivial to check for condition 1 which is for a tree edge and a non-tree edge. Conditions 2 and 3, however, are for two tree edges, and checking involves the computation of *high* and *low* values. To compute *high* and *low*, every non-tree edge of the graph is inspected, which is very time consuming when the graph is not extremely sparse. The fewer edges the graph has, the faster the *Low-high* step. Also, when building the auxiliary graph, the fewer edges in the original graph means the smaller the auxiliary graph and the faster the *Label-edge* and *Connected-components* steps.

Combining the filtering algorithm for eliminating *nonessential* edges and TV, the new biconnected components algorithm runs in $\max(O(d), O(\log n))$ time with $O(n)$ processors on CRCW PRAM, where d is the diameter of the graph. Asymptotically, the new algorithm is not faster than TV. In practice, however, parallel speedups up to 4 with 12 processors are achieved on SUN Enterprise 4500 using the filtering technique.

Implementation on Multithreaded Architectures

Graph algorithms have been observed to run well on multi-threaded architectures such as the CRAY MTA-2 [5] and its successor, the Cray XMT. The Cray MTA [14] is a flat, shared-memory multiprocessor system. All memory is accessible and equidistant from all processors. There is no local memory and no data caches. Parallelism, and not caches, is used to tolerate memory and synchronization latencies.

An MTA processor consists of 128 hardware streams and one instruction pipeline. Each stream can have up to 8 outstanding memory operations. Threads from the

same or different programs are mapped to the streams by the runtime system. A processor switches among its streams every cycle, executing instructions from non-blocked streams in a fair manner. As long as one stream has a ready instruction, the processor remains fully utilized.

Bader et al. compared the performance of list-ranking algorithm on SMPs and MTA [5]. For list ranking, they used two classes of list to test the algorithms: *Ordered* and *Random*. Ordered places each element in the array according to its rank; thus, node i is the i^{th} position of the array and its successor is the node at position $(i + 1)$. Random places successive elements randomly in the array. Since the MTA maps contiguous logical addresses to random physical addresses, the layout in physical memory for both classes is similar, the performance on the MTA is independent of order. This is in sharp contrast to SMP machines which rank Ordered lists much faster than Random lists. On the SMP, there is a factor of 3–4 difference in performance between the best case (an ordered list) and the worst case (a randomly-ordered list). On the ordered lists, the MTA is an order of magnitude faster than the SMP, while on the random list, the MTA is approximately 35 times faster.

Implementation on Distributed Memory Machines

As data partitioning and explicit communication are required, implementing highly irregular algorithms is hard on distributed-memory machines. As a result, although many fast theoretic algorithms exist in the literature, few experimental results are known. As for performance, the adverse impact of irregular accesses is magnified in the distributed-memory environment when memory requests served by remote nodes experience long network latency. Two studies have demonstrated reasonable parallel performance with distributed-memory machines [28, 35]. Both studies implement parallel breadth-first search (BFS), one on BlueGene/L [2] and the other on CELL/BE [22]. The CELL architecture resembles a distributed-memory setting as explicit data transfer is necessary between the local storage on an SPE and the main memory. Neither study establishes a strong evidence for fast execution of parallel graph algorithms on distributed-memory systems. The individual CPU in BlueGene and the PPE

of CELL are weak compared with other Power processors or the SPE. It is hard to establish a meaningful baseline to compare the parallel performance against. Indeed, in both studies, either only wall clock times or speedups compared with other reference architectures are reported.

Partitioned global address space (PGAS) languages such as UPC and X10 [6, 32] have been proposed recently that present a shared-memory abstraction to the programmer for distributed-memory machines. They allow the programmer to control the data layout and work assignment for the processors. Mapping shared-memory graph algorithms onto distributed-memory machines is straightforward with PGAS languages.

Performance wise, straightforward PGAS implementation for irregular graph algorithms does not usually achieve high performance due to the aggregate startup cost of many small messages. Cong, Almasi, and Saraswat presented their study in optimizing the UPC implementation of graph algorithm in [12]. They improve both the communication efficiency and the cache performance of the algorithm through improving the locality behavior.

Some Experimental Results

Greiner [20] implemented several connected components algorithms using NESL on the Cray Y-MP/C90 and TMC CM-2. Hsu, Ramachandran, and Dean [23] also implemented several parallel algorithms for connected components. They report that their parallel code runs 30 times slower on a MasPar MP-1 than Greiner's results on the Cray, but Hsu et al.'s implementation uses one-fourth of the total memory used by Greiner's approach. Krishnamurthy et al. [25] implemented a connected components algorithm (based on Shiloach-Vishkin [29]) for distributed memory machines. Their code achieved a speedup of 20 using a 32-processor TMC CM-5 on graphs with underlying 2D and 3D regular mesh topologies, but virtually no speedup on sparse random graphs. Goddard, Kumar, and Prins [18] implemented a connected components algorithm for a mesh-connected SIMD parallel computer, the 8192-processor MasPar MP-1. They achieve a maximum parallel speedup of less than two on a random graph with 4,096 vertices and about one-million edges. For a

random graph with 4,096 vertices and fewer than a half-million edges, the parallel implementation was slower than the sequential code.

Chung and Condon [8] implemented a parallel minimum spanning tree (MST) algorithm based on Boruvka's algorithm. On a 16-processor CM-5, for geometric graphs with 32,000 vertices and average degree 9 and graphs with fewer vertices but higher average degree, their code achieved a parallel speedup of about 4, on 16-processors, over the sequential Boruvka's algorithm, which was 2–3 times slower than their sequential Kruskal algorithm.

Dehne and Götz [16] studied practical parallel algorithms for MST using the BSP model. They implemented a dense Boruvka parallel algorithm, on a 16-processor Parsytec CC-48, that works well for sufficiently dense input graphs. Using a fixed-sized input graph with 1,000 vertices and 400,000 edges, their code achieved a maximum speedup of 6.1 using 16 processors for a random dense graph. Their algorithm is not suitable for sparse graphs.

Woo and Sahni [33] presented an experimental study of computing biconnected components on a hypercube. Their test cases are graphs that retain 70 and 90% edges of the complete graphs, and they achieved parallel efficiencies up to 0.7 for these dense inputs. The implementation uses adjacency matrix as input representation, and the size of the input graphs is limited to less than 2K vertices.

Bader and Cong presented their studies [3, 4, 11] of the spanning tree, minimum spanning tree, and biconnected components algorithms on SMPs. They achieved reasonable parallel speedups on the large, sparse inputs compared with the best sequential implementations.

Bibliography

1. Adler M, Dittrich W, Juurlink B, Kutyłowski M, Rieping I (1998) Communication-optimal parallel minimum spanning tree algorithms (extended abstract). In: SPAA'98: proceedings of the tenth annual ACM symposium on parallel algorithms and architectures. ACM, New York, pp 27–36
2. Allen F, Almasi G et al (2001) Blue Gene: a vision for protein science using a petaflop supercomputer. IBM Syst J 40(21):310–327
3. Bader DA, Cong G (2004) A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). In: Proceedings of the 18th international parallel and distributed processing symposium (IPDPS 2004), Santa Fe

4. Bader DA, Cong G (2004) Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. In: Proceedings of the 18th international parallel and distributed processing symposium (IPDPS 2004), Santa Fe
5. Bader DA, Cong G, Feo J (2005) On the architectural requirements for efficient execution of graph algorithms. In: Proceeding of the 2005 international conference on parallel processing, Oslo, pp 547–556
6. Charles P, Donawa C, Ebcioiglu K, Grothoff C, Kielstra A, Praun CV, Saraswat V, Sarkar V (2005) X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of the 2005 ACM SIGPLAN conference on object-oriented programming systems, languages and applications (OOPSLA), San Diego, pp 519–538
7. Chiang Y-J, Goodrich MT, Grove EF, Tamassia R, Vengroff DE, Vitter JS (1995) External-memory graph algorithms. In: Proceedings of the 1995 symposium on discrete algorithms, San Francisco, pp 139–149
8. Chung S, Condon A (1996) Parallel implementation of Boruvka's minimum spanning tree algorithm. In: Proceedings of the tenth international parallel processing symposium (IPPS'96), Honolulu, pp 302–315
9. Cole R, Vishkin U (1986) Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In: STOC'86: proceedings of the eighteenth annual ACM symposium on theory of computing. ACM, New York, pp 206–219
10. Cole R, Vishkin U (1991) Approximate parallel scheduling, part II: applications to logarithmic-time optimal graph algorithms. *Info Comput* 92:1–47
11. Cong G, Bader DA (2005) An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (SMPs). In: Proceedings of the 19th international parallel and distributed processing symposium (IPDPS 2005), Denver
12. Cong G, Almasi G, Saraswat V (2010) Fast PGAS implementation of distributed graph algorithms. In: Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis (SC2010), SC'10. IEEE Computer Society, Washington, DC, pp 1–11
13. CPI analysis on Power5. On line, 2006. <http://www.ibm.com/developerworks/linux/library/pacpipowerl/index.html>
14. Cray, Inc. (2005) The CRAY MTA-2 system. www.cray.com/products/programs/mta_2/
15. Culler DE, Dusseau AC, Martin RP, Schausser KE (1993) Fast parallel sorting under LogP: from theory to practice. In: Portability and performance for parallel processing. Wiley, New York, pp 71–98 (Chap 4)
16. Dehne F, Götz S (1998) Practical parallel algorithms for minimum spanning trees. In: Workshop on advances in parallel and distributed systems, West Lafayette, pp 366–371
17. Eckstein DM (1979) BFS and biconnectivity. Technical Report 79-11, Department of Computer Science, Iowa State University of Science and Technology, Ames
18. Goddard S, Kumar S, Prins JF (1997) Connected components algorithms for mesh-connected parallel computers. In: Bhatt SN (ed) Parallel algorithms: third DIMACS implementation challenge, 17–19 October 1994. DIMACS series in discrete mathematics and theoretical computer science, vol 30. American Mathematical Society, Providence, pp 43–58
19. Goodrich MT (1996) Communication-efficient parallel sorting. In STOC'96: proceedings of the twenty-eighth annual ACM symposium on theory of computing. ACM, New York, pp 247–256
20. Greiner J (1994) A comparison of data-parallel algorithms for connected components. In: Proceedings of the sixth annual symposium on parallel algorithms and architectures (SPAA-94), Cape, May, pp 16–25
21. Helman DR, JáJá J (1999) Designing practical efficient algorithms for symmetric multiprocessors. In: Algorithm engineering and experimentation (ALENEX'99). Lecture notes in computer science, vol 1619. Springer, Baltimore, pp 37–56
22. Hofstee HP (2005) Power efficient processor architecture and the cell processor. In: International symposium on high-performance computer architecture, San Francisco, pp 258–262
23. Hsu T-S, Ramachandran V, Dean N (1997) Parallel implementation of algorithms for finding connected components in graphs. In: Bhatt SN (ed) Parallel algorithms: third DIMACS implementation challenge, 17–19 October 1994. DIMACS series in discrete mathematics and theoretical computer science, vol 30. American Mathematical Society, Providence, pp 23–41
24. JáJá J (1992) An introduction to parallel algorithms. Addison-Wesley, New York
25. Krishnamurthy A, Lumetta SS, Culler DE, Yelick K (1997) Connected components on distributed memory machines. In: Bhatt SN (ed) Parallel algorithms: third DIMACS implementation challenge, 17–19 October 1994. DIMACS series in discrete mathematics and theoretical computer science, vol 30. American Mathematical Society, Providence, pp 1–21
26. Kruskal CP, Rudolph L, Snir M (1990) Efficient parallel algorithms for graph problems. *Algorithmica* 5(1):43–64
27. Paul WJ, Vishkin U (1983) Parallel dictionaries in 2–3 trees. In: Tenth colloquium on automata, languages and programming (ICALP), Barcelona. Lecture notes in computer science. Springer, Berlin, pp 597–609
28. Scarpazza DP, Villa O, Petrini F (2008) Efficient breadth-first search on the Cell/BE processor. *IEEE Trans Parallel Distr Syst* 19(10):1381–1395
29. Shiloach Y, Vishkin U (1982) An $O(\log n)$ parallel connectivity algorithm. *J Algorithms* 3(1):57–67
30. Tarjan RE (1972) Depth-first search and linear graph algorithms. *SIAM J Comput* 1(2):146–160
31. Tarjan RE, Vishkin U (1985) An efficient parallel biconnectivity algorithm. *SIAM J Comput* 14(4):862–874
32. Unified Parallel C, URL: http://en.wikipedia.org/wiki/Unified_Parallel_C
33. Woo J, Sahni S (1991) Load balancing on a hypercube. In: Proceedings of the fifth international parallel processing symposium, Anaheim. IEEE Computer Society, Los Alamitos, pp 525–530
34. Woo SC, Ohara M, Torrie E, Singh JP, Gupta A (1995) The SPLASH-2 programs: characterization and methodological

- considerations. In: Proceedings of the 22nd annual international symposium computer architecture, pp 24–36
35. Yoo A, Chow E, Henderson K, McLendon W, Hendrickson B, Çatalyürek ÜV (2005) A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In: Proceedings of supercomputing (SC 2005), Seattle

Graph Analysis Software

- [SNAP \(Small-World Network Analysis and Partitioning\) Framework](#)

Graph Partitioning

BRUCE HENDRICKSON

Sandia National Laboratories, Albuquerque, NM, USA

Definition

Graph partitioning is a technique for dividing work amongst processors to make effective use of a parallel computer.

Discussion

When considering the data dependencies in a parallel application, it is very convenient to use concepts from graph theory. A *graph* consists of a set of entities called *vertices*, and a set of pairs of entities called *edges*. The entities of interest in parallel computing are small units of computation that will be performed on a single processor. They might be the work performed to update the state of a single atom in a molecular dynamics simulation, or the work required to compute the contribution of a single row of a matrix to a matrix-vector multiplication. Each such work unit will be a vertex in the graph which describes the computation. If two units have a data dependence between them (i.e., the output of one computation is required as input to the other), then there will be an edge in the graph that joins the two corresponding vertices.

For a computation to perform efficiently on a parallel machine each of the P processors needs to have about the same amount of work to perform, and the amount of inter-processor communication must be small. These two conditions can be viewed in terms of the computational graph. The vertices of the graph (signifying

units of work) need to be divided into P sets with about the same number of vertices in each. Additionally, the number of edges that connect vertices in two different sets needs to be kept small since these will reflect the need for interprocessor communication. This problem is known as *graph partitioning* and is an important approach to the parallelization of many applications.

More generally, the vertices of the graph can have weights associated with them, reflecting different amounts of computation, and the edges can also have weights corresponding to different quantities of communication. The graph partitioning problem involves dividing the set of vertices into P sets with about the same amount of total vertex weight, while keeping small the total weight of edges that cross between partitions. This problem is known to be NP-hard, but a number of heuristics have been devised that have proven to be effective for many parallel computing applications. Several software tools have been developed for this problem, and they are an important piece of the parallel computing ecosystem. Important algorithms and tools are discussed below.

Parallel Computing Applications of Graph Partitioning

Graph partitioning is a useful technique for parallelizing many scientific applications. It is appropriate when the calculation consists of a series of steps in which the computational structure and data dependencies do not vary much. Under such circumstances the expense of partitioning is rewarded by improved parallel performance for many computational steps.

The partitioning model is most applicable for bulk synchronous parallel applications in which each step consists of local computation followed by a global data exchange. Fortunately, many if not most scientific applications exhibit this basic structure. Particle simulations are one such important class of applications. The particles could be atoms in a material science or biological simulation, stars in a simulation of galaxy formation, or units of charge in an electromagnetic application.

But by far the most common uses of graph partitioning involve computational meshes for the solution of differential equations. Finite volume, finite difference, and finite element methods all involve the decomposition of a complex geometry into simple shapes that interact only with near neighbors. Various graphs can be

constructed from the mesh and a partition of the graph identifies subregions to be assigned to processors. The numerical methods associated with such approaches are often very amenable to the graph partitioning approach. These ideas have been used to solve problems from many areas of computational science including fluid flow, structural mechanics, electromagnetics, and many more.

Graph Partitioning Algorithms for Parallel Computing

A wide variety of graph partitioning algorithms have been proposed for parallel computing applications. Here we review some of the more important approaches.

Geometric partitioning algorithms are very fast techniques for partitioning sets of entities that have an underlying geometry. For parallel computing applications involving simulations of physical phenomena in two or three dimensions, the corresponding data structures typically have geometric coordinates associated with each entity. Examples include molecules in atomistic simulations, masses in gravitational models, or mesh points in a finite element method. Recursive coordinate partitioning is a method in which the elements are recursively divided by planar cuts that are orthogonal to one of the axes [1]. This has the advantage of producing geometrically simple subdomains – just rectangular parallelepipeds. Recursive inertial bisection also uses planar cuts, but instead of being orthogonal to an axis, they are orthogonal to the direction of greatest inertia [9]. Intuitively, this is a direction in which the point set is elongated, so cutting perpendicular to this direction is likely to produce a smaller cut. Yet another alternative is to cut with circles or spheres instead of planes [8]. Geometric methods tend to be very fast, but produce low-quality partitions. They can be improved via local refinement methods like the approach of Fiduccia-Mattheyses discussed below.

A quite different set of approaches uses eigenvectors of a matrix associated with the graph. The most popular method in this class uses the second-smallest eigenvector of the Laplacian matrix of the graph [9]. A justification for this approach is beyond the scope of this article, but spectral methods generally produce partitions of fairly high quality. In a global sense, they find attractive regions for cutting a graph, but they are often poor

in the fine details. This can be rectified by the application of a local refinement method. The main drawback of spectral methods is their high computational cost.

Local refinement methods are epitomized by the approach proposed by Fiduccia and Mattheyses [5] (FM). This method works by iteratively moving vertices between partitions in a manner that maximally reduces the size of the set of cut edges. Moves are considered even if they make the cut size larger since they may enable subsequent moves that lead to even better partitions. Thus, this method has a limited ability to escape from local minima to search for even better solutions. The key advance underlying FM is the use of clever data structures that allow all the moves and their consequences to be explored and updated efficiently. The FM algorithm is quite fast, and consistently improves results generated by other approaches. But since it only explores sets of partitions that are not far from the initial one, it is generally limited to making small changes and will not find better partitions that are quite different.

The most widely used class of graph partitioning techniques are multilevel algorithms as they provide a good balance between speed and quality. They were independently invented by several research groups more or less simultaneously [2, 4, 7]. Multilevel algorithms work by applying a local refinement method like FM at multiple scales. This largely overcomes the myopia that limits the effectiveness of local methods. This is accomplished by constructing a series of smaller and smaller graphs that roughly approximate the original graph. The most common way to do this is to merge small clusters of vertices within the original graph (e.g., combine two vertices sharing an edge into a single vertex). Once this series of graphs is constructed, the smallest graph is partitioned using any global method. Then the partition is refined locally and extended to the next larger graph. The refinement/extension process is repeated on larger and larger graphs until a partitioning of the original graph has been produced.

A number of general purpose global optimization approaches have been proposed for graph partitioning including simulated annealing, genetic algorithms, and tabu search. These methods can produce high-quality partitions but are usually very expensive and so are limited to niche applications within parallel computing.

Graph partitioning is often used as a preprocessing step to set up a parallel computation. The output of a

graph partitioner determines which objects are assigned to which processors, and appropriate input files and data structures are prepared for a parallel run. However, there are several situations in which the partitioning must be done in parallel. For a very large problem, the memory of a serial machine may be insufficient to hold the graph that needs to be partitioned. Also, for some classes of applications the structure of the computation changes over time. Examples include adaptive mesh simulations, or particle methods in which the particles move significantly. For such problems the work load must be periodically redistributed across the processors, and a parallel partitioning tool is required. Simple geometric algorithms have the advantage of being easy to parallelize, but multilevel partitioners have also been parallelized to provide higher-quality solutions. Techniques for effectively parallelizing such methods is an ongoing area of research.

A variety of open source graph partitioning tools have been developed in serial or parallel including Chaco, METIS, Jostle, and SCOTCH. Several of these are discussed in companion articles.

Limitations of Graph Partitioning

Although widely used to enable the parallelization of scientific applications, graph partitioning is an imperfect abstraction. For a parallel application to perform well, the work must be evenly distributed among processors and the cost of interprocessor communication must be minimized. Graph partition provides only a crude approximation for achieving these objectives.

In the graph partitioning model, each vertex is assigned a weight that is supposed to represent the time required to perform a piece of computation. On modern processors with complex memory hierarchies it is very difficult to accurately predict the runtime of a piece of code *a priori*. Cache performance can dominate runtime, and this is very hard to predict in advance. So the weights assigned to vertices in the graph partitioning model are just rough approximations.

An even more significant shortcoming of graph partitioning has to do with communication. For most applications, a vertex has data that needs to be known by all of its neighbors. If two of those neighbors are owned by the same processor, then that data need only be communicated once. In the graph partitioning model, two

edges would be cut and so the actual volume of communication would be over-counted. Several alternatives to standard graph partitioning have been proposed to address this problem. In one approach, the number of vertices with off-processor neighbors is counted instead of the number of edges cut. A more powerful and elegant alternative uses hypergraphs and is sketched below.

Yet another deficiency in graph partitioning is that it emphasizes the total volume of communication. In many practical situations, latency is the performance-limiting factor, so it is the number of messages that matters most, not the size of messages.

As discussed above, graph partitioning is most appropriate for bulk synchronous applications. If the calculation involves complex interleaving of computations with communication or partial synchronizations then graph partitioning is less useful. An important application with this character is the factorization of a sparse matrix.

Finally, graph partitioning is only appropriate for applications in which the work and communication pattern are predictable and stable. This happens to be the case for many important scientific computing kernels, but there are other applications that do not fit this model.

Hypergraph Partitioning

A hypergraph is a generalization of a graph. Whereas a graph edge connects exactly two vertices, a *hyperedge* can connect any subset of vertices. This seemingly simple generalization leads to improved and more general partitioning models for parallel computing [3].

Consider a graph in which vertices represent computation and edges represent data dependencies. For each vertex, replace all the edges connected to it with a single hyperedge that joins the vertex and all of its neighbors. When the vertices are partitioned, if a particular vertex is separated from any of its neighbors, the corresponding hyperedge will be cut. For the common situation in which the vertex needs to communicate the same information to all of its neighbors, this single hyperedge will reflect the amount of data that needs to be shared with another processor. Thus, the number of cut hyperedges (or more generally the total weight of cut hyperedges) correctly captures the total volume of communication induced by a partitioning. In this way, the

hypergraph model resolves an important shortcoming of standard graph partitioning.

Hypergraphs also address a second deficiency of the graph model. If the communication is not symmetric (e.g., vertex i needs to send data to j , but j does not need to send data to i), then the graph model has difficulty capturing the communication requirements. The hypergraph model does not have this problem. A hyper-edge simply spans a vertex i and every other vertex that i needs to send data to. There is no implicit assumption of symmetry in the construction of the hypergraph model.

Graph and hypergraph partitioning models, algorithms, and software continue to be active areas of research in parallel computing. PaToH and hMETIS are widely used hypergraph partitioning tools for parallel computing.

Related Entries

- ▶ [Chaco](#)
- ▶ [Load Balancing, Distributed Memory](#)
- ▶ [Hypergraph Partitioning](#)
- ▶ [METIS and ParMETIS](#)

Bibliographic Notes and Further Reading

Graph partitioning is a well-studied problem in theoretical computer science and is known to be difficult to solve optimally. For parallel computing, the challenge is to find algorithms that are effective in practice. Algebraic methods like Laplacian partitioning [9] are an important class of techniques, but can be expensive. Local refinement techniques like FM are also important [5], but get caught in local optima. Multilevel methods seem to offer the best trade off between cost and performance [2, 4, 7].

Hypergraph partitioning provides an important alternative to graph partition in many instances [3]. A survey of different partitioning models can be found in the paper by Hendrickson and Kolda [6].

Several good codes for graph partitioning are available on the Internet including Chaco, METIS, PaToH, and Scotch.

Acknowledgment

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the

US Department of Energy under contract DE-AC04-94AL85000.

Bibliography

1. Berger MJ, Bokhari SH (1987) A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans Comput C-36(5)*:570–580
2. Bui T, Jones C (1993) A heuristic for reducing fill in sparse matrix factorization. In: Proceedings of the 6th SIAM Conference on parallel processing for scientific computing, SIAM, Portsmouth, Virginia, pp 445–452
3. Çatalyürek U, Aykanat C (1996) Decomposing irregularly sparse matrices for parallel matrix-vector multiplication. In: Lecture notes in computer science 1117, Proceedings Irregular'96, Springer-Verlag, Heidelberg, pp 75–86
4. Cong J, Smith ML (1993) A parallel bottom-up clustering algorithm with application to circuit partitioning in VLSI design. In: Proceedings of the 30th Annual CAM/IEEE International Design Automation Conference, DAC'93, ACM, San Diego, CA, pp 755–760
5. Fiduccia CM, Mattheyses RM (1982) A linear time heuristic for improving network partitions. In: Proceedings of the 19th ACM/IEEE Design Automation Conference, ACM/IEEE, Las Vegas, NV, June 1982, pp 175–181
6. Hendrickson B, Kolda T (2000) Graph partitioning models for parallel computing. *Parallel Comput* 26:1519–1534
7. Hendrickson B, Leland R (1995) A multilevel algorithm for partitioning graphs. In: Proceedings of Supercomputing '95, ACM, New York, December 1995. Previous version published as Sandia Technical Report SAND93-1301, Albuquerque, NM
8. Miller GL, Teng SH, Vavasis SA (1991) A unified geometric approach to graph separators. In: Proceedings of the 32nd Symposium on Foundations of Computer Science, IEEE, Pittsburgh, PA, October 1991, pp 538–547
9. Simon HD (1991) Partitioning of unstructured problems for parallel processing. In: Proceedings of the conference on parallel methods on large scale structural analysis and physics applications. Pergamon Press, Elmsford, NY

Graph Partitioning Software

- ▶ [Chaco](#)
- ▶ [METIS and ParMETIS](#)
- ▶ [PaToH \(Partitioning Tool for Hypergraphs\)](#)

Graphics Processing Unit

- ▶ [NVIDIA GPU](#)

Green Flash: Climate Machine (LBNL)

JOHN SHALF¹, DAVID DONOFRIO¹, CHRIS ROWEN²,
LEONID OLICKER¹, MICHAEL WEHNER¹

¹Lawrence Berkeley National Laboratory, Berkeley,
CA, USA

²CEO, Tensilica, Santa Clara, CA, USA

Synonyms

LBNL climate computer; Manycore; Tensilica; [View from Berkeley](#)

Definition

Green Flash is a research project focused on an application-driven manycore chip design that leverages commodity-embedded circuit designs and hardware/software codesign processes to create a highly programmable and energy-efficient HPC design. The project demonstrates how a multidisciplinary hardware/software codesign process that facilitates close interactions between applications scientists, computer scientists, and hardware engineers can be used to develop a system tailored for the requirements of scientific computing. By leveraging the efficiency gained from application-driven design philosophy, advanced processor synthesis tools from Tensilica, FPGA-accelerated architectural simulation from RAMP, auto-tuning for rapid optimization of the software implementation, the project demonstrated how a hardware/software codesign process can achieve a 100× increase in energy efficiency over its contemporaries using cost-effective commodity-embedded building blocks. To demonstrate application-driven design process, *Green Flash* was tailored for high-resolution global cloud resolving models, which are the leading justification for exascale computing systems. However, the approach can be generalized to a broader array of scientific applications. As such, *Green Flash* represents a vision of a new design process that could be used to develop effective exascale-class HPC systems.

Discussion

Introduction

The scientific community is facing one of its greatest challenges in the prediction of global climate change –

a question whose answer has staggering economic, political, and sociological ramifications. The computational power required to inform such critical policy decisions requires a new breed of extreme scale computers to accurately model the global climate. The “business as usual” approach of using commercial off-the-shelf (COTS) hardware to build ever-larger clusters is increasingly unsustainable beyond the petaflop scale due to the constraints of power and cooling. Some estimates indicate an exaflop-capable machine would consume close to 180 MW of power. Such unreasonable power costs drive the need for a radically new approach to HPC system design. *Green Flash* is a theoretical system designed with an application-driven hardware and software codesign for HPC systems that leverages the innovative and low-power architectures and design processes of the low-power/embedded computing industry. *Green Flash* is the result of Berkeley Lab’s research into energy-efficient system design – many details that are common to all system design, such as power, cooling, mechanical design, etc. are not addressed in this research as they are not unique to *Green Flash* and are challenges that would need to be overcome regardless of system architecture. The work presented here represents the energy efficiency gained through application-tailored architectures that leverage embedded processors to build energy efficient manycore processors.

History

In 2004, a group of University of California researchers, with backgrounds ranging from circuit design, computer architecture, CAD, embedded hardware/software, programming languages, compilers, applied math, to HPC, met for a period of two years to consider how current constraints on device physics at the silicon level would affect CPU design, system architecture, and programming models for future systems. The results of the discussions are documented in the University of California Berkeley Technical Report entitled “The Landscape of Parallel Computing Research: A View from Berkeley.” This report was the genesis of the UC Berkeley ParLab, which was funded by Intel and Microsoft as well as the *Green Flash* project. Whereas the ParLab carried the work of the View from Berkeley forward for desktop and handheld applications,

the Green Flash project took the same principles and applied them to the design of energy-efficient scientific computing systems.

Hardware/software codesign, a methodology that allows both software optimization and semi-specialized processor design to be simultaneously developed, has long been a feature of power-sensitive embedded system designs, but thus far has seen very little application in the HPC space. However, given power has become the leading design constraint of future HPC systems and codesign, and other application-driven design processes have received considerably more attention. Green Flash leverages tools that were developed by Tensilica for rapid-synthesis of application-optimized CPU designs, and retargets them to designing processors that are optimized for scientific applications. The project also created novel inter-processor communication to enable that easier-to-program environment than its GPU contemporaries – providing hardware support for more natural programming environments based on partitioned global address space programming models.

Approach

It is widely agreed that architectural specialization can significantly improve efficiency, however, creating full-custom designs of HPC systems has often proven impractical due to excessive design/verification costs and lead-times. The embedded processor market relies on architectural customization to meet the demanding cost and power efficiency requirements of its products with a short turn-around time. With time to market a key element in profitability, sophisticated toolchains have been developed to enable rapid and cost-effective turn-around of power-efficient semicustom designs implementations appropriate to each specific processor design. Green Flash leverages these same toolchains to design power-efficient exascale systems, tailoring embedded chips to target scientific applications and providing a feedback path from the application programmer to the hardware design enabling a tight hardware/software codesign loop that is unprecedented in the HPC industry. *Auto-tuning* technologies are used to automate the software tuning process and maintain portability across the differing architectures produced inside the codesign loop. Auto-tuners can automatically search over a broad parameter space

of optimizations to improve the computational efficiency of application kernels and help produce a more balanced architecture. To enable fast, accurate performance evaluation a Field Programmable Gate Array (FPGA) based hardware emulation platform will be used to allow an experimental architecture to be evaluated at speeds 1,000× faster than typical software-based simulation methods – fast enough to allow execution of a real application rather than an arbitrary benchmark.

High-resolution global cloud system resolving models are the target application that will motivate Green Flash's architectural decisions. A truly exascale problem, a 1.5 km scale model would decompose the earth's atmosphere into twenty-billion individual cells and a machine with unprecedented performance would need to be realized in order for the model to run faster than real time. While using more power-efficient, off-the-shelf, embedded processors is a crucial first in meeting this challenge it is still insufficient. Green Flash will offer many other novel optimizations, both hardware and software, including alternatives to cache coherence that enable far more efficient inter-processor communication than a conventional symmetric multiprocessing (SMP) approach and aggressive, architecture-specific software optimization through auto-tuning. All these specialization techniques will allow Green Flash to efficiently meet the exascale computation requirements of global climate change prediction.

Modeling the Earth's Climate System

Current generation climate models are comprehensive representations of the various systems that determine the Earth's climate. Models prepared for the fourth report of the Intergovernmental Panel on Climate Change coupled submodels of the atmosphere, ocean, and sea ice together to provide simulations of the past, present, and future climate. It is expected that the major remaining components of the climate system, the terrestrial and oceanic biosphere, the Greenland and Antarctic ice sheets, and certain aspects of atmospheric chemistry will be represented in models currently being prepared for the next report. Each of the subsystem models has their own strengths and weaknesses, and each introduces a certain amount of uncertainty into projections of the future. Current computational resources limit the resolution of these submodels and are a contributor to these uncertainties. In

particular, resolution constraints on models of atmospheric processes do not allow clouds to be resolved forcing model developers to rely on sub-grid scale parameterizations based on statistical methods. However, simulations of the recent past produce cloud distributions that do not agree well with observations. These disagreements, traceable to the cumulus convection parameterizations, lead to other errors in patterns of the Earth's radiation and moisture budgets. Current global atmospheric models have resolutions of order 200 km, obviously many times larger than individual clouds. Development of models at the limit of the validity of cumulus parameterization (~ 25 km) is now underway by a few groups, although the necessary century scale integrations are just barely feasible on the largest current computing platforms. It is expected that many issues will be rectified by this increase in horizontal fidelity but that the fundamental limitations of cumulus parameterization will remain. The solution to this problem is to directly simulate cloud processes rather than attempt to model them statistically. At horizontal grid spacing of order ~ 1 km, cloud systems can be individually resolved providing this direct numerical simulation. However, the computation burden of fluid dynamics algorithms scales nonlinearly with the number of grid points due to time step limitations imposed by numerical stability requirements. Hence, the computational resources necessary to carry out century scale simulations of the Earth's climate dwarfs any traditional machine currently under development.

Climate Model Requirements

Extrapolation from measured computational requirements of existing atmospheric models allow estimates of what would be necessary at resolutions of order 1 km to support Global Cloud Resolving Models. To better make these estimates, the Green Flash project has partnered with Prof. David Randall's group at the Colorado State University (CSU). In their approach, the globe is represented by a mesh based on an icosahedron as the starting point. By successively bisecting the sides of the triangles making up this object, a remarkably uniform mesh on the sphere can be generated. However, this is not the only way to discretize the globe at this resolution and it will be important to have a variety of independent cloud system-resolving models if projections of the future are to have any credibility. For this reason it

is important to emphasize that Green Flash will not be built to only run this particular discretization. Rather, this approach calls for optimizing a system for a class of scientific applications; therefore, Green Flash will be able to efficiently run most global climate models.

Extrapolation based on today's cluster-based, general purpose, HPC systems produce estimates that the *sustained* computational rate necessary to simulate the Earth's climate 1,000 times faster than it actually occurs was 10 Pflops. A tentative estimate from the CSU model is as much as 70 Pflops. This difference can be regarded as one measure of the considerable uncertainty in making these estimates. As the CSU model matures, there will be the opportunity to determine this rate much more accurately. Multiple realizations of individual simulations are necessary to address the statistical complexities of climate system. Hence, an exaflop scale machine would be necessary to carry out this kind of science. The exact peak flop rate required depends greatly on the efficiency that the machine could be used.

These enormous sustained computational rates are not even imaginable if there is not enough parallelism in the climate problem. Fortunately, cloud system resolving models at the kilometer scale do offer plenty of opportunity to decompose the physical domain. Bisection of the triangles composing the icosahedron twelve successive times produces a global mesh with 167,772,162 vertices spaced between 1 and 2 km apart. A logically rectangular two-dimensional domain decomposition strategy can be applied horizontally to the icosahedral grid. Choosing square segments of the mesh containing 64 grid points each (8×8) results in 2,621,440 horizontal domains. The vertical dimension offers additional parallelism. Assuming that 128 layers could be decomposed in 8 separate vertical domains, the total number of physical sub-domains could be 20,971,520.

Twenty-one million way parallelism may seem mind-boggling but this particular strawman decomposition was devised with practical constraints on the performance of an individual core in an SMP in mind. Each of the 21-million cores in this system will be assigned a small group of sub-domains on which they will execute the full (physics, dynamics, etc.) climate model. Flops per watt is the key performance metric for designing the SMP for Green Flash, and the goal of $100\times$ energy efficiency over existing machines will be achieved by

tailoring the architecture to the needs of the climate model. One example that drives the need for a high core count per socket is the model's communication pattern. While the model is nearest-neighbor dominated, the majority of the more latency-sensitive communication occurs between the vertical layers. By keeping this communication on-chip the more latency tolerant horizontal communication can be sent off-chip with less performance penalty. Looking at per-core features, by running with a lower (500 MHz) clock speed relative to today's server-class processors Green Flash gains a cubic improvement in power consumption. Removal of processor features that are nonoptimal for science such as Translation Look-aside Buffers (TLBs) and Out of Order (OOO) processing creates a smaller die, which reduces leakage to help further reduce power.

Hardware Design Flow

Verification costs are quickly becoming the dominant force in custom hardware solutions. Large arrays of simple processors again hold a significant advantage here, as the work to verify to a simple processing element and then replicate them on die is significantly lower. The long lead times and high costs of doing custom designs has generally dissuaded the HPC community from custom solutions and pushed more for clusters of COTS (Commercial off-the-self) hardware. The traditional definition of COTS in the HPC space is typically at the board or socket level; Green Flash seeks to redefine this notion of COTS and asserts that a custom processor made up of pre-verified building blocks can still be considered COTS hardware. This fine grained view permits Green Flash to benefit from both the architectural specialization afforded by these specialized processing elements and the shorter lead times and reduced verification costs that come with using a building block approach.

The constraints of power have long directed the development of embedded architectures and so it is advantageous to begin with an embedded core and leverage the sophisticated tool chains developed to minimize time from architectural specifications to ASIC. These toolchains start with a collection of pre-verified functional units and allow them to be combined in a myriad of ways rapidly producing power-efficient semi-custom designs. For instance, starting with a base architecture a designer may wish to add floating-point

support to a processor, or perhaps add a larger cache or local store. These functional units can be added to a processor design as easily as clicking a checkbox or dropdown menu. The tool will then select the correct functional unit from its library and integrate it into the design – all without the designer needing to intervene. These tools eliminate large amounts of not only boilerplate, but also full custom logic that once needed to be written and re-written in order to change a processor's architecture. Of course the tools are not boundless and are subject to the same design limitations as any other physical design process – for instance, one cannot efficiently have hundreds of read ports from a single memory, but the amount of flexibility created through these tools vastly outweighs any inherent limitations.

The rapid generation of processor cores alone makes these tools very interesting, however, the overhead of generating a usable software stack for each processor would negate the time saved developing the hardware. While adding caches or changing bus widths has little effect on the ISA, and therefore a minimal software impact, adding a new functional unit such as floating-point or integer division has a large impact on the software flow. Building custom hardware creates significant work not only in the creation of a potentially complex software stack but also a time-consuming verification process. As with the software stack, without a method to jump-start the verification process the tools would begin to lose their effectiveness. To address both of these critical issues these tools generate optimizing compilers, test benches as well as a functional simulator in parallel with the RTL for the design. Having the processor constructed of pre-verified building blocks combined with the automatic generation of test benches greatly reduces the risk and time required for formal verification. To help maintain backward and general purpose compatibility the processor's ISA is restricted to one that is functionally complete and allows for the execution of general purpose code.

A Science Optimized Processor Design

The processor design for Green Flash is driven by efficiency and the best way to reduce power consumption and increase efficiency is to reduce waste. With that in mind, the target architecture calls for a very simple, in order core with no branch prediction. The heavy memory and communication requirements demanded by the

climate model have imparted the greatest influence on the design of the Green Flash core. Building on prior work from Williams et al. where it was shown that for memory-intensive applications, cores with a local store, such as Cell, were able to utilize a higher percentage of the available DRAM bandwidth, the target processor architecture includes a local store. In the Green Flash SMP design there will be two on-chip networks – as illustrated in Fig. 1. As can be somewhat expected, the majority of communication that occurs between sub-domains within the climate model is nearest neighbor. Building on work from both Balfour and Dally [11] a packet switched network with a concentrated torus topology was chosen for the SMP as it has been shown to provide superior performance and energy efficiency for codes where the dominant communication pattern is nearest neighbor.

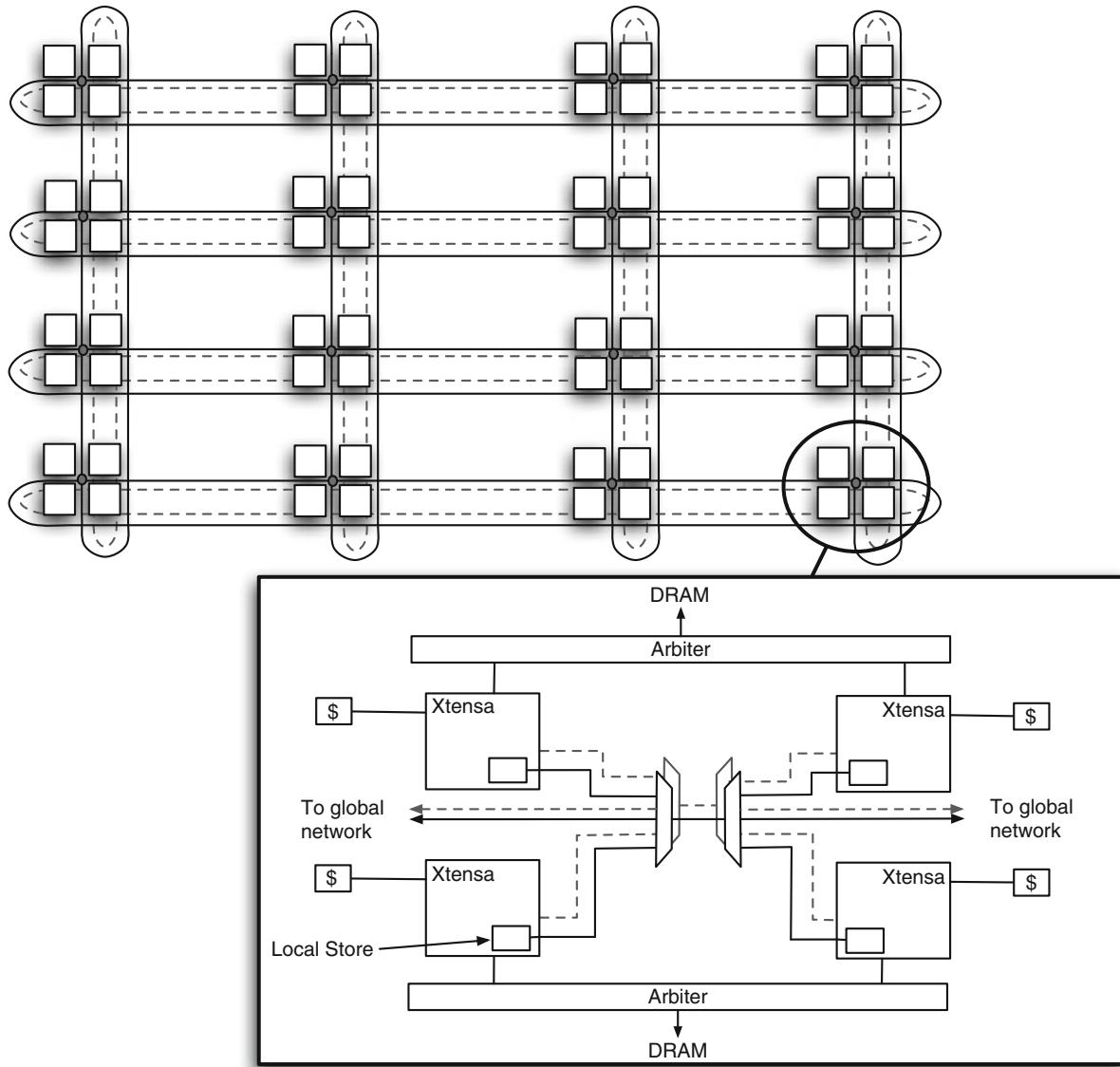
To further optimize the Green Flash processor for science the programming model is being considered a first class citizen when designing the architecture. Traditional cache coherent models found in many modern SMPs do not allow fine-grained synchronization between cores. In fact, when benchmarking the current climate model on present day machines it is shown that greater than 90% of execution time is spent in communication. By creating an architecture where an individual core will not pay a huge overhead penalty for sending or receiving a relatively small message the amount of time spent in communication can be greatly reduced. The processing cores used in the Green Flash SMP have powerful, flexible streaming interfaces. Each processor can have multiple, designer defined ports with a simple FIFO-like interface with each port capable of sending and receiving a packet of data on each clock. This low-overhead streaming interface will bypass the cache and connect to one of the torus networks on chip. This narrow network can be used for exchange of addresses, while the wider torus network is used for exchange of data. Following a Partitioned Global Address Space (PGAS) model the address space for each processor's local store is mapped into the global address space and the data exchange is done as a DMA from local store to local store. This allows the communication between processors to map very well to a MPI send/receive model used by the climate model and many other scientific codes. The view to the programmer will be as though all processors are directly connected to their

neighbors. To further simplify programming, a traditional cache hierarchy is also in place to allow codes to be slowly ported to the more efficient local-store based interprocessor network. In order to minimize power, the use of photonic interlinks for the inter-core network is being investigated as an efficient method of transferring long messages. In the case of Green Flash, the data network is one cache line in width and will consist of several phases per message.

Hardware/Software Codesign Strategy

Conventional approaches to hardware design generally have a long latency between hardware design and software development/optimization so designers frequently rely on benchmark codes to find a power-efficient architecture. However, modern compilers fail to generate even close to optimal code for target machines. Therefore, a benchmark-based approach to hardware design does not exploit the full performance potential of the architecture design points under consideration leading to possibly sub-optimal hardware solutions. The success of auto-tuners has shown that it is still possible to generate efficient code using domain knowledge. In combination with the ability to rapidly produce semi-custom hardware designs a tight, effective hardware/software codesign loop can be created. The codesign approach, as shown in Fig. 2 incorporates extensive software tuning into the process of hardware design. Hardware design space exploration is routinely done to tailor the hardware design parameters to the target applications. The auto-tuned software tailors the application to the hardware design point under consideration by empirically searching over a range of software implementations to find the best mapping of the software to the micro-architecture. One of the hindrances toward the practical relevance of codesign is the large hardware/software design space exploration. Conventional hardware design approaches use software simulation of hardware to perform hardware design space exploration. Because codesign involves searching over a much larger design space (there is now a need to explore the software design space at each hardware design point), codesign is impractical if software simulation of hardware is used.

Rather than be constrained by the limitations of a software simulation environment, it is possible instead to take advantage of the processor generation

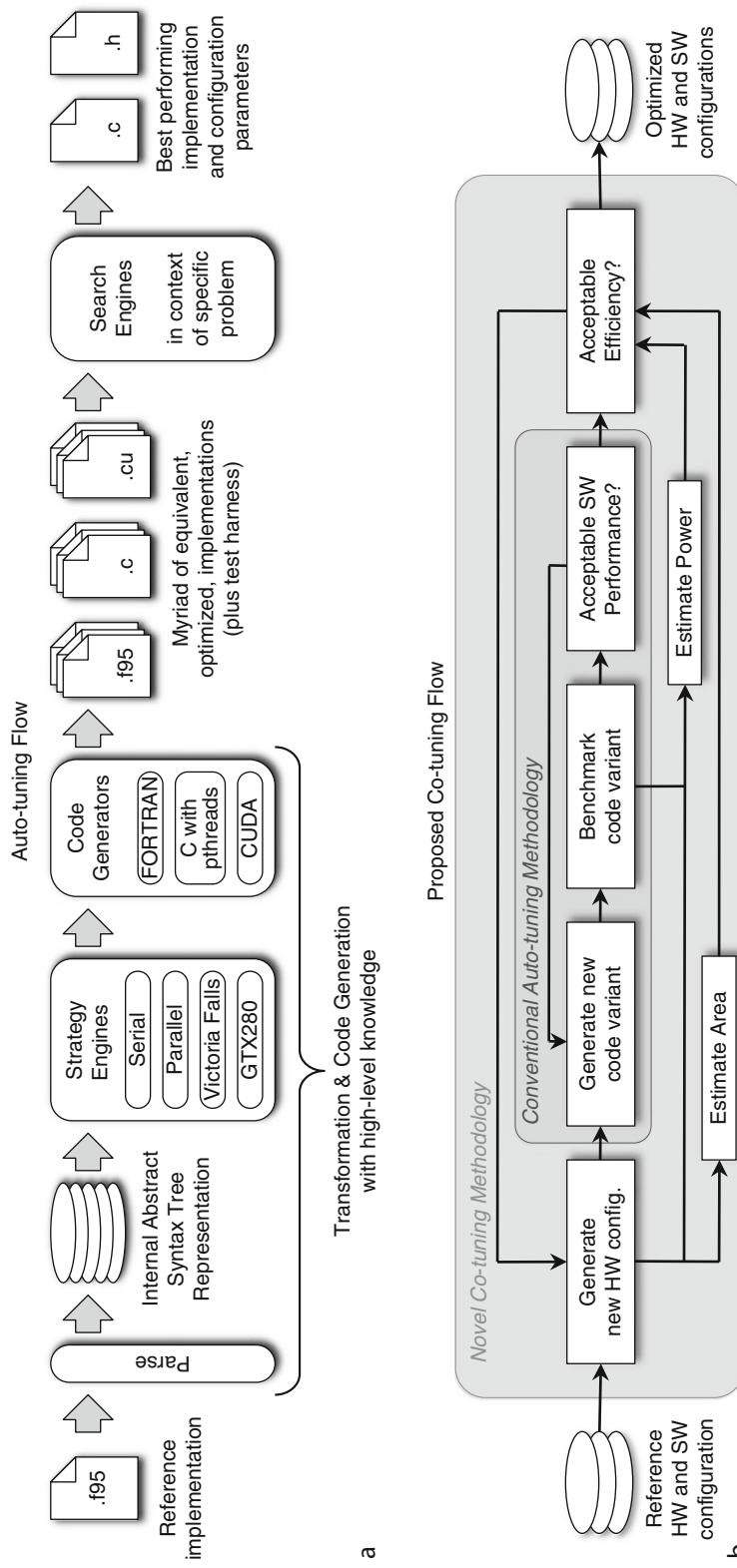


Green Flash: Climate Machine (LBNL). Fig. 1 A concentrated torus network fabric yields the highest performance and most power efficient design for scientific codes

toolchain's ability to create synthesizable RTL for any given processor. By loading this design onto an FPGA, a potential processor design can be emulated running 500 \times faster than a functional simulator. This speedup allows the benchmarking of true applications rather than being forced to rely on representative code snippets or statically defined benchmarks. Furthermore, this speed advantage does not come at the expense of accuracy; to the contrary, FPGA emulation is arguably much

more accurate than a software simulation environment as it truly represents the hardware design. This fast accurate emulation environment provides the ability to run and benchmark the actual climate model as it is being developed and allows the codesign infrastructure to quickly search a large design space.

The speed and accuracy advantages of using FPGAs have typically been dwarfed by the increased complexity of coding in Verilog or VHDL versus C++ or Python



Green Flash: Climate Machine (LBNL). Fig. 2 The result of combining an existing auto-tuning framework (a) with a rapid hardware design cycle and FPGA emulation is (b) a proposed hardware/software codesign flow

as well as the ability to emulate large designs due to limitations in FPGA area/LUT count. The practicality of using FPGAs for large system emulation has increased dramatically over the past decade. The ability to access relatively large dynamic memories, such as DDR, has always been a difficult challenge with FPGAs due to the tight timing requirements. FPGA vendors, such as Xilinx, have eased this difficulty by providing IP through its Memory Interface Generator (MIG) tool and adding IO features to the Virtex-5 series. Freely available Verilog IP libraries – whether they are Xilinx CoreGen, or the RAMP group's GateLib – allow for a modular, building block approach to HW design. Finally, while commercial microprocessors are experiencing a plateau in their clock rates and power consumption, FPGAs are not. FPGA LUT count continues to increase allowing the emulation of more complex designs and FPGA clocks, while traditionally significantly slower than commercial microprocessor clock rates have been growing steadily, closing the gap between emulated and production clock rates. In the case of Green Flash, the relatively low target clock frequency (500 MHz) of the final ASIC is an additional motivation to target an FPGA emulation environment. The current emulated processor design runs at 33 MHz – a significant fraction of the target clock rate. This relatively high speed enables the efficient benchmarking of an entire application rather than a representative portion.

While the steady growth in LUT count on FPGAs has enabled the emulation of more complex designs, with a strawman architecture of 128 cores per socket it is necessary to emulate more than the two or four cores that will fit on a single FPGA. To scale beyond the cores that will fit on a single FPGA a multi-FPGA system, such as the Berkeley Emulation Engine (BEE3) can be used. The BEE3 board has four Virtex-5 155 FPGAs connected in a ring with a cross-over connection. Each FPGA has access to two channels of DDR2 memory allowing 4 GB of memory per FPGA. The BEE3 will allow effective emulation of eight cores with the appropriate NoC infrastructure per board. To scale beyond eight cores, the BEE3 includes 10 Gb connections allowing the boards to be linked and emulation of an entire socket becomes possible. There is significant precedence for emulation of massively multithreaded architectures across multiple FPGAs. One recent example was demonstrated by the Berkeley RAMP Blue project

where over 1,000 cores were emulated using a stack of 16 BEE2 boards.

Hardware Support for New Programming Models

Applications and algorithms will need to rely increasingly on fine-grained parallelism and strong scaling and support fault resilience to accommodate the massive growth of explicit on-chip parallelism and constrained bandwidth anticipated for future chip architectures. History shows that the application-driven approach offers the most productive strategy for evaluating and selecting among the myriad choices for refactoring algorithms for full scientific application codes as the industry moves through this transitional phase. Green Flash functions as a testbed to explore novel programming models together with hardware support to express fine-grained parallelism to achieve performance, productivity, and correctness for leading-edge application codes in the face of massive parallelism and increasingly hierarchical hardware. The goal of this development thrust is to create a new software model that can provide a stable platform for software development for the next decade and beyond for all scales of scientific computing.

The Green Flash design created direct hardware support for both the message passing interface (MPI) and partitioned global address space (PGAS) programming models to enable scaling of these familiar single program, multiple data (SPMD) programming styles to much larger-scale systems. The modest hardware support enables relatively well-known programming paradigms to utilize massive on-chip concurrency and to use hierarchical parallelism to enable use of larger messages for interchip communication.

However, not all applications will be able to express parallelism through simple divide-and-conquer problem partitioning. So the message-queues and software-managed memories that are used to implement PGAS are also being used to explore new asymmetric and asynchronous approaches to achieving strong-scaling performance improvements from explicit parallelism. Techniques that resemble class static dataflow methods are garnering renewed interest because of their ability to flexibly schedule work and to accommodate state migration to correct load imbalances and failures. In the case of the climate code, dataflow techniques can be used to concurrently schedule the physics computations

with the dynamic core of the climate code, thereby doubling the effective concurrency without moving to a finer domain decomposition. This approach also benefits from the unique interprocessor communication interfaces developed for Green Flash.

Fault Resilience

A question that comes up when proposing a 20 million processor computing system is how to deal with fault resilience. While trying not to trivialize the issue, it should be noted that this is a problem for everyone designing large-scale machines. The proposed approach of using many simpler cores does not introduce any unique challenges that are different than the challenges faced for aggregating conventional server chips into large-scale systems provided the total number of discrete chips in the system is not dramatically different. The following observations are made to qualify this point.

For a given silicon process (e.g., a 65 nm process and same design rules)

1. Hard failure rates are primarily proportional to the number of sockets in a system (e.g., solder joint failures, weak wire bonds, and a variety of mechanical and electrical issues) and secondarily related to the total chip surface area (probability of defect vs tolerance of the design rules to process variation). It is not proportional to the number of processor cores per se given that the cores come in all shapes and sizes.
2. Soft error rates caused by cosmic rays roughly proportional to chip surface area when comparing circuits that employ the same process technology (e.g., 65 nm).
3. Bit error rates for data transfer tend to increase proportionally with clock rate.
4. Thermal stress is also a source of hard errors.

For hard errors:

1. Spare cores can be designed into each ASIC to tolerate defects due to process variation. This approach is already used by the 188 core Cisco Metro chip, which incorporates 8 spare cores (192 cores in total) to cover chip defects.
2. Each chip is expected to dissipate a relatively small 7–15 W (or that is the target) subjecting them to less mechanical/thermal stress.
3. It has been demonstrated that Green Flash can achieve more *delivered* performance out of fewer

sockets, which reduces exposure to hard-failures due to bad electrical connections or other mechanical/electrical defects.

4. Like BlueGene, memory and CPUs can be flow-soldered onto the board to reduce hard and soft failure rates for electrical connections given removable sockets are far more susceptible to both kinds of faults. So eliminating removable sockets can greatly reduce error rates.

For soft errors:

1. All of the basics for reliability and error recovery in the memory subsystem including full ECC (error correcting code) protection for caches and memory interfaces are included in the design.
2. Using many simpler cores allows fewer sockets to be used and less silicon surface area to achieve the same delivered performance. So that is to say, Green Flash has less exposure to major sources of failure than a conventional high-frequency core design. Therefore, fewer sockets and fewer random bit-flips due to mechanical noise and other stochastic error sources.
3. The core clock frequency of 500 MHz improves Signal to Noise Ratio for on-chip data transfers.
4. Incorporation of a Nonvolatile Random Access Memory (NVRAM) memory controller and channel on each System on Chip (SoC). Each node can copy the image of memory to the NVRAM periodically to do local checkpoints. If there is a soft-error (e.g., an uncorrectable memory error), then the node can initiate a roll-back to the last available checkpoint. For hard failures (e.g., a node does and cannot be revived), the checkpoint image will be copied to neighboring nodes on a periodic basis to facilitate localized state recovery. Both strategies enable much faster roll-back when errors are encountered than the conventional user-space checkpointing approach.

Therefore, the required fault resilience strategies will bear similarity to other systems that employ a similar number of sockets (~120,000), which are not unprecedented. The BlueGene system at Lawrence Livermore National Laboratory contains a comparable number of sockets, and achieves a 7–15 day Mean Time Between Failures (MTBF), which is far longer than systems that contain a fraction the number of processor cores. Therefore, careful application of well-known fault-resilience techniques together with a few

novel “extended fault resilience” mechanisms such as localized NVRAM checkpoints can achieve an acceptable MTBF for extreme-scale implementations of this approach to system design.

Conclusions

Green Flash proposes a radical approach of application-driven computing design to break through the slow pace of incremental changes, and foster a sustainable hardware/software ecosystem with broad-based support across the IT industry. Green Flash has enabled the exploration of practical advanced programming models together with lightweight hardware support mechanisms that allow programmers to utilize massive on-chip concurrency, thereby creating the market demand for massively concurrent components that can also be the building block of midrange and extreme-scale computing systems. New programming models must be part of a new software development ecosystem that spans all scales of systems, from midrange to the extreme-scale to facilitate a viable migration path from development to large-scale production computing systems. The use of the FPGA-based hardware emulation platforms, such as RAMP, to prototype and run hardware prototypes at near-realtime speeds before it is built allow testing of full-fledged application codes and advanced software development to commence many years before the final hardware platform is constructed. These tools have enabled a tightly coupled software/hardware codesign process that can be applied effectively to the complex HPC application space.

Rather than ask “what kind of scientific applications can run on our HPC cluster after it arrives,” the question should be turned around to ask “what kind of system should be built to meet the needs of the most important science problems.” This approach is able to realize its most substantial gains in energy-efficiency by peeling back the complexity of high-frequency microprocessor design point to reduce sources of waste (wasted opcodes, wasted bandwidth, waste caused by orienting architectures toward serial performance). BlueGene and SiCortex have demonstrated the advantages of using the simpler low-power embedded processing elements to create energy-efficient computing platforms. However, the Green Flash codesign approach goes beyond traditional embedded core design point of BlueGene and SiCortex by using explicit message queues and software controlled memories to further optimize

data movement, while still retaining a smaller conventional cache-hierarchy only to support incremental porting to the more energy and bandwidth-efficient design point. Furthermore, simple hardware support for lightweight on-chip interprocessor synchronization and communication make it much simpler and straightforward and efficient to program massive arrays of processors than more exotic programming models such as CUDA and Streaming.

Green Flash has been a valuable research vehicle to understand how the evolution of massively parallel chip architectures can be guided by close-coupled feedback with the design of the application, algorithms, and hardware together. Application-driven design ensures hardware design decisions do not evolve in reaction to hardware constraints, without regard to programmability and delivered application performance. The design study has been driven by a deep dive into the climate application space, but enables explorations that cut across all application areas and have ramifications to the next generation of fully general-purpose architectures. Ultimately, Green Flash should consist of an architecture that can maximally leverage reusable components from the mass market of the embedded space while improving the programmability for the many-core design point. The building blocks of a future HPC system must be the preferred solution in terms of performance and programmability for everything from the smallest high-performance energy-efficient embedded system, to midrange departmental systems, to the largest-scale systems.

Bibliography

- Asanovic K, Bodik R, Catanzaro BC, Gebis JJ, Husbands P, Keutzer K, Patterson DA, Plishker WL, Shalf J, Williams SW, Yelick KA (2006) The landscape of parallel computing research: a view from Berkeley. Technical report no. UCB/EECS-2006-183, EECS Department University of California, Berkeley
- Wehner M, Oliker L, Shalf J (2008) Towards ultra-high resolution models of climate and weather. *Int J High Perform Comput Appl* 22:149–165
- Shalf J (2007) The new landscape of parallel computer architecture. *J Phys: Conf Ser* 78:012066
- Donofrio D, Oliker L, Shalf J, Wehner MF, Rowen C, Krueger J, Kamil S, Mohiyuddin M (2009) Energy-efficient computing for extreme-scale science. *IEEE Computer*, Los Almitos
- Wehner M, Oliker L, Shalf J (2009) Low-power supercomputers. *IEEE Spectrum* 29(9):66–68
- Shalf J, Wehner M, Oliker L, Hules J (2009) Green flash project: the challenge of energy-efficient HPC. *SciDAC Review*, Fall

7. Kamil SA, Shalf J, Oliker L, Skinner D (2005) Understanding ultra-scale application communication requirements. In: IEEE international symposium on workload characterization (IISWC) Austin, 6–8 Oct 2005 (LBNL-58059)
8. Kamil S, Chan Cy, Oliker L, Shalf J, Williams S (2010) An auto-tuning framework for parallel multicore stencil computations. In: IPDPS 2010, Atlanta
9. Hendry G, Kamil SA, Biberman A, Chan J, Lee BG, Mohiyuddin M, Jain A, Bergman K, Carloni LP, Kubiatocis J, Oliker L, Shalf J (2009) Analysis of photonic networks for chip multiprocessor using scientific applications. In: NOCS 2009, San Diego
10. Mohiyuddin M, Murphy M, Oliker L, Shalf J, Wawrzynek J, Williams S (2009) A design methodology for domain-optimized power-efficient supercomputing, In: SC 09, Portland
11. Balfour J, Dally WJ (2006) Design tradeoffs for tiled cmp on-chip networks. In ICS '06: Proceedings of the 20th annual international conference on supercomputing

Grid Partitioning

► Domain Decomposition

Gridlock

► Deadlocks

Group Communication

► Collective Communication

Gustafson's Law

JOHN L. GUSTAFSON
Intel Labs, Santa Clara, CA, USA

Synonyms

Gustafson–Barsis Law; Scaled speedup; Weak scaling

Definition

Gustafson's Law says that if you apply P processors to a task that has serial fraction f , scaling the task to take the

same amount of time as before, the speedup is

$$\begin{aligned}\text{Speedup} &= f + P(1 - f) \\ &= P - f(P - 1).\end{aligned}$$

It shows more generally that the serial fraction does not theoretically limit parallel speed enhancement, if the problem or workload scales in its parallel component. It models a different situation from that of Amdahl's Law, which predicts time reduction for a fixed problem size.

Discussion

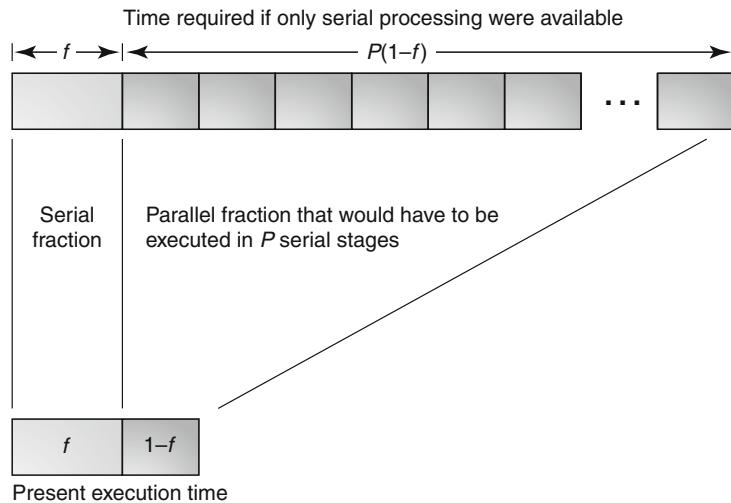
Graphical Explanation

Figure 1 explains the formula in the Definition:

The time the user is willing to wait to solve the workload is unity (lower bar). The part of the work that is observably serial, f , is unaffected by parallelization. The remaining fraction of the work, $1 - f$, parallelizes perfectly so that a serial processor would take P times longer to execute it. The ratio of the top bar to the bottom bar is thus $f + P(1 - f)$. Some prefer to rearrange this algebraically as $P - f(P - 1)$.

The diagram resembles the one used in the explanation of Amdahl's Law (see ►Amdahl's Law) except that Amdahl's Law fixes the *problem size* and answers the question of how parallel processing can reduce the execution time. Gustafson's Law fixes the *run time* and answers the question of how much longer time the present workload would take in the absence of parallelism [5]. In both cases, f is the experimentally observable fraction of the current workload that is serial. The similarity of the diagram to the one that explains Amdahl's Law has led some to "unify" the two laws by a change of variable. It is an easy algebraic exercise to set the upper bar to unit time and express the f of Gustafson's Law in terms of the variables of Amdahl's Law, but this misses the point that the two laws proceed from *different premises*. Every attempt at unification begins by applying the same premise, resulting in a circular argument that the two laws are the same.

The fundamental underlying observation of Gustafson's Law is that more powerful computer systems usually solve larger problems, not the same size problem in less time. Hence, a performance enhancement like parallel processing expands what a user can do with a computing system to match the time the user is willing to wait for the answer. While computing power has increased by many orders of magnitude over the last



Gustafson's Law. Fig. 1 Graphical derivation of Gustafson's Law

half-century (see ▶ [Moore's Law](#)), the execution time for problems of interest has been constant, since that time is tied to human timescales.

History

In a 1967 conference debate over the merits of parallel computing, IBM's Gene Amdahl argued that a considerable fraction of the work of computers was inherently serial, from both algorithmic and architectural sources. He estimated the serial fraction f at about 0.25–0.45. He asserted that this would sharply limit the approach of parallel processing for reducing execution time [1]. Amdahl argued that even the use of two processors was less cost-effective than a serial processor. Furthermore, the use of a large number of processors would never reduce execution time by more than $1/f$, which by his estimate was a factor of about 2–4.

Despite many efforts to find a flaw in Amdahl's argument, "Amdahl's Law" held for over 20 years as justification for the continued use of serial computing hardware and serial programming models.

The Rise of Microprocessor-Based Systems

By the late 1970s, microprocessors and dynamic random-access memory (DRAM) had dropped in price to the point where academic researchers could afford them as components in experimental parallel designs. Work in 1983 by Charles Seitz at Caltech using a message-passing collection of 64 microprocessors [11]

showed excellent absolute performance in terms of floating-point operations per second, and seemed to defy Amdahl's pessimistic prediction. Seitz's success led John Gustafson at FPS to drive development of a massively parallel cluster product with backing from the Defense Advanced Research Projects Agency (DARPA). Although the largest configuration actually sold of that product (the FPS T Series) had only 256 processors, the architecture permitted scaling to 16,384 processors. The large number of processors led many to question: *What about Amdahl's Law?* Gustafson formulated a counterargument in April 1986, which showed that performance is a function of both the problem size and the number of processors, and thus Amdahl's Law need not limit performance. That is, the serial fraction f is not a constant but actually decreases with increased problem size. With no experimental evidence to demonstrate the idea, the counterargument had little impact on the computing community.

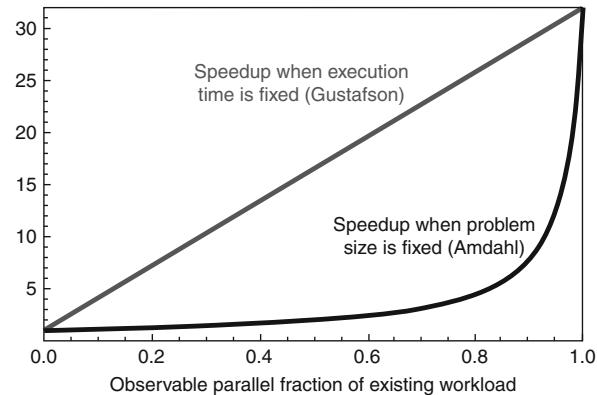
An idea for a source of experimental evidence arose in the form of a challenge that Alan Karp had publicized the year before [8]. Karp had seen announcements of the 65,536-processor CM-1 from Thinking Machines and the 1,024-processor NCUBE10 from nCUBE, and believed Amdahl's Law made it unlikely that such massively parallel computers would achieve a large fraction of their rated performance. He published a skeptical challenge and a financial reward for anyone who could demonstrate a parallel speedup of over 200 times on

three real applications. Karp suggested computational fluid dynamics, structural analysis, and econometric modeling as the three application areas and gave some ground rules to insure that entries focused on honest parallel speedup without tricks or workarounds. For example, one could not cripple the serial version to make it artificially 200 times slower than the parallel system. And the applications, like the three suggested, had to be ones that had interprocessor communication throughout their execution as opposed to “embarrassingly parallel” problems that had communication only at the beginning and end of a run.

By 1987, no one had met Karp’s challenge, so Gordon Bell adopted the same set of rules and suggested applications as the basis for the Gordon Bell Award, softening the goal from 200 times to whatever the best speedup developers could demonstrate. Bell expected the initial entries to achieve about tenfold speedup [2].

The purchase by Sandia National Laboratories of the first 1,024-processor NCUBE 10 system created the opportunity for Gustafson to demonstrate his argument on the experiment outlined by Karp and Bell, so he joined Sandia and worked with researchers Gary Montry and Robert Benner to demonstrate the practicality of high parallel speedup. Sandia had real applications in fluid dynamics and structural mechanics, but none in econometric modeling, so the three researchers substituted a wave propagation application. With a few weeks of tuning and optimization, all three applications were running at over 500-fold speedup with the fixed-size Amdahl restriction, and over 1,000-fold speedup with the scaled model proposed by Gustafson. Gustafson described his model to Sandia Director Edwin Barsis, who suggested explaining scaled speedup using a graph like that shown in Fig. 2.

Barsis also insisted that Gustafson publish this concept, and is probably the first person to refer to it as “Gustafson’s Law.” With the large experimental speedups combined with the alternative model, *Communications of the ACM* published the results in May 1988 [5]. Since Gustafson credited Barsis with the idea of expressing the scaled speedup model as graphed in Fig. 2, some refer to Gustafson’s Law as the Gustafson–Barsis Law. The three Sandia researchers



Gustafson's Law. Fig. 2 Speedup possible with 32 processors, by Gustafson's Law and Amdahl's Law

G

published the detailed explanation of the application parallelizations in a *Society of Industrial and Applied Mathematics* (SIAM) journal [4].

Parallel Computing Watershed

Sandia’s announcement of 1,000-fold parallel speedups created a sensation that went well beyond the computing research community. Alan Karp announced that the Sandia results had met his Challenge, and Gordon Bell gave his first award to the three Sandia researchers. The results received publicity beyond that of the usual technical journals, appearing in *TIME*, *Newsweek*, and the US Congressional Record. Cray, IBM, Intel, and Digital Equipment began work in earnest developing commercial computers with massive amounts of parallelism for the first time.

The Sandia announcement also created considerable controversy in the computing community, partly because some journalists sensationalized it as a proof that Amdahl’s Law was false or had been “broken.” This was never the intent of Gustafson’s observation. He maintained that Amdahl’s Law was the correct answer but to the wrong question: “How much can parallel processing reduce the run time of a current workload?”

Observable Fraction and Scaling Models

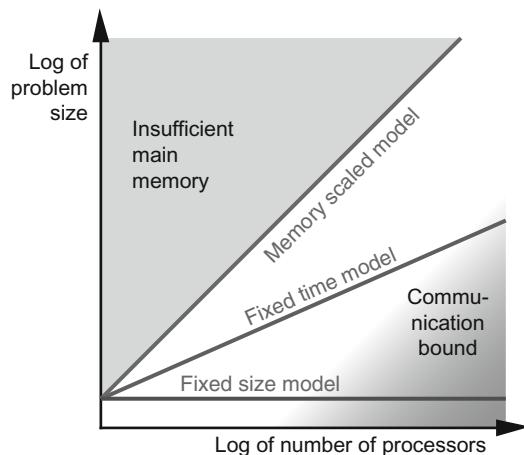
As part of the controversy, many maintained that Amdahl’s Law was still the appropriate model to use in all situations, or that Gustafson’s Law was simply

a corollary to Amdahl's Law. For scaled speedup, the argument went that one simply works backward to determine what the f fraction in Amdahl's Law must have been to yield such performance. This is an example of circular reasoning, since the proof that Amdahl's Law applies begins by assuming it applies.

For many programs, it is possible to instrument and measure the fraction of time f spent in serial execution. One can place timers in the program around serial regions and obtain an estimate of f . This fraction then allows Amdahl's Law estimates of time reduction, or Gustafson's Law estimates of scaled speedup. Neither law takes into account communication costs or intermediate degrees of parallelism. (When communication costs are included in Gustafson's fixed-time model, the speedup is again limited as the number of processors grows, because communication costs rise to the point where there is no way to increase the size of the amount of work without increasing the execution time.)

A more common practice is to measure the parallel speedup as the number of processors is varied, and fit the resulting curve to derive f . This approach confuses serial fraction with communication overhead, load imbalance, changes in the relative use of the memory hierarchy, and so on. Some refer to the requirement to keep the problem size the same yet use more processors as "strong scaling." Still, a common phenomenon that results from "strong scaling" is that it is *easier*, not *harder*, to obtain high amounts of speedup. When spreading a problem across more and more processors, the *memory per processor* goes down to the point where the data fits entirely in cache, resulting in *superlinear speedup* [3]. Sometimes, the superlinear speedup effects and the communication overheads partially cancel out, so what appears to be a low value of f is actually the result of the combination of the two effects. In modern parallel systems, performance analysis with either Amdahl's Law or Gustafson's Law will usually be inaccurate since communication costs and other parallel processing phenomena have large effects on the speedup.

In Fig. 3, Amdahl's Law governs the Fixed-Sized Model line, Gustafson's Law governs the Fixed-Time Model line, and what some call the Sun-Ni Law governs the Memory Scaled Model [12]. The fixed-time model line is an irregular curve in general, because of



Gustafson's Law. Fig. 3 Different scaling types and communication costs

the communication cost effects and because the percentage of the problem that is in each memory tier (mass storage, main RAM, levels of cache) changes with the use of more processors.

Analogy

There are many aspects of technology where an enhancement for time reduction actually turns out to be an enhancement for what one can accomplish in the same time as before. Just as Amdahl's Law is an expression of the more general Law of Diminishing Returns, Gustafson's Law is an expression of the more general observation that technological advances are used to improve what humans accomplish in the length of time they are accustomed to waiting, not to shorten the waiting time.

Commuting Time

As civilization has moved from walking to horses to mechanical transportation, the average speed of getting to and from work every day has gone up dramatically. Yet, people take about half an hour to get to or from work as a tolerable fraction of the day, and this amount of time is probably similar to what it has been for centuries. Cities that have been around for hundreds or thousands of years show a concentric pattern that reflect the increasing distance people could commute for the amount of time they were able to tolerate.

Transportation provides many analogies for Gustafson's Law that expose the fallacy of fixing the size of a problem as the control variable in discussing large performance gains. A commercial jet might be able to travel 500 miles per hour, yet if one asks "How much will it reduce the time it takes me presently to walk to work and back?" the answer would be that it does not help at all. It would be easy to apply an Amdahl-type argument to the time to travel to an airport as the serial fraction, such that the speedup of using a jet only applies to the remaining fraction of the time and thus is not worth doing. However, this does not mean that commercial jets are useless for transportation. It means that faster devices are for larger jobs, which in this case means longer trips.

Here is another transportation example: If one takes a trip at 30 miles per hour and immediately turns around, how fast does one have to go to average 60 miles per hour? This is a trick question that many people incorrectly answer, "90 miles per hour." To average 60 miles per hour, one would have to travel back at infinite speed, that is, return *instantly*. Amdahl's Law applies to this fixed-distance trip. However, suppose the question were posed differently: "If one travels for an hour at 30 miles per hour, how fast does one have to travel in the next hour to average 60 miles per hour?" In that case, the intuitive answer of "90 miles per hour" is the correct one. Gustafson's Law applies to this fixed-time trip.

The US Census

In the early debates about scaled speedup, Heath and Worley [6] provided an example of a fixed-sized problem that they said was not appropriate for Gustafson's Law and for which Amdahl's Law should be applied: the US Census. While counting the number of people in the USA would appear to be a fixed-sized problem, it is actually a perfect example of a fixed-time problem since the Constitution mandates a complete headcount every 10 years. It was in the late nineteenth century, when Hollerith estimated that the population had grown to the point where existing approaches would take longer than 10 years that he developed the card punch tabulation methods that made the process fast enough to fit the fixed-time budget.

With much faster computing methods now available, the Census process has grown to take into

account many more details about people than the simple head count that the Constitution mandates. This illustrates a connection between Gustafson's Law and the jocular Parkinson's Law: "Work expands to fill the available time."

Printer Speed

In the 1960s, when IBM and Xerox were developing the first laser printers that could print an entire page at a time, the goal was to create printers that could print several pages per second so that printer speed could match the performance improvements of computing speed. The computer printouts of that era were all of monospaced font with a small character set of uppercase letters and a few symbols. Although many laser printer designers struggled to produce such simple output with reduced time per page, the product category evolved to produce high quality output for desktop publishing instead of using the improved technology for time reduction. People now wait about as long for a page of printout from a laser printer as they did for a page of printout from the line printers of the 1960s, but the task has been scaled up to full color, high resolution printing encompassing graphics output, and a huge collection of typeset fonts from alphabets in all the world's languages. This is an example of Gustafson's Law applied to printing technology.

Biological Brains

Kevin Howard, of Massively Parallel Technologies Inc., once observed that if Amdahl's Law governed the behavior of biological brains, then a human would have about the same intelligence as a starfish. The human brain has about 100 billion neurons operating in parallel, so for us to avoid passing the point of diminishing returns for all that parallelism, the Amdahl serial fraction f would have to be about 10^{-14} . The fallacy of this seeming paradox is in the underlying assumption that a human brain must do the same task a starfish brain does, but must reduce the execution time to nanoseconds. There is no such requirement, and a human brain accomplishes very little in a few nanoseconds no matter how many neurons it uses at once. Gustafson's Law says that on a time-averaged basis, the human brain will accomplish *vastly more complex tasks* than what a starfish can attempt, and thus avoids the absurd conclusion of the fixed-task model.

Perspective

The concept of scaled speedup had a profound enabling effect on parallel computing, since it showed that simply asking a different question (and perhaps a more realistic one) renders the pessimistic predictions of Amdahl's Law moot. Gustafson's 1988 announcement of 1,000-fold parallel speedup created a turning point in the attitude of computer manufacturers towards massively parallel computing, and now all major vendors provide platforms based on the approach. Most (if not all) of the computer systems in the TOP500 list of the worlds' fastest supercomputers are comprised of many thousands of processors, a degree of parallelism that computer builders regarded as sheer folly prior to the introduction of scaled speedup in 1988.

A common assertion countering Gustafson's Law is that "Amdahl's Law still holds for scaled speedup; it's just that the serial fraction is a lot smaller than had been previously thought." However, this requires inferring the small serial fraction from the measured speedup. This is an example of circular reasoning since it involves choosing a conclusion, then working backward to determine the data that make the conclusion valid. Gustafson's Law is a simple formula that predicts scaled performance from experimentally measurable properties of a workload.

Some have misinterpreted "scaled speedup" as simply increasing the amount of memory for variables, or increasing the fineness of a grid. It is more general than this. It applies to every way in which a calculation can be improved somehow (accuracy, reliability, robustness, etc.) with the addition of more processing power, and then asks *how much longer the enhanced problem would have taken to run without the extra processing power*.

Horst Simon, in his 2005 keynote talk at the International Conference on Supercomputing, "*Progress in Supercomputing: The Top Three Breakthroughs of the Last 20 Years and the Top Three Challenges for the Next 20 Years*," declared the invention of the Gustafson's scaled speedup model as the number one achievement in high-performance computing since 1985.

Related Entries

- [Amdahl's Law](#)
- [Distributed-Memory Multiprocessor](#)
- [Metrics](#)

Bibliographic Entries and Further Reading

Gustafson's 1988 two-page paper in the *Communications of the ACM* [5] outlines his basic idea of fixed-time performance measurement as an alternative to Amdahl's assumptions. It contains the rhetorical question, "How can this be, in light of Amdahl's Law?" that some misinterpreted as a serious plea for the resolution of a paradox. Readers may find a flurry of responses in *Communications* and elsewhere, as well as attempts to "unify" the two laws.

An objective analysis of Gustafson's Law and its relation to Amdahl's Law can be found in many modern textbooks on parallel computing such as [7], [9], or [10]. In much the way some physicists in the early twentieth century refused to accept the concepts of relativity and quantum mechanics, for reasons more intuition-based than scientific, there are computer scientists who refuse to accept the idea of scaled speedup and Gustafson's Law, and who insist that Amdahl's Law suffices for all situations.

Pat Worley analyzed the extent to which one can usefully scale up scientific simulations by increasing their resolution [13]. In related work, Xian-He Sun and Lionel Ni built a more complete mathematical framework for scaled speedup [12] in which they promote the idea of memory-bounded scaling, even though execution time generally increases beyond human patience when the memory used by a problem scales as much as linearly with the number of processors. In a related vein, Vipin Kumar proposed "Isoefficiency" for which the memory increases as much as necessary to keep the efficiency of the processors at a constant level even when communication and other impediments to parallelism are taken into account.

Bibliography

1. Amdahl GM (1967) Validity of the single-processor approach to achieve large scale computing capabilities. AFIPS Joint Spring Conference Proceedings 30 (Atlantic City, NJ, Apr. 18–20), pp 483–485. AFIPS Press, Reston VA. At http://www-inst.eecs.berkeley.edu/~n_252/paper/Amdahl.pdf
2. Bell G (interviewed) (1987) An interview with Gordon Bell. IEEE Software, vol 4, No. 4 (July 1987), pp 102–104
3. Gustafson JL (1990) Fixed time, tiered memory, and superlinear speedup. Distributed Memory Computing Conference, 1990, Proceedings of the Fifth, vol 2 (April 1990), pp 1255–1260. ISBN: 0-8186-2113-3

4. Gustafson JL, Montry GR, Benner RE (1988) Development of parallel methods for a 1024-processor hypercube. *SIAM Journal on Scientific and Statistical Computing*, vol 9, No. 4, (July 1988), pp 609–638
5. Gustafson (1988) Reevaluating Amdahl's Law. *Communications of the ACM*, vol 31, No. 5 (May 1988), pp 532–533. DOI=[10.1145/42411.42415](https://doi.org/10.1145/42411.42415)
6. Heath M, Worley P (1989) Once again, Amdahl's Law. *Communications of the ACM*, vol 32, No. 2 (February 1989), pp 258–264
7. Hwang K, Briggs F, Computer Architecture and Parallel Processing, 1990. McGraw-Hill Inc., 1990. ISBN: 0070315566
8. Karp A (1985) <http://www.netlib.org/benchmark/karp-challenge>
9. Lewis TG, El-Rewini H (1992) Introduction to Parallel Computing, Prentice Hall. ISBN: 0-13-498924-4. 32–33
10. Quinn M (1994) Parallel Computing: Theory and Practice. Second edition. McGraw-Hill, Inc
11. Seitz CL (1986) Experiments with VLSI ensemble machines. *Journal of VLSI and Computer Systems*, vol 1, No. 3, pp 311–334
12. Sun X-H, Ni L (1993) Scalable problems and memory-bounded speedup." *Journal of Parallel and Distributed Computing*, vol 19, No. 1, pp 22–37
13. Worley PH (1989) The effect of time constraints on scaled speedup. Report ORNL/TM 11031, Oak Ridge National Laboratory

Gustafson–Barsis Law

► Gustafson's Law

H

Half Vector Length

- ▶ [Metrics](#)

Hang

- ▶ [Deadlocks](#)

Harmful Shared-Memory Access

- ▶ [Race Conditions](#)

Haskell

- ▶ [Glasgow Parallel Haskell \(GpH\)](#)

Hazard (in Hardware)

- ▶ [Dependences](#)

HDF5

QUINCEY KOZIOL
The HDF Group, Champaign, IL, USA

Synonyms

Hierarchical data format

Definition

HDF5 [1] is a data model, software library, and file format for storing and managing data.

Discussion

Introduction

The HDF5 technology suite is designed to organize, store, discover, access, analyze, share, and preserve diverse, complex data in continuously evolving heterogeneous computing and storage environments. It supports an unlimited variety of datatypes, and is designed for flexible and efficient I/O and for high volume and complex data. The HDF5 library and file format are portable and extensible, allowing applications to evolve in their use of HDF5. The HDF5 technology suite also includes tools and applications for managing, manipulating, viewing, and analyzing data in the HDF5 format.

Originally designed within the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign [2], HDF5 is now primarily developed and maintained by The HDF Group [3], a nonprofit organization dedicated to ensuring the sustainable development of HDF technologies and the ongoing accessibility of data stored in HDF files. HDF5 builds on lessons learned from other data storage libraries and file formats, such as the original HDF file format (now known as HDF4 [4]), netCDF [5], TIFF [6], and FITS [7], while adding unique features and extending the boundaries of prior data storage models.

Data Model

HDF5 implements a simple but versatile data model, which has two primary components: groups and datasets. Group objects in an HDF5 file contain a collection of named links to other objects in an HDF5 file. Dataset objects in HDF5 files store arrays of arbitrary element types and are the main method for storing application data.

Groups, which are analogous to directories in a traditional file system, can contain an arbitrary number of uniquely named links. A link can connect a group to another object in the same HDF5 file; include a named

path to an object in the HDF5 file, which may not exist currently; or refer to an object in another HDF5 file. Unlike links in a traditional file system, HDF5 links can be used to create fully cyclic directed graph structures. Each group contains one or more B-tree data structures as indices to its collection of links, which are stored in a heap structure within the HDF5 file.

Dataset objects store application data in an HDF5 file as a multidimensional array of elements. Each dataset is primarily defined by the description of how many dimensions its array has and the size of those dimensions, called a “dataspace”; and the description of the type of element to store at each location in the array, called a “datatype”.

An HDF5 dataspace describes the number of dimensions for an array, as well as the current and maximum number of elements in each dimension. The maximum number of elements in an array dimension can be specified as “unlimited,” allowing an array to be extended over time. An HDF5 dataspace can have multiple dimensions that have unlimited maximum dimensions, allowing that array to be extended in any or all of those dimensions.

An HDF5 datatype describes the type of data to store in each element of an array and can be one of the following classes: integer, floating-point, string, bitfield, opaque, compound, reference, enum, variable-length sequence, and array. These classes generally correspond to the analogous computer science concepts, but the reference and variable-length sequence datatypes are unusual. Reference datatypes contain links to HDF5 objects, allowing HDF5 applications to create datasets that act like groups. The former contain references or pointers to HDF5 objects, allowing HDF5 applications to create datasets that can act as lookup tables or indices. Variable-length sequence datatypes allow a dynamic number of elements of a base datatype to be stored as an element and are one mechanism for creating datasets that represent ragged arrays. All of the HDF5 datatypes can be combined in any arbitrary way, allowing for great flexibility in how an application stores its data.

The elements of an HDF5 dataset can be stored in different ways, allowing an application to choose between various I/O access performance trade-offs.

Dataset elements can be stored as a single sequence in the HDF5 file, called “contiguous” storage, which

allows for constant time access to any element in the array and no storage overhead for locating the elements in the dataset. However, contiguous data storage does not allow a dataset to use a dataspace with unlimited dimensions or to compress the dataset elements.

The dataspace for a dataset can also be decomposed into fixed-size sub-arrays of elements, called “chunks,” which are stored individually in the file. This “chunked” data storage requires an index for locating the chunks that store the data elements. Datasets that have a data space with unlimited dimensions must use chunked data storage for storing their elements.

Using chunked data storage allows an application that will be accessing sub-arrays of the dataset to tune the chunk size to its sub-array size, allowing for much faster access to those sub-arrays than would be possible with contiguous data storage. Additionally, the elements of datasets that use chunked data storage can be compressed or have other operations, like checksums, etc., applied to them.

The advantages of chunked data storage are balanced by some limitations, however. Using an index for mapping dataset element coordinates to chunk locations in the file can slow down access to dataset elements if the application’s I/O access pattern does not line up with the chunk’s sub-array decomposition. Furthermore, there is additional storage overhead for storing an index for the dataset, along with extra I/O operations to access the index data structure.

Datasets with very small amounts of element data can store their elements as part of the dataset description in the file, avoiding any extra I/O accesses to retrieve the dataset elements, since the HDF5 library will read them when accessing the dataset description. This “compact” data storage must be very small (less than 64 KB), and may not be used with a dataspace that has unlimited dimensions or when dataset elements are compressed.

Finally, a dataset can store its elements in a different, non-HDF5, file. This “external” data storage method can be used to share dataset elements between an HDF5 application and a non-HDF5 application. As with contiguous data storage, external data storage does not allow a dataset to use a dataspace with unlimited dimensions or compress the dataset elements.

HDF5 also allows application-defined metadata to be stored with any object in an HDF5 file.

These “attributes” are designed to store information about the object they are attached to, such as input parameters to a simulation, the name of an instrument gathering data, etc. Attributes are similar to datasets in that they have a dataspace, a datatype, and elements values. Attributes require a name that is unique among the attributes for an object, similar to link names within groups. Attributes are limited to dataspaces that do not use unlimited maximum dimensions and cannot have their data elements compressed, but can use any datatype for their elements.

Examples of Using HDF5

The following C code example shows how to use the HDF5 library to store a large array. In the example below, the data to be written to the file is a three-dimensional array of single-precision floating-point values, with dimensions of 1024 elements along each axis:

```

1 float data[1024][1024][1024];
2 hid_t file_id, dataspace_id, dataset_id;
3 hsize_t dims[3] = {1024, 1024, 1024};
4
5 <...acquire or assign data values...>
6
7 file_id = H5Fcreate("example.h5",
8 H5F_ACC_TRUNC, H5P_DEFAULT,
9 H5P_DEFAULT);
10
11 dataset_id = H5Dcreate(file_id, "/Float_data",
12 H5T_NATIVE_FLOAT, dataspace_id,
13 H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
14
15 H5Dwrite(dataset_id, H5T_NATIVE_FLOAT,
16 dataspace_id, dataspace_id, H5P_DEFAULT,
17 data);
18
19 H5Dclose(dataset_id);
20
21 H5Sclose(dataspace_id);
22
23 H5Fclose(file_id);
```

In this example, lines 1–3 declare the variables needed for the example, including the 3-D array of data to store. Line 5 represents the application’s process of filling the data array with information. Lines 7–9 create a new HDF5 file, a new dataspace describing a fixed-size three-dimensional array of dimensions $1024 \times 1024 \times 1024$, and a new dataset using a single-precision floating-point datatype and the dataspace created. Line 11 writes the entire 4 GB array to the file in a single I/O operation, and lines 13–15 close the objects created earlier. Several of the calls use H5P_DEFAULT as a parameter, which is a placeholder for an HDF5 property list object, which can control more complicated properties of objects or operations.

The next C code example creates an identically structured file, but adds the necessary calls to open the file with 8 processes in parallel and to perform a collective write to the dataset created.

1	float data[512][512][512];
2	hid_t file_id, file_dataspace_id, mem_dataspace _id, dataset_id, fa_plist_id, dx_plist_id;
3	hsize_t file_dims[3] = {1024, 1024, 1024};
4	hsize_t mem_dims[3] = {512, 512, 512};
5	
6	<...acquire or assign data values...>
7	
8	fa_plist_id = H5Pcreate(H5P_FILE_ACCESS);
9	H5Pset_fapl_mpio(fa_plist_id, MPI_COMM_WORLD, MPI_INFO_NULL);
10	file_id = H5Fcreate("example.h5", H5F_ACC_ TRUNC, H5P_DEFAULT, fa_plist_id);
11	H5Pclose(fa_plist_id);
12	file_dataspace_id = H5Screate_simple(3, file_dims, NULL);
13	dataset_id = H5Dcreate(file_id, "/Float_data", H5T_NATIVE_FLOAT, file_dataspace_id, H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
14	
15	mem_dataspace_id = H5Screate_simple(3, mem_dims, NULL);

16	
17	< ...select process's elements in file dataspace... >
18	
19	dx plist_id = H5Pcreate(H5P_DATASET_XFER);
20	H5Pset_dxpl_mpio(dx plist_id, H5FD_MPIO_COLLECTIVE);
21	H5Dwrite(dataset_id, H5T_NATIVE_FLOAT, mem dataspace_id, file dataspace_id, dx plist_id, data);
22	
23	H5Pclose(dx plist_id);
24	H5Dclose(dataset_id);
25	H5Sclose(mem dataspace_id);
26	H5Sclose(file dataspace_id);
27	H5Fclose(file id);

In this updated example, the size of the data array on line 1 has been changed to be only one eighth of the total array size, to allow for each of the eight processes to write a portion of the total array in the file. Lines 8–10 have been updated to create a file access property list, change the file driver for opening the file to use MPI-I/O and collectively open the file with all processes, using the file access property list. Lines 12–13 create the dataset in the file in the same way as the previous example. Line 15 creates a dataspace for each process's portion of the dataset in the file, and line 17 represents a section of code for selecting the part of the file's dataspace that each process will write to (which is omitted due to space constraints). Lines 19–21 create a dataset transfer property list, set the I/O operation to be collective, and perform a collective write operation where each process writes a different portion of the dataset in the file. Finally, lines 23–27 release resources used for the example.

This example shows some ways that property lists can be used to modify the operation of HDF5 API calls, as well as demonstrating a simple example of parallel I/O using HDF5 API calls.

Higher-Level Data Models Built on HDF5

HDF5 provides a set of generic higher-level data models that describe how to store images and tables as

datasets and describe the coordinates of dataspace elements. A scientific user community can also use HDF5 as the basis for exchanging data among its members by creating a standardized domain-specific data model that is relevant to their area of interest. Domain-specific data models specify the names of HDF5 groups, datasets and attributes, the dataspace and datatype for the datasets and attributes, etc. Frequently, a domain's user community also creates a “wrapper library” that calls HDF5 library routines while enforcing the domain's standardized data model.

Library Interface

Software applications create, modify, and delete HDF5 objects through an object-oriented library interface that manipulates the objects in HDF5 files. Applications can use the HDF5 library to operate directly on the base HDF5 objects or use a domain-specific wrapper library that operates at a higher level of abstraction. The core software library for accessing HDF5 files is written in C, but library interfaces for the HDF5 data model have been created for many programming languages, including Fortran, C++, Java, Python, Perl, Ada, and C#.

File Format

Objects in the HDF5 data model created by the library interface are stored in files whose structure is defined by the HDF5 file format. The HDF5 file format has many unique aspects, some of which are: a mechanism for storing non-HDF5 formatted data at the beginning of a file, a method of “micro-versioning” file data structures that makes incremental changes to the format possible, and data structures that enable constant-time lookup of data within the file in situations which previously required a logarithmic number of operations.

The HDF5 file format is designed to be flexible and extensible, allowing for evolution and expansion of the data model in an incremental and structured way. This allows new releases of the HDF5 software library to continue to access all previous versions of the HDF5 file format. This capability empowers application developers to create HDF5 files and access data contained within them over very long periods of time.

Tools

HDF5 is distributed with command-line utilities that can inspect and operate on HDF5 files. Operations provided by command-line utilities include copying HDF5

objects from one file to another, compacting internally fragmented HDF5 files to reduce their size, and comparing two HDF5 files to determine differences in the objects contained within them. The latter differencing utility, called “h5diff”, is also provided as a parallel computing application that uses the MPI programming interface to quickly compare two files using multiple processes.

Many other applications, both commercial and open source, can access data stored in HDF5 files. Some of these applications include MATLAB [8], Mathematica [9], HDFView [10], VisIt [11], and EnSight [12]. Some of these applications provide generic browsing and modification of HDF5 files, while others provide specialized visualization of domain-specific data models stored in HDF5 files.

Parallel File I/O

Applications that use the MPI programming interface [13] can use the HDF5 library to access HDF5 files in parallel from multiple concurrently executing processes. Internally, the HDF5 library uses the MPI interface for coordinating access to the HDF5 file as well as for performing parallel operations on the file. Efficiently accessing an HDF5 file in parallel requires storing the file on a parallel file system designed for access through the MPI interface.

Two methods of accessing an HDF5 file in parallel are possible: “independent” and “collective.” Independent parallel access to an HDF5 file is performed by a process in a parallel application without coordination with or cooperation from the other processes in the application. Collective parallel access to an HDF5 file is performed with all the processes in the parallel application cooperating and possibly communicating with each other.

The following discussion of HDF5 library capabilities describes parallel I/O features in the current release at the time this entry was written, release 1.8.6. The parallel I/O features in the HDF5 library are continually improving and evolving to address the ongoing changes in the landscape of parallel computing. Unless otherwise stated, limitations in the capabilities of the HDF5 library are not inherent to the HDF5 data model or file format and may be addressed in future library releases.

The HDF5 library requires that operations that create, modify, or delete objects in an HDF5 file be performed collectively. However, operations that only open

objects for reading can be performed independently. Requiring collective operations for modifying the file’s structure is currently necessary so that all processes in the parallel application keep a consistent view of the file’s contents.

Reading or writing the elements of a dataset can be performed independently or collectively. Accessing the elements of a dataset with independent or collective operations involves different trade-offs that application developers must balance.

Using independent operations requires a parallel application to create the overall structure of the HDF5 file at the beginning of its execution, or possibly with a nonparallel application prior to the start of the parallel application. The parallel application can then update elements of a dataset without requiring any synchronization or coordination between the processes in the application. However, the application must subdivide the portions of the file accessed from each process to avoid race conditions that would affect the contents of the file. Additionally, accessing a file independently may cause the underlying parallel file system to perform very poorly, due to its lack of a global perspective on the overall access pattern, which prevents the file system from taking advantage of many caching and buffering opportunities available with collective operations.

Accessing HDF5 dataset elements collectively requires that all processes in the parallel application cooperate when performing a read or write operation. Collective operations use the MPI interface within the HDF5 library to describe the region(s) of the file to access from each process, and then use collective MPI I/O operations to access the dataset elements. Collectively accessing HDF5 dataset elements allows the MPI implementation to communicate between processes in the application to determine the best method of accessing the parallel file system, which can greatly improve performance of the I/O operation. However, the communication and synchronization overhead of collective access can also slow down an application’s overall performance.

To get good performance when accessing an HDF5 dataset, it is important to choose a storage method that is compatible with the type of parallel access chosen. For example, compact data storage requires all writes to dataset elements be performed collectively, while external data storage requires all dataset element accesses be performed independently.

Collective and independent element access also involves the MPI and parallel file system layers, and those layers add their own complexities to the equation. HDF5 application developers must carefully balance the trade-offs of collective and independent operations to determine when and how to use them.

High performance access to HDF5 files is a strongly desired feature of application developers, and the HDF5 library has been designed with the goal of providing performance that closely matches the performance possible when an application accesses unformatted data directly. Considerable effort has been devoted to enhancing the HDF5 library's parallel performance, and this effort continues as new parallel I/O developments unfold.

Significant Parallel Applications and Libraries That Use HDF5

Many applications and software libraries that use HDF5 have become significant software assets, either commercially or as open source projects governed by a user community. HDF5's stability, longevity, and breadth of features have attracted many developers to it for storing their data, both for sequential and parallel computing purposes.

The first software library to use HDF5 was developed collaboratively by developers at the US Department of Energy's (DOE) Lawrence Livermore, Los Alamos and Sandia National Laboratories. This effort was called the "Sets and Fields" (SAF) library [14] and was developed to give the parallel applications that dealt with large, complex finite element simulations on the highest performing computers of the time a way to efficiently store their data.

Many large scientific communities worldwide have adopted HDF5 for storing data with parallel applications. Some significant examples include the FLASH software for simulating astrophysical nuclear flashes from the University of Chicago [15], the Chombo package for solving finite difference equations using adaptive mesh refinement from Lawrence Berkeley National Laboratory [16], and the open source NeXus software and data format for interchanging data in the neutron, x-ray, and muon science communities [17].

netCDF, PnetCDF, and HDF5

Another significant software library that uses HDF5 is the netCDF library [5], developed at the Unidata

Program Center for the University Corporation for Atmospheric Research (UCAR). Originally designed for the climate modeling community, netCDF has since been embraced by many other scientific communities. netCDF has adopted HDF5 as its principal storage method, as of version 4.0, in order to take advantage of several features in HDF5 that its previous file format did not provide, including data compression, a wider array of types for storing data elements, hierarchical grouping structures, and more flexible parallel operations.

Created prior to the development of netCDF-4, parallel-netCDF or "PnetCDF" [18] was developed by Argonne National Laboratory. PnetCDF allows parallel applications to access netCDF format files with collective data operations. PnetCDF does not extend the netCDF data model or format beyond allowing larger objects to be stored in netCDF files. Files written by PnetCDF and versions of the netCDF library prior to netCDF-4 do not use the HDF5 file format and instead store data in the "netCDF classic" format [19].

Future Directions

Both the primary development team at The HDF Group and the user community that has formed around the HDF5 project are constantly improving it. HDF5 continues to be ported to new computer systems and architectures and has its performance improved and errors corrected over time. Additionally, the HDF5 data model is expanding to encompass new developments in the field of high performance storage and computing.

Some improvements being designed or implemented as of this entry's writing include: increasing the efficiency of small file I/O operations through advanced caching mechanisms, finding ways to allow parallel applications to create HDF5 objects in a file with independent operations, and implementing new chunked data storage indexing methods to improve collective access performance.

Additionally, HDF5 continues to lead the scientific data storage field in its adoption of asynchronous file I/O for improved performance, journaled file updates for improved resiliency, and methods for improving concurrency by allowing different applications to read and write to the same HDF5 file without using a locking mechanism.

Related Entries

- ▶ [File Systems](#)
- ▶ [MPI \(Message Passing Interface\)](#)
- ▶ [NetCDF I/O Library, Parallel](#)

Bibliographic Notes and Further Reading

HDF5 has been under development since 1996, a short history of its development is recorded at The HDF Group web site [20].

Datasets in HDF5 files are analogous to sections of trivial fiber bundles [21], where the HDF5 dataspace corresponds to a fiber bundle's base space, the HDF5 datatype corresponds to the fiber, and the dataset, a variable whose value is the totality of the data elements stored, represents a section through the total space (which is the Cartesian product of the base space and the fiber).

HDF5 datatypes can be very complex and many more details are found in reference [22].

The HDF5 file format is documented in [23].

Many more applications and libraries use HDF5 than are discussed in this entry. A partial list can be found in [24].

Bibliography

1. HDF5, <http://www.hdfgroup.org/HDF5/>
2. National Center for Supercomputing Application at the University of Illinois at Urbana-Champaign, <http://www.ncsa.illinois.edu/>
3. The HDF group, <http://www.hdfgroup.org/>
4. HDF4, <http://www.hdfgroup.org/products/hdf4/>
5. netCDF, <http://www.unidata.ucar.edu/software/netcdf/>
6. TIFF, <http://partners.adobe.com/public/developer/tiff/index.html>
7. FITS, <http://fits.gsfc.nasa.gov/>
8. MATLAB, <http://www.mathworks.com/products/matlab/>
9. Mathematica, <http://www.wolfram.com/products/mathematica/index.html>
10. HDFView, <http://www.hdfgroup.org/hdf-java-html/hdfview/>
11. Visit, <https://wci.llnl.gov/codes/visit/>
12. EnSight, <http://www.ensight.com/>
13. MPI, <http://www.mpi-forum.org/>
14. Miller M et al (2001) Enabling interoperation of high performance, scientific computing applications: modeling scientific data with the sets & fields (SAF) modeling system. ICCS – 2001, May, part II, San Francisco. Lecture Notes in Computer Science, vol 2074. Springer, Heidelberg, pp 158–168
15. FLASH, <http://flash.uchicago.edu/web/>
16. Chombo, <https://seesar.lbl.gov/anag/chombo/index.html>

17. NeXus, http://www.nexusformat.org/Main_Page
18. PnetCDF, <http://trac.mcs.anl.gov/projects/parallel-netcdf>
19. netCDF file formats, <http://www.unidata.ucar.edu/software/netcdf/docs/netcdf/File-Format.html>
20. A history of the HDF group, <http://www.hdfgroup.org/about/history.html>
21. Fiber bundle definition, <http://mathworld.wolfram.com/FiberBundle.html>
22. HDF5 user guide, Chapter 6: datatypes, http://www.hdfgroup.org/HDF5/doc/UG/UG_framellDatatypes.html
23. HDF5 file format specification, <http://www.hdfgroup.org/HDF5/doc/H5.format.html>
24. Summary of software using HDF5, http://www.hdfgroup.org/products/hdf5_tools/SWSummaryByName.htm

H

HEP, Denelcor

- ▶ [Denelcor HEP](#)

Heterogeneous Element Processor

- ▶ [Denelcor HEP](#)

Hierarchical Data Format

- ▶ [HDF5](#)

High Performance Fortran (HPF)

- ▶ [HPF \(High Performance Fortran\)](#)

High-Level I/O Library

- ▶ [NetCDF I/O Library, Parallel](#)

High-Performance I/O

- ▶ [I/O](#)

Homology to Sequence Alignment, From

WU-CHUN FENG^{1,2}, HESHAN LIN¹

¹Virginia Tech, Blacksburg, VA, USA

²Wake Forest University, Winston-Salem, NC, USA

Discussion

Two sequences are considered to be *homologous* if they share a common ancestor. Sequences are either homologous or nonhomologous, but not in-between [13]. Determining whether two sequences are actually homologous can be a challenging task, as it requires inferences to be made between the sequences. Further complicating this task is the potential that the sequences may *appear* to be related via chance similarity rather than via common ancestry.

One approach toward determining homology entails the use of sequence-alignment algorithms that maximize the similarity between two sequences. For homology modeling, these alignments could be used to obtain the likely amino-acid correspondence between the sequences.

Introduction

Sequence alignment identifies similarities between a pair of biological sequences (i.e., pairwise sequence alignment) or across a set of multiple biological sequences (i.e., multiple sequence alignment). These alignments, in turn, enable the inference of functional, structural, and evolutionary relationships between sequences. For instance, sequence alignment helped biologists identify the similarities between the SARS virus and the more well-studied coronaviruses, thus enhancing the biologists' ability to combat the new virus.

Pairwise Sequence Alignment

There are two types of pairwise alignment: *global alignment* and *local alignment*. Global alignment seeks to align a pair of sequences entirely to each other, i.e., from one end to the other. As such, it is suitable for comparing sequences with roughly the same length, e.g., two closely homologous sequences. Local alignment seeks to identify significant matches between parts of the sequences. It is useful to analyze partially related

sequences, e.g., protein sequences that share a common domain.

Many approaches have been proposed for aligning a pair of sequences. Among them, dynamic programming is a common technique that can find optimal alignments between sequences. Dynamic programming can be used for both local alignment and global alignment. The algorithms in both cases are quite similar. The scoring model of an alignment algorithm is given by a substitution matrix and a gap-penalty function. A substitution matrix stores a matching score for every possible pair of letters. A matching score is typically measured by the frequency that a pair of letters occurs in the known homologous sequences according a certain statistical model. Popular substitution matrices include PAM and BLOSUM, an example of which is shown in Fig. 1. A gap-penalty function defines how gaps in the alignments are weighed in alignment scores. For instance, with a linear gap-penalty function, the penalty score grows linearly with the length of a gap. With an affine gap-penalty function, the penalty factors are differentiated for the opening and the extension of a gap.

The following discussion will focus on the basic algorithm and its parallelization of Smith–Waterman, a popular local-alignment tool based on dynamic programming. (For more advanced techniques to compute pairwise sequence alignment, the reader should consult the “Bibliographic Notes and Further Reading” section at the end of this entry.)

Case Study: Smith–Waterman Algorithm

Given two sequences $S_1 = a_1a_2\cdots a_m$ and $S_2 = b_1b_2\cdots b_n$, the Smith–Waterman algorithm uses an m by n scoring matrix H to calculate and track the alignments. A cell $H_{i,j}$ stores the highest similarity score that can be achieved by any possible alignment ending at a_i and b_j . The Smith–Waterman algorithm has three phases: *initialization*, *matrix filling*, and *traceback*.

The initialization phase simply assigns a value of 0 to each of the matrix cells in the first row and the first column. In the matrix-filling phase, the problem of aligning the two whole sequences is broken into smaller subproblems, i.e., aligning partial sequences. Accordingly, a cell $H_{i,j}$ is updated based on the values of its preceding neighbors. For the sake of illustration, the rest of the discussion assumes a linear gap-penalty function

H

Homology to Sequence Alignment, From: Fig. 1 BLOSUM62 substitution matrix

where the penalty of a gap is equal to a constant factor g (typically negative) times the length of the gap.

There are three possible alignment scenarios where $H_{i,j}$ is derived from its neighbors: (1) a_i and b_j are associated, (2) there is a gap in sequence S_1 , and (3) there is a gap in sequence S_2 . As such, the scoring matrix can be filled according to (1). The first three terms in (1) correspond to the three scenarios; the zero value ensures that there are no negative scores. $S(a_i, b_j)$ is the matching score derived by looking up the substitution matrix.

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + S(a_i, b_j) \\ H_{i-1,j} + g \\ H_{i,j-1} + g \\ 0 \end{cases} \quad (1)$$

When a cell is updated, the direction from which the maximum score is derived also needs to be stored (e.g., in a separate matrix). After the matrix is filled, a traceback process is used to recover the path of the best alignment. It starts from the cell with the highest score in the matrix and ends at a cell with a value of 0, following the direction information recorded earlier.

The majority of execution time is spent on the matrix-filling phase in Smith-Waterman. Algorithm 1 shows a straightforward implementation of the matrix

Algorithm 1 Matrix Filling in Smith–Waterman

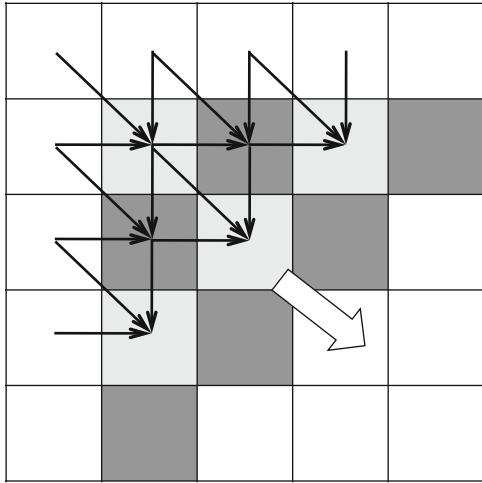
```

for  $i = 1$  to  $m$  do
  for  $j = 1$  to  $n$  do
     $max = H_{i-1,j-1} + S(a_i, b_j) > H_{i-1,j} + g ? H_{i-1,j-1} +$ 
     $S(a_i, b_j) : H_{i-1,j} + g$ 
    if  $H_{i,j-1} + g > max$  then
       $H_{i,j} = H_{i,j-1} + g$ 
    else
       $H_{i,j} = max$ 
    end if
  end for
end for

```

filling. In the inner loop of the algorithm, the cell calculated in one iteration depends on the value updated in the previous iteration, resulting in a “read-after-write” hazard (see [14] for details), which can reduce the instruction-level parallelism that can be exploited by pipelining, and hence, adversely impact performance. In addition, this algorithm is difficult to directly parallelize because of the data dependency between iterations in the inner loop.

As depicted in Fig. 2, the calculation of a particular cell depends on its west, northwest, and north neighbors. However, the updates of individual cells along an anti-diagonal are independent. This observation

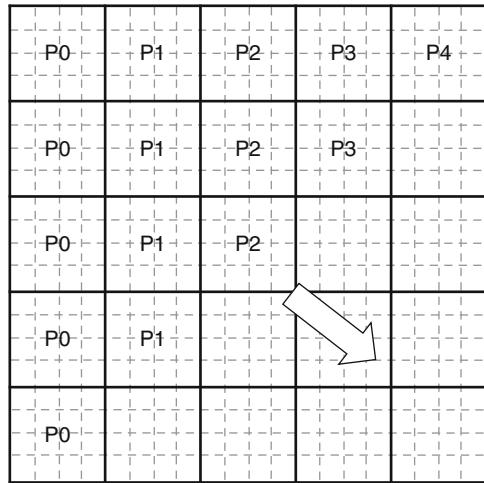


Homology to Sequence Alignment, From. Fig. 2 Data dependency of matrix filling

motivates a waveform-filling algorithm [11], where the matrix cells on an anti-diagonal can be updated simultaneously. That is, because there is no dependency between two adjacent cells along an anti-diagonal, this algorithm greatly reduces read-after-write hazards, and in turn, increases the execution efficiency and ease of parallelization. For example, in a shared-memory environment, individual threads can compute a subset of cells along an anti-diagonal. However, synchronization between threads must occur after computing each anti-diagonal.

Since a scoring matrix is typically stored in “row major” order, as shown in Algorithm 1, the above waveform algorithm may have a large memory footprint when computing an anti-diagonal, thus limiting the benefits of processor caches. One improvement entails partitioning the matrix into tiles and having each parallel processing unit fill a subset of the tiles, as shown in Fig. 3. By carefully choosing the tile size, data processed by a thread can fit in the processor cache. Furthermore, when parallelized in distributed environments, the tiled approach can effectively reduce internode communication, as compared to the fine-grained waveform approach, because only elements at the borders of individual tiles need to be exchanged between different compute nodes.

With the waveform approach, the initial amount of parallelism in the algorithm is low. It gradually increases



Homology to Sequence Alignment, From. Fig. 3 Tiled implementation

along each successive anti-diagonal until reaching the maximum parallelism along the longest anti-diagonal, and then monotonically decreases thereafter. A trade-off needs to be made in choosing the tile size. If the tile size is too large, there is not sufficient parallelism to exploit at the beginning of the waveform computation, which results in idle resources in systems with a large number processing units. On the other hand, too small a tile size will incur much more synchronization and communication overhead. Nonetheless, the waveform approach may generate imbalanced workloads on different processors, especially at the beginning and end of the computation. It is worth noting that there is an alternative parallel algorithm that uses prefix-sum to compute the scoring matrix row by row (or column by column) [4], which can generate uniform task distribution among all processors.

The above discussion assumes a simple linear gap-penalty function. In practice, the Smith-Waterman algorithm uses an affine gap-penalty scheme, which requires maintaining three scoring matrices in order to track the gap opening and extension. Consequently, both the time and space usages increase by a factor of three in implementation using affine gap penalties.

Sequence Database Search

With the proliferation of public sequence data, sequence database search has become an important task in sequence analysis. For example, newly discovered

sequences are typically searched against a database of sequences with known genes and known functions in order to help predict the functions of the newly discovered sequences. A sequence database-search tool compares a set of query sequences against all sequences in a database with a pairwise alignment algorithm and reports the matches that are statistically significant. Although dynamic-programming algorithms can be used for sequence database search, the algorithms are too computationally demanding to keep up with the database growth. Consequently, heuristic-based algorithms, such as BLAST [2, 3] and FASTA [21], have been developed for rapidly identifying similarities in sequence databases.

BLAST is the most widely used sequence database-search tool. It reduces the complexity of alignment computation by filtering potential matches with common words, called k -mers. Specifically, there are four stages in comparing a query sequence and a database sequence.

- Stage 1: The query and the database sequences are parsed into words of length k (k is 3 for protein sequences and 11 for DNA sequences by default). The algorithm then matches words between the query sequence and the database sequence and calculates an alignment score for each matched word, based on a substitution matrix (e.g., BLOSUM62). Only matched words with alignment scores higher than a threshold are kept for the next stage.
- Stage 2: For a high-scoring matched word, ungapped alignment is performed by extending the matched word in both directions. An alignment score will be calculated along the extension. The extension stops when the alignment score stops increasing and slightly drops off from the maximum alignment score (controlled by another threshold).
- Stage 3: Ungapped alignments with scores larger than a given threshold obtained from stage 2 are chosen as seed alignments. Gapped alignments are then performed on the seed alignments using a dynamic-programming algorithm, following both forward and backward directions.
- Stage 4: Traceback is performed to recover the paths of gapped alignments.

BLAST calculates the significance of result alignments using Karlin-Altschul statistics. The Karlin-Altschul

theory uses a statistic called the e-value (E) to measure the likelihood that an alignment is resulted from matches by chance (i.e., matches between random sequences) as compared to true homologous relationships. The e-value can be calculated according to (2):

$$E = Kmn \cdot e^{-\lambda S} \quad (2)$$

where K and λ are the Karlin-Altschul parameters, m and n are the query length and the total length of all database sequences, and S is the alignment score. The e-value indicates how many alignments with a score higher than S can be found by chance in the search space, i.e., the multiple of the query sequence length and the database length. The lower the e-value, the more significant is an alignment. The alignment results of a query sequence are sorted in the order of e-value.

A sequence database-search job needs to compute $M \times N$ pairwise sequence alignments, where M and N are the numbers of the query and database sequences, respectively. This computation can be parallelized with a coarse-grained approach, where the alignments of individual pairs of sequences are assigned to different processing units.

Early parallel sequence-search software adopted a *query segmentation* approach, where a sequence-search job is parallelized by having individual compute nodes concurrently search disjoint subsets of query sequences against the whole sequence database. Since the searches of individual query sequences are independent, this *embarrassingly parallel* approach is easy to implement and scales well. However, the size of sequence databases is growing much faster than the memory size of a typical single computer. When the database cannot fit in memory, data will be frequently swapped in and out of the memory when searching multiple queries, thus causing significant performance degradation, because disk I/O is several orders of magnitude slower than memory access. Query segmentation can improve the search throughput, but it does not reduce the response time taken to search a single query sequence.

Database segmentation is an alternative parallelization approach, where large databases are partitioned and cached in the aggregate memory of a group of compute nodes. By doing so, the repeated I/O overhead of searching large databases is avoided. Database segmentation also improves the search response time since a

compute node searches only a portion of the database. However, database segmentation introduces computational dependencies between individual nodes because the distributed results generated at different nodes need to be merged and sorted to produce the final output. The parallel overhead of merging and sorting increases as the system size grows.

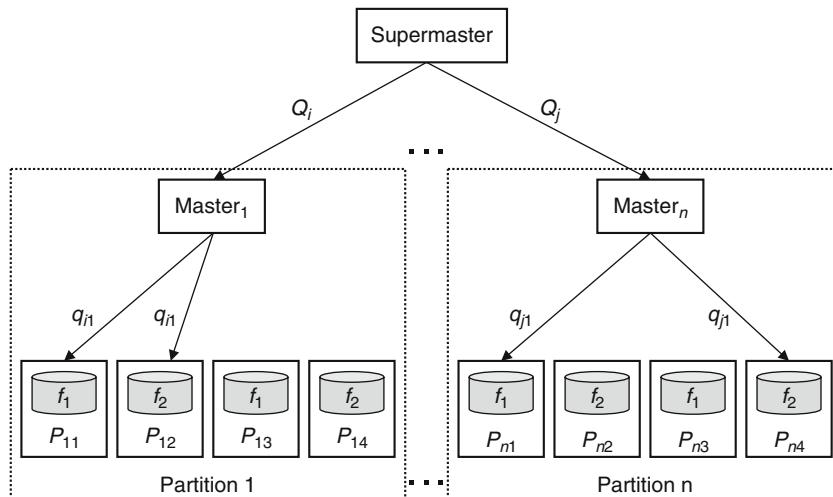
With the astronomical growth of sequence databases, today's large-scale sequence search jobs can be very resource demanding. For instance, BLAST searching in a metagenomics project can consume several millions of processor hours. Massively parallel sequence-search tools, such as mpiBLAST [6, 19, 20] and ScalaBLAST [33], have been developed to accelerate large-scale sequence search jobs on state-of-the-art supercomputers. These tools use a combination of query segmentation and database segmentation to offer massive parallelism needed to scale on a large number of processors.

Case Study: mpiBLAST

mpiBLAST is an open-source parallelization of NCBI BLAST that has been designed for petascale deployment. Adopting a scalable, hierarchical design, mpiBLAST parallelizes a search job via a combination of query segmentation and database segmentation. As shown in Fig. 4, processors in the system are organized into

equal-sized *partitions*, which are supervised by a dedicated *supermaster* process. The supermaster is responsible for assigning tasks to different partitions and handling inter-partition load balancing. Within each partition, there is one *master* process and many *worker* processes. The master is responsible for coordinating both computation and I/O scheduling in a partition. The master periodically fetches a subset of query sequences from the supermaster and assigns them to workers, and it coordinates output processing of queries that have been processed in the partition. The sequence database is partitioned into fragments and replicated to workers in the system. This hierarchical design avoids creating scheduling bottlenecks in large systems by distributing scheduling workloads to multiple masters.

Large-scale sequence searches can be highly data-intensive, and as such, the efficiency of data management is critical to the program scalability. For the input data, having thousands of processors simultaneously load database fragments from shared storage may overwhelm the I/O subsystem. To address this, mpiBLAST designates a set of compute nodes as I/O proxies, which read database fragments from the file system in parallel and replicate them to other workers using the broadcasting mechanism in MPI [28, 29] libraries. In addition, mpiBLAST allows workers to cache



Homology to Sequence Alignment, From. Fig. 4 mpiBLAST hierarchical design. Q_i and Q_j are query batches fetched from the supermaster to masters, and q_{i1} and q_{j1} are query sequences that are assigned by masters to their workers. In this example, the database is segmented into two fragments f_1 and f_2 and replicated twice within each partition

assigned database fragments in the memory or local storage, and it uses a task-scheduling algorithm that takes into account data locality to minimize repeated loading of database fragments.

With database segmentation, result alignments of different database fragments are usually interleaved in the global output because those alignments need to be sorted by e-values. Consequently, the output data generated at each worker is noncontiguous in the output file. Straightforward noncontiguous I/O with many seek-and-write operations is slow on most file systems. This type of I/O can be optimized with collective I/O [27, 44, 46], which is available in parallel I/O libraries such as ROMIO [45]. Collective I/O uses a two-phase process. In the first phase, involved processes exchange data with each other to form large trunks of contiguous data, which are stored as memory buffers in individual processes. In the second phase, the buffered data is written to the actual file system. Collective I/O improves I/O performance because continuous data accesses are much more efficient than noncontiguous ones. Traditional collective I/O implementations require synchronization between all involved processes for each I/O operation. This synchronization overhead will adversely impact sequence-search performance when computation is

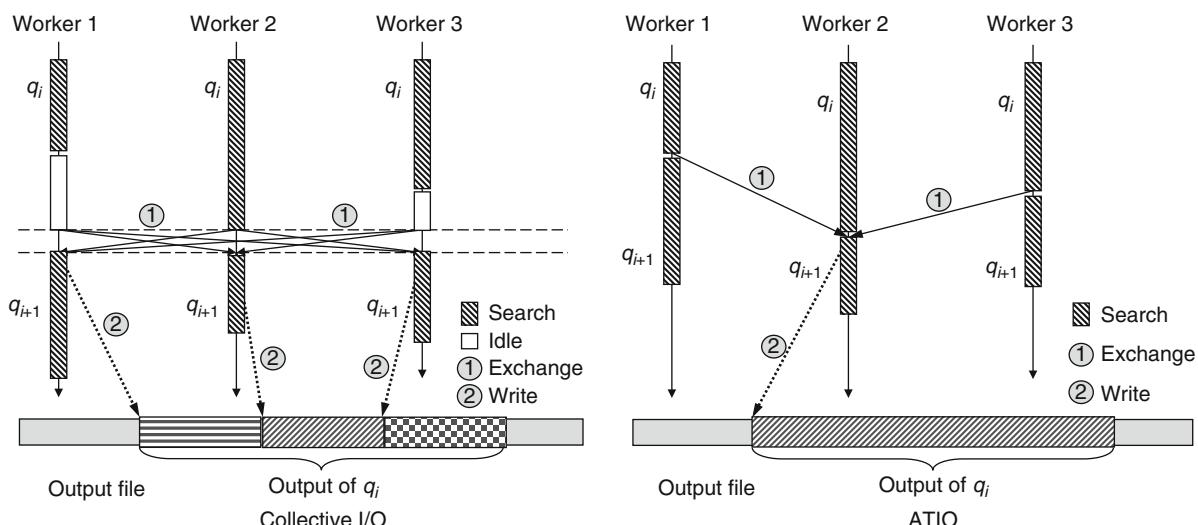
imbalanced across different processes. To address this issue, mpiBLAST introduces a parallel I/O technique called asynchronous, two-phase I/O (ATIO), which allows worker processes to rearrange I/O data without synchronizing with each other. Specifically, mpiBLAST appoints a worker as the write leader for each query sequence. The write leader aggregates output data from other workers via nonblocking MPI communication and carries out the write operation to the file system. ATIO overlaps I/O reorganization and sequence-search computation, thus improving the overall application performance. Figure 5 shows the difference between collective I/O and ATIO within the context of mpiBLAST.

H

Multiple Sequence Alignment

Multiple sequence alignment (MSA) identifies similarities among three or more sequences. It can be used to analyze a family of related sequences to reveal phylogenetic relationships. Other usages of multiple sequence alignment include detection of conserved biological features and genome sequencing. Multiple sequence alignment can also be global or local.

Like pairwise sequence alignment, multiple sequence alignment can be computed using *dynamic*



Homology to Sequence Alignment, From. Fig. 5 Collective I/O and ATIO. Collective I/O requires synchronization and introduces idle waiting in worker processes. ATIO uses a write leader to aggregate noncontiguous data from other workers in an asynchronous manner

programming algorithms. Although these algorithms can find optimal alignments, the required resources for these algorithms grow exponentially as the number of sequences increases. Suppose there are N sequences and the average sequence length is L , the time and space complexities are both $O(L^N)$. Thus, it is computationally impractical to use dynamic programming to align a large number of sequences.

Many heuristic approaches have been proposed to reduce the computational complexity of multiple sequence alignment. *Progressive alignment methods* [8, 12, 32, 35, 47] use guided pairwise sequence alignment to rapidly construct MSA for a large number of sequences. These methods first build a phylogenetic tree based on all-to-all pairwise sequence alignments using neighbor joining [40] or UPGMA [43] techniques. Guided by the phylogenetic tree, the most similar sequences are first aligned, the less similar sequences are then progressively added to the initial MSA. One problem with progressive methods is that errors that occur in early aligning stages are propagated to the final alignment results. *Iterative alignment methods* [10, 49] address this problem by introducing “correction” mechanisms during the MSA construction process. These methods incrementally align sequences as progressive methods, but continuously adjust the structure of the phylogenetic tree and previously computed alignments according to a certain objective function.

Case Study: ClustalW

ClustalW [47] is a widely used MSA program based on progressive methods. The ClustalW algorithm includes three stages: (1) distance matrix computation, (2) guided tree construction, and (3) progressive alignment. Due to the popularity of ClustalW, its parallelization has been well studied on clusters [9, 17] and multiprocessor systems [5, 30].

In the first stage, ClustalW computes a distance matrix by performing all-to-all pairwise alignment over input sequences. This requires a total of $\frac{N(N-1)}{2}$ comparison for N sequences since the alignment of a pair of sequences is symmetric. ClustalW allows users to choose between two alignment algorithms: a faster k -mer matching algorithm and a slower but more accurate dynamic programming algorithm. This stage of the algorithm is embarrassingly parallel. Alignments of individual pairs of sequences can be statically assigned

or dynamically assigned with a greedy algorithm to different processing units. Additional parallelism can be exploited by parallelizing the alignment of a single pair of sequences, similar to the Smith–Waterman algorithm.

In the second stage, a guided tree is constructed using a neighbor-joining algorithm based on the alignment scores in the distance matrix. Initially all sequences are leaf nodes of a star-like tree. The algorithm iteratively selects a pair of nodes and joins them into a new internal node until there are no nodes left to join, at which point a bifurcated tree is created. Algorithm 2 gives the pseudocode of this process. Suppose there are a total of M nodes at an iteration, there are $\frac{n(n-1)}{2}$ possible pairs of nodes. The pair of nodes that results in the smallest length of branches will be selected to join. An example of neighbor-joining process with four sequences is given in Fig. 6.

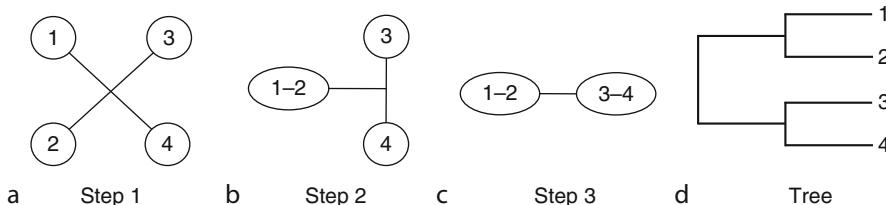
In Algorithm 2, the outermost loop cannot be parallelized because one iteration of neighbor joining is dependent on the previous one. However, within an iteration of neighbor joining, the calculation of branch lengths of all pairs of nodes can be performed in parallel. Since the number of available nodes reduces after each joining operation, the parallelism level decreases as the iteration advances in the outermost loop.

Algorithm 2 Construction of Guided Tree

```

while there are nodes to join do
    Let  $n$  be the number of current available nodes
    Let  $D$  be the current distance matrix
    Let  $L_{min}$  be the smallest length of the tree branches
    for  $i = 2$  to  $n$  do
        for  $j = 1$  to  $i - 1$  do
             $L(i,j) = (n - 2)D_{i,j} - \sum_{k=1}^n D_{i,k} - \sum_{k=1}^n D_{j,k}$ 
            if  $L(i,j) < L_{min}$  then
                 $L_{min} = L(i,j)$ 
            end if
        end for
    end for
    Combine nodes  $i, j$  that result in  $L_{min}$  into a new node
    Update distance matrix with the new node
end while

```



Homology to Sequence Alignment, From. Fig. 6 Neighbor-joining example

In the third stage, the sequence is progressively aligned according to the guided tree. For instance, in the tree shown in Fig. 6(d), sequences 1 and 2 are aligned first, followed by 3 and 4. Finally, the alignment results of $<1-2>$ and $<3-4>$ are aligned. Note that the alignment on a node can only be performed after both of its children are aligned, but the alignments at the same level of the tree can be performed simultaneously. The level of parallelism of the algorithm depends heavily on the tree structure. In the best case, the guided tree is a balanced tree. At the beginning, all the leaves can be aligned in parallel. The number of concurrent alignments decreases by a half at each higher level toward the root of the tree.

Case Study: T-Coffee

T-Coffee [32] is another popular progressive alignment algorithm. Compared to other typical progressive alignment tools, T-Coffee improves the alignment accuracy by adopting a consistency-based scoring function, which uses similarity information among all input sequences to guide the alignment progress. T-Coffee consists of two main steps: *library generation* and *progressive alignment*. The first step constructs a library that contains a mixture of global and local alignments between *every pair* of input sequences. In the library, an alignment is represented as a list of pairwise constraints. Each constraint stores a pair of matched residues along with an alignment weight, which is essentially a three-tuple $<S_{xi}, S_{yj}, w>$, where S_{xi} is the i th residue of sequence S_x and w is the weight. T-Coffee incorporates pairwise global alignments generated by ClustalW as well as top ten nonintersecting local alignments reported by Lalign from the FASTA package [21]. T-Coffee can also take alignment information from other MSA software. The pairwise constraints of a same pair of matched residues from various sources (e.g., global and local alignments) will be combined to

remove duplication. Constructing the library requires $\frac{N(N-1)}{2}$ global and local alignments and thus is highly compute-intensive.

After all pairwise alignments are incorporated in the library, T-Coffee performs *library extension*, a procedure that incorporates transitive alignment information to the weighing of pairwise constraints. Basically, for a pair of sequences x and y , if there is a sequence z that aligns to both x and y , then constraints between x and y will be reweighed by combining the weights of the corresponding constraints between x and z as well as y and z . Suppose the average sequence length is L , since for each pair of sequences, the extension algorithm needs to scan the rest of $N - 2$ sequences, and there are at most L constraints between a pair of sequences, the worst computation complexity of library extension is $O(N^3L^2)$.

In the second step, T-Coffee performs progressive alignment guided by a phylogenetic tree built with the neighbor-joining method, similar to the ClustalW algorithm. However, T-Coffee uses the weights in the extended library to align residues when grouping sequences/alignments. Since those weights bear complete alignment information from all input sequences, the progressive alignment in T-Coffee can reduce errors caused by the greediness of classic progressive methods.

Parallel T-Coffee (PTC) is a parallel implementation of T-Coffee in cluster environments [53]. PTC adopts a master-worker architecture and uses MPI to communicate between different processes. As mentioned earlier, the library generation in T-Coffee needs to compute all-to-all global and local alignments. The computation tasks of these alignments are independent of each other. PTC uses guided self-scheduling (GSS) [36] to distribute alignment tasks to different worker nodes. GSS first assigns a portion of the tasks to the workers. Each worker monitors its performance when processing the initial assignments; this performance information is

then sent to the master and used for dynamic scheduling for subsequent assignments.

After pairwise alignments are finished, duplicated constraints generated on distributed workers need to be combined. PTC implements this with parallel sorting. Each constraint is assigned to a bucket resident at a worker. Each worker can then concurrently combine duplicated constraints within its own bucket. The constraints in the library are then transformed into a three-dimensional lookup table, with rows and columns indexed by sequences and residues, respectively. Each element in the lookup table stores all constraints for a residue of a sequence. The lookup table will be accessed by all processors during the progressive alignment. PTC evenly distributes the lookup table by rows to all processors and allows table entries to be accessed by other processors through one-sided remote memory access. An efficient caching mechanism is also implemented to improve lookup performance.

During the progressive alignment, PTC schedules tree nodes to a processor according to their readiness; a tree node that has a fewer number of unprocessed child nodes has a higher scheduling priority. For tree nodes that have all child nodes processed, PTC gives higher priority to the ones with shorter estimated execution time. Similar to ClustalW, the parallelism of progress alignment in PTC can be limited if the guided tree is unbalanced. To address this issue, Orobital et al. proposed a heuristic approach that can construct a more balanced guided tree by allowing a pair of nodes to be grouped if their similarity value is smaller than the average similarity value between all sequences in the distance matrix [34].

Related Entries

- Bioinformatics
- Genome Assembly

Bibliographic Notes and Further Reading

As discussed in the case study of Smith-Waterman, typical sequence-alignment algorithms require $O(mn)$ space and time, where m and n are the lengths of compared sequences. Such a space requirement can be impractical for computing alignments of large sequences (e.g., those with a length of multiple

megabytes) on commodity machines. To address this issue, Mayers and Miller introduced a space-efficient alignment algorithm [31] adapted from the Hirschberg technique [15], which was originally developed for finding the longest common subsequence between two strings. By recursively dividing the alignment problem into subproblems at a “midpoint” along the *middle column* of the scoring matrix, Mayers and Miller’s approach can find the optimal alignment within $O(m + n)$ space but still $O(mn)$ time. Huang showed that a straightforward parallelization of the Hirschberg algorithm would require more than linear aggregate space [16], i.e., each processor needed to store more than $O\left(\frac{m+n}{p}\right)$ data, where p is the number of concurrent processors. In turn, Huang proposed an improved algorithm that recursively divides the alignment problem at the midpoint along the *middle anti-diagonal* of the scoring matrix. By doing so, Huang’s algorithm required only $O\left(\frac{mn}{p}\right)$ space per processor but with an increased time complexity of $O\left(\frac{(m+n)^2}{p}\right)$. Aluru et al. presented an alternative space-efficient parallel algorithm [4] that is more time efficient ($O\left(\frac{mn}{p}\right)$) but also consumes more space ($O\left(m + \frac{n}{p}\right), m \leq n$) than Huang’s approach. In their approach, an $O(mn)$ algorithm is first used to partition the scoring matrix into p vertical slices, and the last column of each slice as well as its intersection with the optimal alignment is stored. Each processor then takes a slice and uses a Hirschberg-based algorithm to compute the optimal alignment within the slice. In a subsequent study, Aluru et al. proposed an improved parallel algorithm [37] that requires $O\left(\frac{mn}{p}\right)$ time and $O\left(\frac{m+n}{p}\right)$ space, when $p = O\left(\frac{n}{\log n}\right)$ processors are used. In other words, such a parallel algorithm achieves “optimal” time and space complexities because this algorithm delivers a linear speedup with respect to the best known sequential algorithm.

The computational intensity of sequence-alignment algorithms has motivated studies in parallelizing these algorithms on accelerators. Various algorithms have been accelerated using the SIMD instruction extensions of commodity processors [38, 51], field-programmable gate array (FPGA) [18, 26], Cell Broadband Engine [1, 39, 42], and graphics processing units (GPUs) [22–24, 41, 48, 51, 52]. Developing and optimizing applications on traditional accelerators is much more

difficult than on CPUs, which may partially explain why accelerator-based solutions have not been widely adopted even if these solutions have demonstrated very promising performance results. However, the continuing improvement of software environments on commodity GPUs has made them increasingly popular for accelerating sequent alignments. To cope with the astronomical growth of sequence data, cloud-based solutions [7, 25] have also been developed to enable users to tackle large-scale problems with elastic compute resources from public clouds such as Amazon EC2.

Bibliography

1. Aji AM, Feng W, Blagojevic F, Nikolopoulos DS (2008) Cell-SWat: modeling and scheduling waveform computations on the cell broadband engine. In: CF '08: Proceedings of the 5th conference on computing frontiers. ACM, New York, pp 13–22
2. Altschul S, Gish W, Miller W, Myers E, Lipman D (1990) Basic local alignment search tool. *J Mol Biol* 215(3):403–410
3. Altschul S, Madden T, Schiffer A, Zhang J, Zhang Z, Miller W, Lipman D (1997) Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res* 25(17):3389–3402
4. Aluru S, Futamura N, Mehrotra K (2003) Parallel biological sequence comparison using prefix computations. *J Parallel Distrib Comput* 63:264–272
5. Chaichoompu K, Kittitornkun S, Tongsim S (2006) MT-ClustalW: multithreading multiple sequence alignment. In: International parallel and distributed processing symposium. Rhodes Island, Greece, p 280
6. Darling A, Carey L, Feng W (2003) The design, implementation, and evaluation of mpiBLAST. In: Proceedings of the Cluster-World conference and Expo, in conjunction with the 4th international conference on Linux clusters: The HPC revolution 2003, San Jose
7. Di Tommaso P, Orobio M, Guirado F, Cores F, Espinosa T, Notredame C (2010) Cloud-Coffee: implementation of a parallel consistency-based multiple alignment algorithm in the T-Coffee package and its benchmarking on the Amazon Elastic-Cloud. *Bioinformatics* 26(15):1903–1904
8. Do CB, Mahabhashyam MS, Brudno M, Batzoglou S (2005) Prob-Cons: probabilistic consistency-based multiple sequence alignment. *Genome Res* 15(2):330–340
9. Ebedes J, Datta A (2004) Multiple sequence alignment in parallel on a workstation cluster. *Bioinformatics* 20(7):1193–1195
10. Edgar R (2004) MUSCLE: a multiple sequence alignment method with reduced time and space complexity. *BMC Bioinformatics* 5(1):113
11. Edmiston EE, Core NG, Saltz JH, Smith RM (1989) Parallel processing of biological sequence comparison algorithms. *Int J Parallel Program* 17:259–275
12. Feng DF, Doolittle RF (1987) Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *J Mol Evol* 25(4):351–360
13. Fitch W, Smith T (1983) Optimal sequences alignments. *Proc Natl Acad Sci* 80:1382–1386
14. Hennessy JL, Patterson DA (2006) Computer architecture: a quantitative approach, 4th edn. Morgan Kaufmann Publishers, San Francisco
15. Hirschberg DS (1975) A linear space algorithm for computing maximal common subsequences. *Commun ACM* 18: 341–343
16. Huang X (1990) A space-efficient parallel sequence comparison algorithm for a message-passing multiprocessor. *Int J Parallel Program* 18:223–239
17. Li K (2003) W-MPI: ClustalW analysis using distributed and parallel computing. *Bioinformatics* 19(12):1585–1586
18. Li I, Shum W, Truong K (2007) 160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA). *BMC Bioinformatics* 8(1):185
19. Lin H, Ma X, Chandramohan P, Geist A, Samatova N (2005) Efficient data access for parallel BLAST. In: Proceedings of the 19th IEEE international parallel and distributed processing symposium (IPDPS'05). IEEE Computer Society, Los Alamitos
20. Lin H, Ma X, Feng W, Samatova NF (2010) Coordinating computation and I/O in massively parallel sequence search. *IEEE Trans Parallel Distrib Syst* 99:529–543
21. Lipman D, Pearson W (1988) Improved toolsW, HT for biological sequence comparison. *Proc Natl Acad Sci* 85(8):2444–2448
22. Liu W, Schmidt B, Voss B, Müller-Wittig W (2006) GPU-ClustalW: using graphics hardware to accelerate multiple sequence alignment, Chapter 37 In: Robert Y, Parashar M, Badrinath R, Prasanna VK (eds) High performance computing – HiPC 2006. Lecture notes in computer science, vol 4297. Springer, Berlin/Heidelberg, pp 363–374
23. Liu Y, Maskell D, Schmidt B (2009) CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Res Notes* 2(1):73
24. Liu Y, Schmidt B, Maskell DL (2009) MSA-CUDA: multiple sequence alignment on graphics processing units with CUDA. In: ASAP '09: Proceedings of the 2009 20th IEEE international conference on application-specific systems, architectures and processors, Washington, DC. IEEE Computer Society, Los Alamitos, California, USA, pp 121–128
25. Lu W, Jackson J, Barga R (2010) AzureBlast: a case study of cloud computing for science applications. In: 1st workshop on scientific cloud computing, co-located with ACM HPDC 2010 (High performance distributed computing). Chicago, Illinois, USA
26. Mahram A, Herbordt MC (2010) Fast and accurate NCBI BLASTP: acceleration with multiphase FPGA-based prefiltering. In: Proceedings of the 24th ACM international conference on supercomputing. Tsukuba, Ibaraki, Japan
27. May J (2001) Parallel I/O for high performance computing. Morgan Kaufmann Publishers, San Francisco
28. Message Passing Interface Forum (1995) MPI: message-passing interface standard

29. Message Passing Interface Forum (1977) MPI-2 extensions to the message-passing standard
30. Mikhailov D, Cofer H, Gomperts R (2001) Performance optimization of Clustal W: parallel Clustal W, HT Clustal, and MULTICLUSTAL. White Papers, Silicon Graphics, Mountain View
31. Myers EW, Miller W (1988) Optimal alignments in linear space. *Comput Appl Biosci (CABIOS)* 4(1):11–17
32. Notredame C (2000) T-coffee: a novel method for fast and accurate multiple sequence alignment. *J Mol Biol* 302(1):205–217
33. Oehmen C, Nieplocha J (2006) ScalaBLAST: a scalable implementation of BLAST for high-performance data-intensive bioinformatics analysis. *IEEE Trans Parallel Distrib Syst* 17(8):740–749
34. Oorbit M, Guirado F, Notredame C, Cores F (2009) Exploiting parallelism on progressive alignment methods. *J Supercomput* 1–9. doi: 10.1007/s11227-009-0359-5
35. Pei J, Sadreyev R, Grishin NV (2003) PCMA: fast and accurate multiple sequence alignment based on profile consistency. *Bioinformatics* 19(3):427–428
36. Polychronopoulos CD, Kuck DJ (1987) Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. *IEEE Trans Comput* 36:1425–1439
37. Rajko S, Aluru S (2004) Space and time optimal parallel sequence alignments. *IEEE Trans Parallel Distrib Syst* 15:1070–1081
38. Rognes T, Seeberg E (2000) Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics* 16(8):699–706
39. Sachdeva V, Kistler M, Speight E, Tzeng TK (2008) Exploring the viability of the cell broadband engine for bioinformatics applications. *Parallel Comput* 34(11):616–626
40. Saitou N, Nei M (1987) The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Mol Biol Evol* 4(4):406–425
41. Sandes EFO, de Melo ACMA (2010) CUDAAlign: using GPU to accelerate the comparison of megabase genomic sequences. *SIGPLAN Not* 45(5):137–146
42. Sarje A, Aluru S (2009) Parallel genomic alignments on the cell broadband engine. *IEEE Trans Parallel Distrib Syst* 20(11): 1600–1610
43. Sneath PH, Sokal RR (1962) Numerical taxonomy. *Nature* 193:855–860
44. Thakur R, Choudhary A (1996) An extended two-phase method for accessing sections of out-of-core arrays. *Sci Program* 5(4): 301–317
45. Thakur R, Gropp W, Lusk W (1999) Data sieving and collective I/O in ROMIO. In: Symposium on the frontiers of massively parallel processing. Annapolis, Maryland, USA, p 182
46. Thakur R, Gropp W, Lusk E (1999) On implementing MPI-IO portably and with high performance. In: Proceedings of the sixth workshop on I/O in parallel and distributed systems. Atlanta, Georgia, USA
47. Thompson JD, Higgins DG, Gibson TJ (1994) CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Res* 22(22):4673–4680
48. Vouzis PD, Sahinidis NV (2011) GPU-BLAST: using graphics processors to accelerate protein sequence alignment. *Bioinformatics* 27(2):182–188
49. Wallace IM, Orla O, Higgins DG (2005) Evaluation of iterative alignment algorithms for multiple alignment. *Bioinformatics* 21(8):1408–1414
50. Wozniak A (1997) Using video-oriented instructions to speed up sequence comparison. *Comput Appl Biosci* 13(2):145–150
51. Xiao S, Aji AM, Feng W (2009) On the robust mapping of dynamic programming onto a graphics processing unit. In: ICPADS '09: proceedings of the 2009 15th international conference on parallel and distributed systems, Washington, DC. IEEE Computer Society, Los Alamitos, California, USA, pp 26–33
52. Xiao S, Lin H, Feng W (2011) Characterizing and optimizing protein sequence search on the GPU. In: Proceedings of the 19th IEEE international parallel and distributed processing symposium Anchorage, Alaska. IEEE Computer Society, Los Alamitos, California, USA
53. Zola J, Yang X, Rospondek A, Aluru S (2007) Parallel-TCoffee: a parallel multiple sequence aligner. In: ISCA international conference on parallel and distributed computing systems (ISCA PDCS 2007), pp 248–253

Horizon

► [Tera MTA](#)

HPC Challenge Benchmark

JACK DONGARRA, PIOTR LUSZCZEK
University of Tennessee, Knoxville, TN, USA

Definition

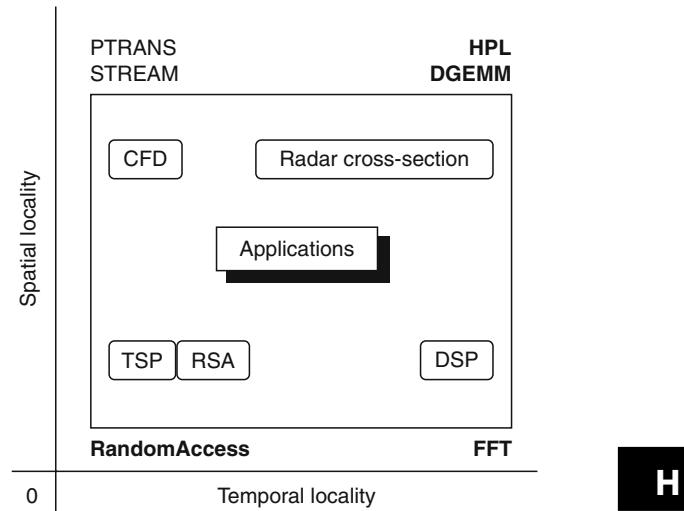
HPC Challenge (HPCC) is a benchmark that measures computer performance on various computational kernels that span the memory access locality space. HPCC includes tests that are able to take advantage of nearly all available floating point performance: High Performance LINPACK and matrix-matrix multiply allow for data reuse that is only bound by the size of large register file and fast cache. The twofold benefit from these tests is the ability to answer the question how well the hardware is able to work around the

“memory wall” and how today’s machines compare to the systems of the past as they are cataloged by the LINPACK Benchmark Report [3] and TOP500. HPCC also includes other tests, STREAM, PTRANS, FFT, RandomAccess – when they are combined together they span the memory access locality space. They are able to reveal the growing inefficiencies of the memory subsystem and how they are addressed in the new computer infrastructures. HPCC also offers scientific rigor to the benchmarking effort. The tests stress double precision floating point accuracy: the absolute prerequisite in the scientific world. In addition, the tests include careful verification of the outputs – undoubtedly an important fault-detection feature at extreme computing scales.

Discussion

The HPC Challenge benchmark suite was initially developed for the DARPA HPCS program [6] to provide a set of standardized hardware probes based on commonly occurring computational software kernels. The HPCS program involves a fundamental reassessment of how to define and measure performance, programmability, portability, robustness and, ultimately, productivity across the entire high-end domain. Consequently, the HPCC suite aimed both to give conceptual expression to the underlying computations used in this domain, and to be applicable to a broad spectrum of computational science fields. Clearly, a number of compromises needed to be embodied in the current form of the suite, given such a broad scope of design requirements. HPCC was designed to provide approximate bounds on computations that can be characterized by either high or low spatial and temporal locality (see Fig. 1, which gives the conceptual design space for the HPCC component tests). In addition, because the HPCC tests consist of simple mathematical operations, HPCC provides a unique opportunity to look at language and parallel programming model issues. As such, the benchmark is designed to serve both the system user and designer communities.

Figure 2 shows a generic memory subsystem in the leftmost column and how each level of the hierarchy is tested by the HPCC software (the second column from the left), along with the design goals for the future

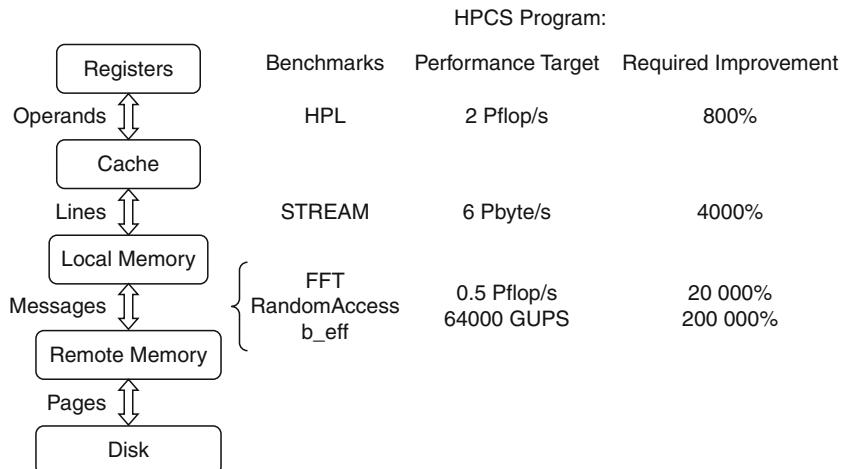


HPC Challenge Benchmark. Fig. 1 The application areas targeted by the HPCS Program are bound by the HPCC tests in the memory access locality space

systems that originated in the HPCS program (the third column from the right). In other words, these are the projected target performance numbers that are to come out of the winning HPCS vendor designs. The last column shows the relative improvement in performance that needs to be achieved in order to meet the goals.

The TOP500 Influence

The most commonly known ranking of supercomputer installations around the world is the TOP500 list [14]. It uses the equally well-known LINPACK benchmark [4] as a single figure of merit to rank 500 of the world’s most powerful supercomputers. The often-raised question about the relation between the TOP500 list and HPCC can be addressed by recognizing the positive aspects of the former. In particular, the longevity of the TOP500 list gives an unprecedented view of the high-end arena across the turbulent era of Moore’s law [10] rule and the emergence of today’s prevalent computing paradigms. The predictive power of the TOP500 list is likely to have a lasting influence in the future, as it has had in the past. HPCC extends the TOP500 list’s concept of exploiting a commonly used kernel and, in the context of the HPCS goals, incorporates a larger, growing suite of computational kernels. HPCC has already



HPC Challenge Benchmark. Fig. 2 HPCS program benchmarks and performance targets

HPC Challenge Benchmark. Table 1 All of the top 10 entries of the 27th TOP500 list that have results in the HPCC database

Rank	Name	Rmax	HPL	PTRANS	STREAM	FFT	RandomAccess	Lat.	B/w
1	BG/L	280.6	259.2	4665.9	160	2,311	35.47	5.92	0.16
2	BG W	91.3	83.9	171.5	50	1,235	21.61	4.70	0.16
3	ASC purple	75.8	57.9	553.0	44	842	1.03	5.11	3.22
4	Columbia	51.9	46.8	91.3	21	230	0.25	4.23	1.39
9	Red storm	36.2	33.0	1813.1	44	1,118	1.02	7.97	1.15

begun to serve as a valuable tool for performance analysis. Table 1 shows an example of how the data from the HPCC database can augment the TOP500 results (for the current version of the table please visit the HPCC website).

Short History of the Benchmark

The first reference implementation of the HPCC suite of codes was released to the public in 2003. The first optimized submission came in April 2004 from Cray, using the then-recent X1 installation at Oak Ridge National Lab. Ever since, Cray has championed the list of optimized HPCC submissions. By the time of the first HPCC birds-of-a-feather session at the Supercomputing conference in 2004 in Pittsburgh, the public database of results already featured major

supercomputer makers – a sign that vendors were participating in the new benchmark initiative. At the same time, behind the scenes, the code was also being tried out by government and private institutions for procurement and marketing purposes. A 2005 milestone was the announcement of the HPCC Awards contest. The two complementary categories of the competition emphasized performance and productivity – the same goals as the sponsoring HPCS program. The performance-emphasizing Class 1 award drew the attention of many of the biggest players in the supercomputing industry, which resulted in populating the HPCC database with most of the top 10 entries of the TOP500 list (some exceeding their performances reported on the TOP500 – a tribute to HPCC's continuous results update policy). The contestants competed to achieve the highest raw performance in one of the four

	HPL	Compute x from the system of linear equations $Ax = b$.
	DGEMM	Compute update to matrix C with a product of matrices A and B .
	STREAM	Perform simple operations on vectors a , b , and c .
	PTRANS	Compute update to matrix A with a sum of its transpose and another matrix B .
	RandomAccess	Perform integer update of random vector T locations using pseudo-random sequence.
	FFT	Compute vector z to be the Fast Fourier Transform (FFT) of vector x .
	b.eff	Perform ping-pong and various communication ring exchanges.

HPC Challenge Benchmark. Fig. 3 Detail description of the HPCC component tests (A, B, C – matrices, a, b, c, x, z – vectors, α , β – scalars, T – array of 64-bit integers)

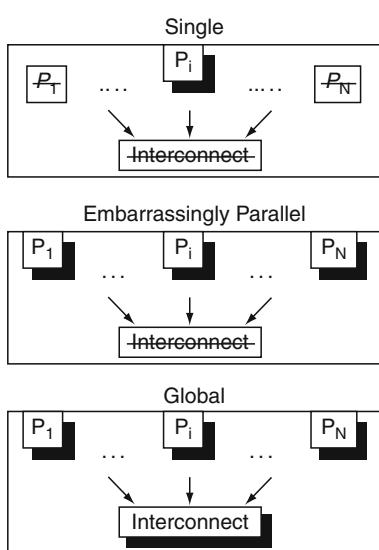
tests: HPL, STREAM, RANDA, and FFT. At the SC09 conference in Portland, Oregon, HPCC listed its first Pflop/s machine – Cray XT 5 called Jaguar from Oak Ridge National Laboratory. The Class 2 award, by solely focusing on productivity, introduced a subjectivity factor into the judging and also into the submission criteria, regarding what was appropriate for the contest. As a result, a wide range of solutions were submitted, spanning various programming languages (interpreted and compiled) and paradigms (with explicit and implicit parallelism). The Class 2 contest featured openly available as well as proprietary technologies, some of which were arguably confined to niche markets and some were

widely used. The financial incentives for entering turned out to be all but needless, as the HPCC seemed to have gained enough recognition within the high-end community to elicit entries even without the monetary assistance. (HPCwire provided both press coverage and cash rewards for the four winning contestants in Class 1 and the single winner in Class 2.) At the HPCC's second birds-of-a-feather session during the SC07 conference in Seattle, the former class was dominated by IBM's BlueGene/L at Lawrence Livermore National Lab, while the latter class was split among MTA pragmatically decorated C and UPC codes from Cray and IBM, respectively.

The Benchmark Tests' Details

Extensive discussion and various implementations of the HPCC tests are available elsewhere [5, 8, 15]. However, for the sake of completeness, this section provides the most important facts pertaining to the HPCC tests' definitions.

All calculations use double precision floating-point numbers as described by the IEEE 754 standard [1], and no mixed precision calculations [9] are allowed. All the tests are designed so that they will run on an arbitrary number of processors (usually denoted as p). Figure 3 shows a more detailed definition of each of the seven tests included in HPCC. In addition, it is possible to run the tests in one of three testing scenarios to stress various hardware components of the system. The scenarios are shown in Fig. 4. In the “Single” scenario, only one process is chosen to run the test. Accordingly, the remaining processes remain idle and so does the interconnect (shown with strike-out font in the Figure). In the “Embarrassingly Parallel” scenario, all processes run the tests simultaneously but they do not communicate with each other. And finally, in the “Global” scenario, all components of the system work together on all tests.



HPC Challenge Benchmark. Fig. 4 Testing scenarios of the HPCC components

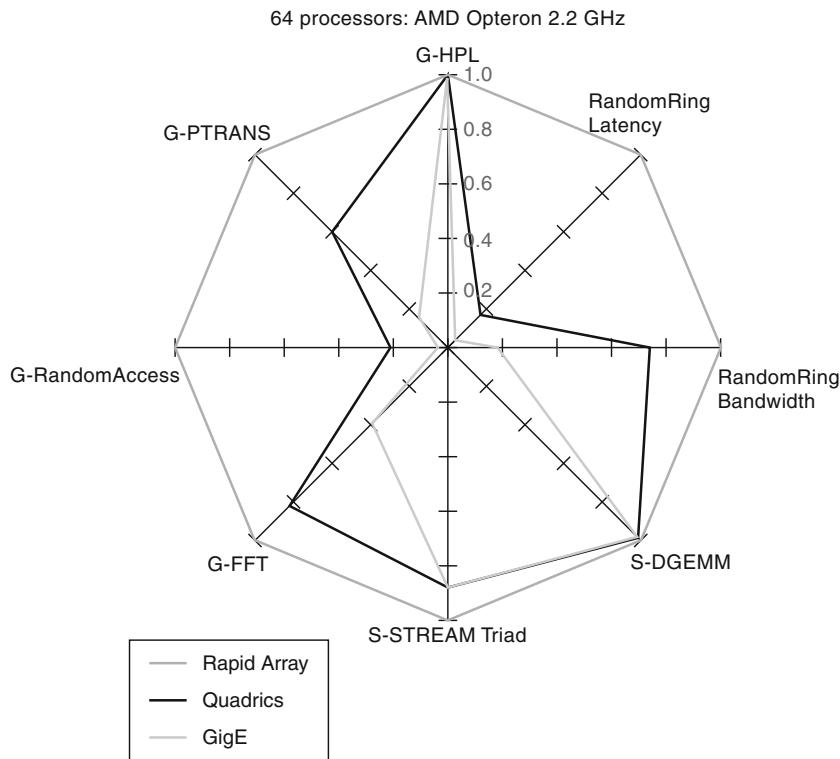
Benchmark Submission Procedures and Results

The reference implementation of the benchmark may be obtained free of charge at the benchmark's web site (<http://icl.cs.utk.edu/hpcc/>). The reference implementation should be used for the base run: it is written in a portable subset of ANSI C [7] using a hybrid programming model that mixes OpenMP [2, 16] threading with MPI [11–13] messaging. The installation of the software requires creating a script file for Unix's make(1) utility. The distribution archive comes with script files for many common computer architectures. Usually, a few changes to any of these files will produce the script file for a given platform. The HPCC rules allow only standard system compilers and libraries to be used through their supported and documented interface, and the build procedure should be described at submission time. This ensures repeatability of the results and serves as an educational tool for end users who wish to use a similar build process for their applications.

After a successful compilation, the benchmark is ready to run. However, it is recommended that changes be made to the benchmark's input file that describes the sizes of data to use during the run. The sizes should reflect the available memory on the system and the number of processors available for computations.

There must be one baseline run submitted for each computer system entered in the archive. An optimized run for each computer system may also be submitted. The baseline run should use the reference implementation of HPCC, and in a sense it represents the scenario when an application requires use of legacy code – a code that cannot be changed. The optimized run allows the submitter to perform more aggressive optimizations and use system-specific programming techniques (languages, messaging libraries, etc.), but at the same time still includes the verification process enjoyed by the base run.

All of the submitted results are publicly available after they have been confirmed by email. In addition to the various displays of results and exportable raw data, the HPCC website also offers a kiviat chart display to visually compare systems using multiple performance numbers at once. A sample chart that uses actual HPCC results data is shown in Fig. 5.



HPC Challenge Benchmark. Fig. 5 Sample kiviat diagram of results for three different interconnects that connect the same processors

Related Entries

- [Benchmarks](#)
- [LINPACK Benchmark](#)
- [Livermore Loops](#)
- [TOP500](#)

Bibliography

1. ANSI/IEEE Standard 754-1985 (1985) Standard for binary floating point arithmetic. Technical report, Institute of Electrical and Electronics Engineers, 1985
2. Chandra R, Dagum L, Kohr D, Maydan D, McDonald J, Menon R (2001) Parallel programming in OpenMP. Morgan Kaufmann Publishers, San Francisco, 2001
3. Dongarra JJ (1996) Performance of various computers using standard linear equations software. Computer Science Department. Technical Report, University of Tennessee, Knoxville, TN, April 1996. Up-to-date version available from <http://www.netlib.org/benchmark/>
4. Dongarra JJ, Luszczek P, Petitet A (2003) The LINPACK benchmark: past, present, and future. In: Dou Y, Gruber R, Joller JM (2003) Concurrency and computation: practice and experience, vol 15, pp 1–18
5. Dongarra J, Luszczek P (2005) Introduction to the HPC challenge benchmark suite. Technical Report UT-CS-05-544, University of Tennessee, Knoxville
6. Kepner J (2004) HPC productivity: an overarching view. Int J High Perform Comput Appl 18(4):393–397
7. Kernighan BW, Ritchie DM (1978) The C Programming Language. Prentice-Hall, Upper Saddle River, New Jersey
8. Luszczek P, Dongarra J (2006) High performance development for high end computing with Python Language Wrapper (PLW). Int J High Perform Comput Appl. Accepted to Special Issue on High Productivity Languages and Models
9. Langou J, Langou J, Luszczek P, Kurzak J, Buttari A, Dongarra J (2006) Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy. In: Proceedings of SC06, Tampa, Florida, November 11–17 2006. See <http://icl.cs.utk.edu/iter-ref>
10. Moore GE (1965) Cramming more components onto integrated circuits. Electronics 8(8):114–117
11. Message Passing Interface Forum (1994) MPI: A Message-Passing Interface Standard. The International Journal of Supercomputer Applications and High Performance Computing 8(3/4):165–414
12. Message Passing Interface Forum (1995) MPI: A Message-Passing Interface Standard (version 1.1), 1995. Available at: <http://www mpi-forum.org/>

13. Message Passing Interface Forum (1997) MPI-2: Extensions to the Message-Passing Interface, 18 July 1997. Available at <http://www.mpi-forum.org/docs/mpi-20.ps>
14. Meuer HW, Strohmaier E, Dongarra JJ, Simon HD (2006) TOP500 Supercomputer Sites, 28th edn. November 2006. (The report can be downloaded from <http://www.netlib.org/benchmark/top500.html>)
15. Nadya Travinin and Jeremy Kepner (2007) pMatlab parallel Matlab library. International Journal of High Performance Computing Applications 21(3):336–359
16. OpenMP: Simple, portable, scalable SMP programming. <http://www.openmp.org/>

HPF (High Performance Fortran)

High Performance Fortran (HPF) is an extension of Fortran 90 for parallel programming. In HPF programs, parallelism is represented as data parallel operations in a single thread of execution. HPF extensions included statements to specify data distribution, data alignment, and processor topology, which were used for the translation of HPF codes onto an SPMD message-passing form.

Bibliography

1. Kennedy K, Koelbel C, Zima H (2007) The rise and fall of high performance Fortran: an historical object lesson. In: Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III), ACM, New York, pp. 7-1-7-22, doi:10.1145/1238844.1238851, <http://doi.acm.org/10.1145/1238844.1238851>
2. High Performance Fortran Forum Website. <http://hpff.rice.edu>

HPS Microarchitecture

YALE N. PATT

The University of Texas at Austin, Austin, TX, USA

Synonyms

The high performance substrate

Definition

The microarchitecture specified by Yale Patt, Wen-mei Hwu, Stephen Melvin, and Michael Shebanow in 1984 for implementing high-performance microprocessors.

It achieves high performance via aggressive branch prediction, speculative execution, wide issue, and out-of-order execution, while retaining the ability to handle precise exceptions via in-order retirement.

Discussion

The High Performance Substrate (HPS) was the name given to the microarchitecture conceived by Professor Yale Patt and his three PhD students, Wen-mei Hwu, Michael Shebanow, and Stephen Melvin, at the University of California, Berkeley in 1984 and first published in Micro 18 in October, 1985 [1, 2]. Its goal was high-performance processing of single-instruction streams by combining aggressive branch prediction, speculative execution, wide issue, dynamic scheduling (out-of-order execution), and retirement of instructions in program order (i.e., in-order).

Out-of-order execution had appeared in previous machines from Control Data [3] and IBM [4], but had pretty much been dismissed as a nonviable mechanism due to its lack of in-order retirement, which prevented the processor from implementing precise exceptions. The checkpoint retirement mechanisms of HPS removed that problem [5, 6]. It should also be noted that solutions to the precise exception problem were also being developed simultaneously and independently by James Smith and Andrew Plezskun [7].

HPS was first targeted for the VAX instruction set architecture and demonstrated that an HPS implementation of the VAX could process instruction streams at a rate of three cycles per instruction (CPI), as compared to the VAX-11/780, which processed at the rate of 10.5 CPI [8].

Instructions are processed as follows: Using an aggressive branch predictor and wide-issue fetch/decode mechanism, multiple instructions are fetched each cycle, decoded into data flow graphs (one per instruction), and merged into a global data flow graph containing all instructions in process. Instructions are scheduled for execution when their flow dependencies (RAW hazards) have been satisfied and executed speculatively and out-of-order with respect to the program order of the program. Results produced by these

instructions are stored temporarily in a results buffer (aka re-order buffer) until they can be retired in-order. The essence of the paradigm is that the global data graph consists of nodes corresponding to micro-operations and edges corresponding to linkages between micro-ops that produce operands and micro-ops that source them. The edges of the data flow graph produced as a result of decode correspond to internal linkages within an instruction. Edges created as a result of merging an individual instruction's data flow graph into the global data flow graph correspond to linkages between live-outs of one instruction and live-ins of a subsequent instruction. A Register Alias Table was conceived to maintain correct linkages between live-outs and live-ins. A node, corresponding to a micro-op, is available for execution when all its flow dependencies are resolved.

The HPS research group refers to the paradigm as Restricted Data Flow (RDF) since at no time does the data flow graph for the entire program exist. Rather, the size of the global data flow graph is increased every cycle as a result of new instructions being decoded and merged, and decreased every cycle as a result of old instructions retiring. At every point in time, only those instructions in the active window – the set of instructions that have been fetched but not yet retired – are present in the data flow graph. The set of instructions in the active window are often referred to as “in-flight” instructions. The number of in-flight instructions is orders of magnitude smaller than the size of a data flow graph for the entire program. The result is data flow processing of a program without incurring any of the problems of classical data flow.

Since 1985, the HPS microarchitecture has seen continual development and improvement by many research groups at many universities and industrial labs. The basic paradigm has been adopted for most cutting-edge high-performance microprocessors, starting with Intel on its Pentium Pro microprocessor in the late 1990s [9].

Bibliography

1. Patt YN, Hwu W, Shebanow M (1985) HPS, a new microarchitecture: rationale and introduction. In: Proceedings of the 18th microprogramming workshop, Asilomar, CA
2. Patt YN, Melvin S, Hwu W, Shebanow M (1985) Critical issues regarding HPS, a high performance microarchitecture. In: Proceedings of the 18th microprogramming workshop, Asilomar, CA
3. Thornton JE (1970) Design of a computer – the Control Data 6600. Scott, Foresman and Co. Glenview, IL
4. Anderson DW, Sparacio FJ, Tomasulo RM (1967) The IBM system/360 model 91: machine philosophy and instruction-handling. IBM J Res Development 11(1):8–24
5. Hwu W, Patt Y (1986) HPSm, a high performance restricted data flow architecture having minimal functionality. In: Proceedings, 13th annual international symposium on computer architecture, Tokyo
6. Hwu W, Patt Y (1987) Checkpoint repair for high performance out-of-order execution machines. IEEE Trans Computers 36(12):1496–1514
7. Smith JE, Pleszkun A (1985) Implementing precise interrupts. In: Proceedings, 12th annual international symposium on computer architecture, Boston, MA
8. Hwu W, Melvin S, Shebanow M, Chen C, Wei J, Patt Y (1986) An HPS implementation of VAX; initial design and analysis. In: Proceedings of the Hawaii international conference on systems sciences, Honolulu, HI
9. Colwell R (2005) The pentium chronicles: the people, passion, and politics behind intel's landmark chips. Wiley-IEEE Computer Society Press, NJ, ISBN: 978-0-471-73617-2

HT

► [HyperTransport](#)

HT3.10

► [HyperTransport](#)

Hybrid Programming With SIMPLE

GUOJING CONG¹, DAVID A. BADER²

¹IBM, Yorktown Heights, NY, USA

²Georgia Institute of Technology, Atlanta, GA, USA

Definition

Most high performance computing systems are clusters of shared-memory nodes. Hybrid parallel programming handles distributed-memory parallelization across the nodes and shared-memory parallelization within a node.

SIMPLE refers to the joining of the **SMP** and **MPI**-like message passing paradigms [7] and the *simple* programming approach. It provides a methodology of programming cluster of SMP nodes. It advocates a hybrid methodology which maps directly to underlying architectural aspects. SIMPLE combines shared memory programming on shared memory nodes with message passing communication between these nodes. SIMPLE provides (1) a complexity model and set of efficient communication primitives for SMP nodes and clusters; (2) a programming methodology for clusters of SMPs which is both efficient and portable; and (3) high performance algorithms for sorting integers, constraint-satisfied searching, and computing the two-dimensional FFT.

The SIMPLE Computational Model

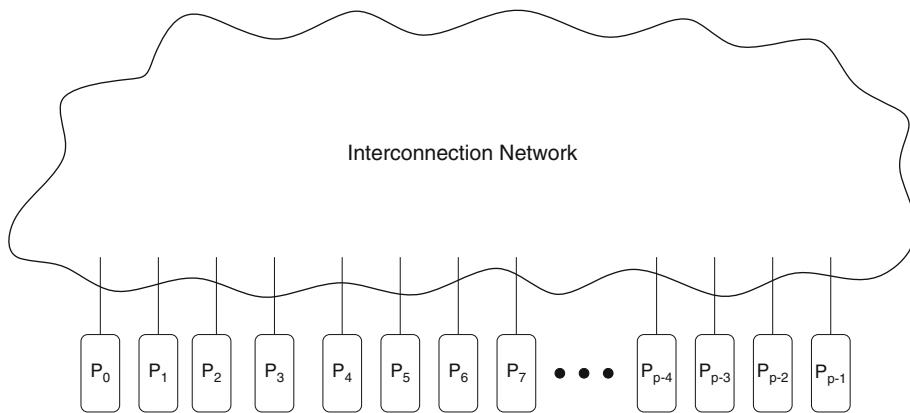
A simple paradigm is used in SIMPLE for designing efficient and portable parallel algorithms. The architecture consists of a collection of SMP nodes interconnected by a communication network (as shown in Fig. 1) that can be modeled as a complete graph on which communication is subject to the restrictions imposed by the latency and the bandwidth properties of the network. Each SMP node contains several identical processors, each typically with its own on-chip cache and a larger off-chip cache, which have uniform access to a shared memory and other resources such as the network interface.

Parameter r is used to represent the number symmetric processors per node (see Fig. 2 for a diagram of a typical node). Notice that each CPU typically has its own on-chip cache (L1) and a larger off-chip level

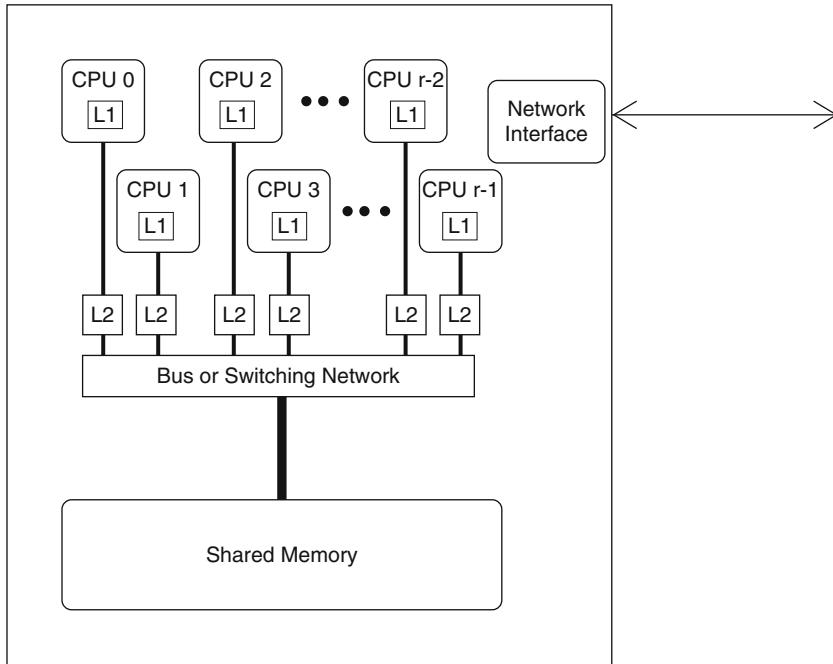
two cache (L2), which can be tightly integrated into the memory system to provide fast memory accesses and cache coherence. The shared memory programming of each SMP node is based on threads which communicate via coordinated accesses to shared memory. SIMPLE provides several primitives that synchronize the threads at a barrier, enable one thread to broadcast data to the other threads, or calculate reductions across the threads. In SIMPLE, only the CPUs from a certain node have access to that node's configuration. In this manner, there is no restriction that all nodes must be identical, and certainly configuration can be constructed from SMP nodes of different sizes. Thus, the number of threads on a specific remote node is not globally available. Because of this, SIMPLE supports only node-oriented communication, meaning that communication is restricted such that, given any source node s and destination node d , with $s \neq d$, only one thread on node s can send (receive) a message to (from) node d at any given time.

Complexity Model

In the SIMPLE complexity model, each SMP is viewed as a two-level hierarchy for which good performance requires both good load distribution and the minimization of secondary memory access. The cluster is viewed as a collection of powerful processors connected by a communication network. Maximizing performance on the cluster requires both efficient load balancing and regular, balanced communication. Hence, our performance model combines two separate but complimentary models.



Hybrid Programming With SIMPLE. Fig. 1 Cluster of processing elements



H

Hybrid Programming With SIMPLE. Fig. 2 A typical symmetric multiprocessing (SMP) node used in a cluster. L1 is on-chip level-one cache, and L2 is off-chip level-two cache

The SIMPLE model recognizes that efficient algorithm design requires the efficient decomposition of the problem among the available processors, and so, unlike some other models for hierarchical memory, the cost of computation is included in the complexity. The cost model also encourages the exploitations of temporal and spatial locality. Specifically, memory at each SMP is seen as consisting of two levels: cache and main memory. A block of m contiguous words can be read from or written to main memory in $(\epsilon + \frac{mr}{\alpha})$ time, where ϵ is the latency of the bus, r is the number of processors competing for access to the bus, and α is the bandwidth. By contrast, the transfer of m noncontiguous words would require $m(\epsilon + \frac{r}{\alpha})$ time.

A parallel algorithm is viewed as a sequence of local SMP computations interleaved with communication steps, where computation and communication is allowed to overlap. Assuming no congestion, the transfer of a block consisting of m words between two nodes takes $(\tau + \frac{m}{\beta})$ time, where τ is the latency of the network, and β is the bandwidth per node. SIMPLE assumes that the bisection bandwidth is sufficiently high to support block permutation routings among the p nodes at the rate of $\frac{1}{\beta}$. In particular, for any subset

of q nodes, a block permutation among the q nodes takes $(\tau + \frac{m}{\beta})$ time, where m is the size of the largest block. Using this cost model, the communication time $T_{comm}(n, p)$ of an algorithm can be viewed as a function of the input size n , the number of nodes p , and the parameters τ and β . The overall complexity of algorithm for the cluster $T(n, p)$ is given by the sum of T_{smp} and $T_{comm}(n, p)$.

Communication Primitives

The communication primitives are grouped into three modules: Internode Communication Library (ICL), **SMP Node**, and SIMPLE. ICL communication primitives handle internode communication, **SMP Node** primitives aid shared-memory node algorithms, and SIMPLE primitives combine **SMP Node** with ICL on SMP clusters.

The ICL communication library services internode communication and can use any of the vendor-supplied or freely available thread-safe implementation of MPI. The ICL libraries are based upon a reliable, application-layer send and receive primitive, as well as a send-and-receive primitive which handles the exchanging of messages between

sets of nodes where each participating node is the source and destination of one message. The library also provides a barrier operation based upon the send and receive which halts the execution at each node until all nodes check into the barrier, at which time, the nodes may continue execution. In addition, ICL includes collective communication primitives, for example, scan, reduce, broadcast, allreduce, alldtoall, alldtoallv, gather, and scatter. See [1].

SMP Node

The **SMP Node** Library contains important primitives for an SMP node: barrier, replicate, broadcast, scan, reduce, and allreduce, whereby on a single node, barrier synchronizes the threads, broadcast ensures that each thread has the most recent copy of a shared memory location, scan (reduce) performs a prefix (reduction) operation with a binary associative operator (e.g., addition, multiplication, maximum, minimum, bitwise-AND, and bitwise-OR) with one datum per thread, and allreduce replicates the result from reduce.

Each of these collective SMP primitives can be implemented using a *fan-out* or *fan-in* tree constructed as follows. A logical k -ary balanced tree is built for an SMP node with r processors, which serves as both a fan-out and fan-in communication pattern. In a k -ary tree, level 0 has one processor, level 1 has k processors, level 2 has k^2 processors, and so forth, with level j containing k^j processors, for $0 \leq j \leq L - 1$, where there are L levels in the tree. If $\log_k r$ is not an integer, then the last level ($L - 1$) will hold less than k^{L-1} processors. Thus, the number of processors r is bounded by

$$\sum_{j=0}^{L-2} k^j < r \leq \sum_{j=0}^{L-1} k^j. \quad (1)$$

Solving for the number of levels L , it is easy to see that

$$L = \lceil \log_k(r(k-1) + 1) \rceil \quad (2)$$

where the ceiling function $\lceil x \rceil$ returns the smallest integer greater than or equal to x .

An efficient algorithm for replicating a data buffer B such that each processor i , ($0 \leq i \leq r - 1$), receives a unique copy B_i of B makes use of the fan-out tree, with the source processor situated as the root node of the k -ary tree. During step j of the algorithm,

for ($0 \leq j \leq L - 2$), each of k^j processors writes k unique copies of the data for its k children. The time complexity of this SMP replication algorithm given an m -word buffer is

$$\begin{aligned} T_{smp} &= \sum_{j=0}^{L-2} k \left(\epsilon + \frac{k^j m}{\alpha} \right) \\ &= k(L-1)\epsilon + \frac{m}{\alpha}(r-1) \\ &\leq k(\log_k(r(k-1) + 1) - 1)\epsilon + \frac{m}{\alpha}(r-1) \\ &\leq k(\log_k(r + 1/k))\epsilon + \frac{m}{\alpha}(r-1) \\ &= O((\log r)\epsilon + \frac{mr}{\alpha}). \end{aligned} \quad (3)$$

The best choice of k , ($2 \leq k \leq r - 1$), depends on SMP size r and machine parameters ϵ and α , but can be chosen to minimize Eq. 3.

An algorithm which barrier synchronizes r SMP processors can use a fan-in tree followed by a fan-out tree, with a unit message size m , taking twice the replication time, namely, $O((\log r)\epsilon + r/\alpha)$.

For certain SMP algorithms, it may not be necessary to replicate data, but to share a read-only buffer for a given step. A broadcast SMP primitive supplies each processor with the address of the shared buffer by replicating the memory address in $T_{smp} = O((\log r)\epsilon + r/\alpha)$.

A reduce primitive, which performs a reduction with a given binary associative operator, uses a fan-in tree, combining partial sums during each step. For initial data arrays of size m per processor, this takes $O((\log r)\epsilon + mr/\alpha)$. The allreduce primitive performs a reduction followed by replicate so that each processor receives a copy of the result with a cost of $O((\log r)\epsilon + \frac{mr}{\alpha})$.

Scans (also called prefix-sums) are defined as follows. Each processor i , ($0 \leq i \leq r - 1$), initially holds an element a_i , and at the conclusion of this primitive, holds the prefix-sum $b_i = a_0 * a_1 * \dots * a_i$, where $*$ is any binary associative operator. An SMP algorithm similar to the PRAM algorithm (e.g., [5]) is employed which uses a binary tree for finding the prefix-sums. Given an array of elements A of size $r = 2^d$ where d is a nonnegative integer, the output is array C such that $C(0, i)$ is the i th prefix-sum, for ($0 \leq i \leq r - 1$).

In fact, arrays A , B , and C in the SMP prefix-sum algorithm (Alg. 1) can be the same array. The analysis is as follows. The first for loop takes $\sum_{h=1}^{\log r} 3(\epsilon + \frac{r}{2^h} \frac{m}{\alpha})$,

and the second **for** loop takes $\sum_{h=0}^{\log r} \binom{3+2}{2} (\epsilon + \frac{r}{2^h} \frac{m}{\alpha})$ for a total complexity of $T_{smp} \leq 8\epsilon \log r + 3\frac{mr}{\alpha} = O((\log r)\epsilon + \frac{mr}{\alpha})$.

Simple

Finally, the SIMPLE communication library, built on top of ICL and **SMP Node**, includes the primitives for the SIMPLE model: barrier, scan, reduce, broadcast, allreduce, alltoall, alltoallv, gather, and scatter. These hierarchical layers of our communication libraries are pictured in Fig. 3.

The **SMP Node**, ICL, and SIMPLE libraries are implemented at a high level, completely in user space

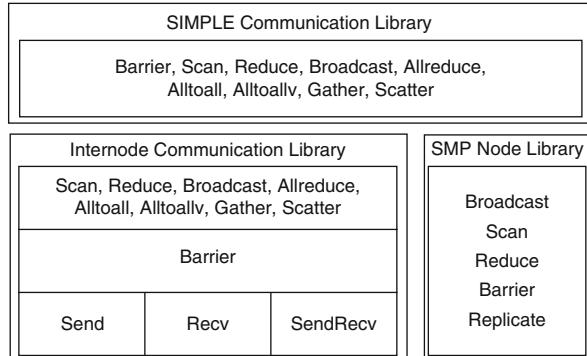
Algorithm 1 SMP scan algorithm for processor i , ($0 \leq i \leq r - 1$) and binary associative operator $*$

```

set  $B(0, i) = A(i)$ .

for  $h = 1$  to  $\log r$  do
  if  $1 \leq i \leq r/2^h$  then
    set  $B(h, i) = B(h - 1, 2i - 1) * B(h - 1, 2i)$ .

for  $h = \log r$  downto 0 do
  if  $1 \leq i \leq r/2^h$  then
    {
      If  $i$  even, Set  $C(h, i) = C(h + 1, i/2)$ ;
      If  $i = 1$ , Set  $C(h, 1) = B(h, 1)$ ;
      If  $i$  odd, Set  $C(h, i) = C(h + 1, (i - 1)/2) * B(h, i)$ .
    }
  
```



Hybrid Programming With SIMPLE. Fig. 3 Hierarchy of SMP, message passing, and SIMPLE communication libraries

(see Fig. 4). Because no kernel modification is required, these libraries easily port to new platforms.

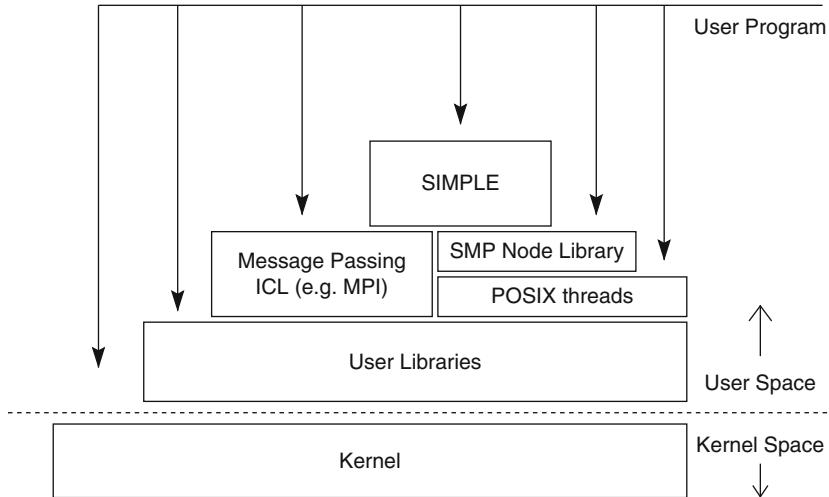
As mentioned previously, the number of threads per node can vary, along with machine size. Thus, each thread has a small set of context information which holds such parameters as the number of threads on the given node, the number of nodes in the machine, the rank of that node in the machine, and the rank of the thread both (1) on the node and (2) across the machine. Table 1 describes these parameters in detail.

Because the design of the communication libraries is modular, it is easy to experiment with different implementations. For example, the ICL module can make use of any of the freely available or vendor-supplied thread-safe implementations of MPI, or a small communication kernel which provides the necessary message passing primitives. Similarly, the **SMP Node** primitives can be replaced by vendor-supplied SMP implementations.

The Alltoall Primitive

One of the most important collective communication events is the alltoall (or transpose) primitive which transmits regular sized blocks of data between each pair of nodes. More formally, given a collection of p nodes each with an m element sending buffer, where p divides m , the alltoall operation consists of each node i sending its j th block of $\frac{m}{p}$ data elements to node j , where node j stores the data from i in the i th block of its receiving buffer, for all $(0 \leq i, j \leq p - 1)$. An efficient message passing implementation of alltoall would be as follows. The notation “ var_i ” refers to memory location “ $var + (\frac{m}{p} * i)$,” and src and dst point to the source and destination arrays, respectively.

To implement this algorithm (Alg. 2), multiple threads ($r \leq p$) per node are used. The local memory copy trivially can be performed concurrently by one thread while the remaining threads handle the internode communication as follows. The $p - 1$ iterations of the loop are partitioned in a straightforward manner to the remaining threads. Each thread has the information necessary to calculate its subset of loop indices, and thus, this loop partitioning step requires no synchronization overheads. The complexity of this primitive is twice $(\epsilon + \frac{m}{r} \frac{r}{\alpha})$ for the local memory read



Hybrid Programming With SIMPLE. Fig. 4 User code can access SIMPLE, SMP, message passing, and standard user libraries. Note that SIMPLE operates completely in user space

Hybrid Programming With SIMPLE. Table 1 The local context parameters available to each SIMPLE thread

Parameter	Description
NODES = p	Total number of nodes in the cluster
MYNODE	My node rank, from 0 to NODES – 1
THREADS = r	Total number of threads on my node
MYTHREAD	The rank of my thread on this node, from 0 to THREADS – 1
TID	Total number of threads in the cluster
ID	My thread rank, with respect to the cluster, from 0 to TID – 1

Algorithm 2 SIMPLE Alltoall primitive

copy the appropriate $\frac{m}{p}$ elements from src_{MYNODE} to dst_{MYNODE} .

```
for i = 1 to NODES – 1 do
    set k = MYNODE  $\oplus$  i;
    send  $\frac{m}{p}$  elements from  $src_k$  to node k, and
    receive  $\frac{m}{p}$  elements from node k to  $dst_k$ .
```

Computation Primitives

SIMPLE computation primitives do not communicate data but affect a thread's execution through (1) loop parallelization, (2) restriction, or (3) shared memory management. Basic support for data parallelism, that is, “parallel do” concurrent execution of loops across processors on one or more nodes, is provided.

Data Parallel

The SIMPLE methodology contains several basic “pardo” directives for executing loops concurrently on one or more SMP nodes, provided that no dependencies exist in the loop. Typically, this is useful when an independent operation is to be applied to every location in an array, for example, in the element-wise addition of two arrays. Pardo implicitly partitions the loop to the threads without the need for coordinating overheads such as synchronization or communication between processors. By default, pardo uses block partitioning of the loop assignment values to the threads, which typically results in better cache utilization due to the array locations on left-hand side of the assignment being owned by local caches more often than not. However, SIMPLE explicitly provides both block and cyclic partitioning interfaces for the pardo directive.

Similar mechanisms exist for parallelizing loops across nodes. The `all_pardo_cyclic (i, a, b)` directive will cyclically assign each iteration of the loop

and writes, and $(\tau + \frac{m}{\beta})$ for internode communication, for a total cost of $O(\tau + \frac{m}{\beta} + \epsilon + \frac{m}{\alpha})$.

across the entire collection of processors. For example, $i = a$ will be executed on the first processor of the first node, $i = a + 1$ on the second processor of the first node, and so on, with $i = a + r - 1$ on the last processor of the first node. The iteration with $i = a + r$ is executed by the first processor on the second node. After $r \cdot p$ iterations are assigned, the next index will again be assigned to the first processor on the first node. A similar directive called `all_pardo_block`, which accepts the same arguments, assigns the iterations in a block fashion to the processors; thus, the first $\frac{b-a}{rp}$ iterations are assigned to the first processor, the next block of iterations are assigned to the second processor, and so forth. With either of these SIMPLE directives, each processor will execute at most $\lceil \frac{n}{rp} \rceil$ iterations for a loop of size n .

Control

The second category of SIMPLE computation primitives control which threads can participate in the context by using restrictions.

[Table 2](#) defines each control primitive and gives the largest number of threads able to execute the portion of the algorithm restricted by this statement. For example, if only one thread per node needs to execute a command, it can be preceded with the `on_one_thread` directive. Suppose data has been gathered to a single node. Work on this data can be accomplished on that node by preceding the statement with `on_one_node`. The combination of these two primitives restricts execution to exactly one thread, and can be shortcut with the `on_one` directive.

Memory Management

Finally, shared memory allocations are the third category of SIMPLE computation primitives. Two directives are used:

1. `node_malloc` for dynamically allocating a shared structure
2. `node_free` for releasing this memory back to the heap

The `node_malloc` primitive is called by all threads on a given node, and takes as a parameter the number of bytes to be allocated dynamically from the heap. The primitive returns to each thread a valid pointer to the shared memory location. In addition, a thread may allow others to access local data by broadcasting the corresponding memory address. When this shared memory is no longer required, the `node_free` primitive releases it back to the heap.

H

SIMPLE Algorithmic Design

Programming Model

The user writes an algorithm for an arbitrary cluster size p and SMP size r (where each node can assign possibly different values to r at runtime), using the parameters from [Table 1](#). SIMPLE expects a standard main function (called `SIMPLE_main()`) that, to the user's view, is immediately up and running on each thread. Thus, the user does not need to make any special calls to initialize the libraries or communication channels. SIMPLE makes available the rank of each thread on its node or across the cluster, and algorithms can use these ranks in a straightforward fashion to break symmetries and

Hybrid Programming With SIMPLE. [Table 2](#) Subset of SIMPLE control primitives

Control Primitives				
Primitive	Definition	Max number of participating threads	MYNODE restriction	MYTHREAD restriction
<code>on_one_thread</code>	only one thread per node	p		0
<code>on_one_node</code>	all threads on a single node	r	0	
<code>on_one</code>	only one thread on a single node	1	0	0
<code>on_thread(i)</code>	one thread (i) per node	p		i
<code>on_node(j)</code>	all threads on node j	r	j	

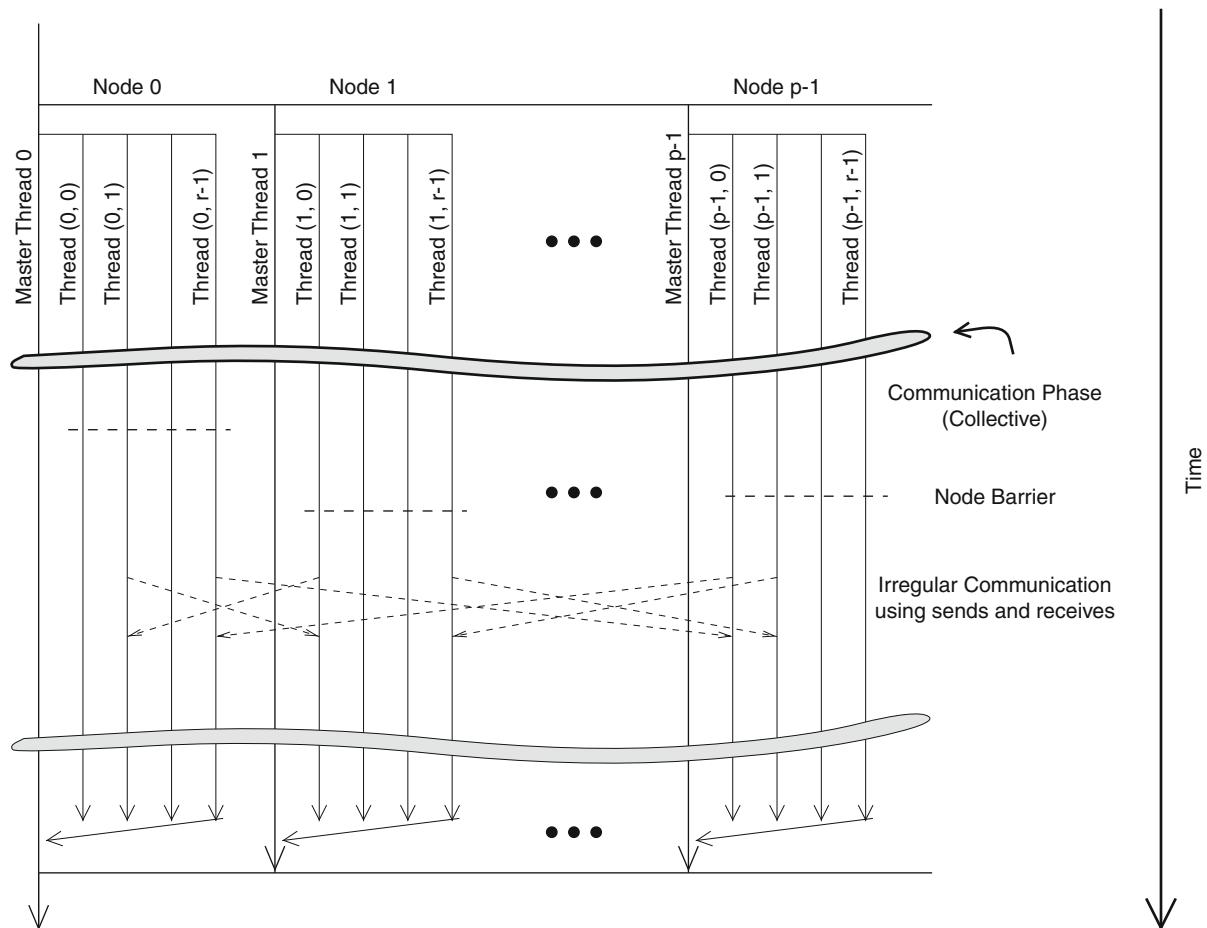
partition work. The only argument of SIMPLE_main() is “**THREADED**,” a macro pointing to a private data structure which holds local thread information. If the user’s main function needs to call subroutines which make use of the SIMPLE library, this information is easily passed via another macro “**TH**” in the calling arguments. After all threads exit from the main function, the SIMPLE code performs a shut down process.

Runtime Support

When a SIMPLE algorithm first begins its execution, the SIMPLE runtime support has already initialized parallel mechanisms such as barriers and established the network-based internode communication channels which remain open for the life of the program. The various libraries described in Sect. “The SIMPLE

Computational Model” have runtime initializations which take place as follows.

The runtime start-up routines for a SIMPLE algorithm are performed in two steps. First, the ICL initialization expands computation across the nodes in a cluster by launching a master thread on each of the p nodes and establishing communication channels between each pair of nodes. Second, each master thread launches r user threads, where each node is at least an r -way SMP. (A rule of thumb in practice is to use r threads on an $r + 1$ -way SMP node, which allows operating system tasks to fully utilize at least one CPU.) It is assumed that the r CPUs concurrently execute the r threads. The thread flow of an example SIMPLE algorithm is shown in Fig. 5. As mentioned previously, our methodology supports



Hybrid Programming With SIMPLE. Fig. 5 Example of a SIMPLE algorithm flow of master and compute-based user threads. Note that the only responsibility of each master thread is only to launch and later join user threads, but never to participate in computation or communication

only node-oriented communication, that is, given any source node s and destination node d , with $s \neq d$, only one thread on node s can send (receive) a message to (from) node d during a communication step. Also note that the master thread does not participate in any computation, but sits idle until the completion of the user code, at which time it coordinates the joining of threads and exiting of processes.

The programming model is simply implemented using a portable thread package called POSIX threads (**pthreads**).

A Possible Approach

The latency for message passing is an order of magnitude higher than accessing local memory. Thus, the most costly operation in a SIMPLE algorithm is internode communication, and algorithmic design must attempt to minimize the communication costs between the nodes.

Given an efficient message passing algorithm, an incremental process can be used to design an efficient SIMPLE algorithm. The computational work assigned to each node is mapped into an efficient SMP algorithm. For example, independent operations such as those arising in *functional parallelism* (e.g., independent I/O and computational tasks, or the local memory copy in the SIMPLE `alltoall` primitive presented in the previous section) or *loop parallelism* typically can be *threaded*. For functional parallelism, this means that each thread acts as a functional process for that task, and for loop parallelism, each thread computes its portion of the computation concurrently. Loop transformations may be applied to reduce data dependencies between the threads. Thread synchronization is a costly operation when implemented in software and, when possible, should be avoided.

Example: Radix Sort

Consider the problem of sorting n integers spread evenly across a cluster of p shared-memory r -way SMP nodes, where $n \geq p^2$. Fast integer sorting is crucial for solving problems in many domains, and as such, is used as a kernel in several parallel benchmarks such as NAS (Note that the NAS IS benchmark requires that the integers be ranked and not necessarily placed in sorted order.) [4] and SPLASH [8].

Consider the problem of sorting n integer keys in the range $[0, M - 1]$ that are distributed equally in the

shared memories of a p -node cluster of r -way SMPs. **Radix Sort** decomposes each key into groups of ρ -bit digits, for a suitably chosen ρ , and sorts the keys by applying a counting sort routine on each of the ρ -bit digits beginning with the digit containing the least significant bit positions [6]. Let $R = 2^\rho \geq p$. Assume (w.l.o.g.) that the number of nodes is a power of two, say $p = 2^k$, and hence $\frac{R}{p}$ is an integer $= 2^{\rho-k} \geq 1$.

SIMPLE Counting Sort Algorithm

Counting Sort algorithm sorts n integers in the range $[0, R - 1]$ by using R counters to accumulate the number of keys equal to the value i in bucket B_i , for $0 \leq i \leq R - 1$, followed by determining the rank of each key. Once the rank of each key is known, each key can be moved into its correct position using a permutation ($\frac{n}{p}$ -relation) routing [2, 3], whereby no node is the source or destination of more than $\frac{n}{p}$ keys. Counting Sort is a **stable** sorting routine, that is, if two keys are identical, their relative order in the final sort remains the same as their initial order.

Algorithm 3 SIMPLE Counting Sort Algorithm

Step (1): For each node i , $(0 \leq i \leq p - 1)$, count the frequency of its $\frac{n}{p}$ keys; that is, compute $H_{i[k]}$, the number of keys equal to k , for $(0 \leq k \leq R - 1)$.

Step (2): Apply the `alltoall` primitive to the H arrays using the block size $\frac{R}{p}$. Hence, at the end of this step, each node will hold $\frac{R}{p}$ consecutive rows of H .

Step (3): Each node locally computes the prefix-sums of its rows of the array H .

Step (4): Apply the (inverse) `alltoall` primitive to the R corresponding prefix-sums augmented by the total count for each bin. The block size of the `alltoall` primitive is $2\frac{R}{p}$.

Step (5): On each node, compute the ranks of the $\frac{n}{p}$ local elements using the arrays generated in **Steps (1)** and **(4)**.

Step (6): Perform a personalized communication of keys to rank location using an $\frac{n}{p}$ -relation algorithm.

The pseudocode for our Counting Sort algorithm (Alg. 3) uses six major steps and can be described as follows.

In **Step (1)**, the computation can be divided evenly among the threads. Thus, on each node, each of r threads (**A**) histograms $\frac{1}{r}$ of the input concurrently, and (**B**) merges these r histograms into a single array for node i . For the prefix-sum calculations on each node in **Step (3)**, since the rows are independent, each of r threads can compute the prefix-sum calculations for $\frac{R}{rp}$ rows concurrently. Also, the computation of ranks on each node in **Step (5)** can be handled by r threads, where each thread calculates $\frac{n}{rp}$ ranks of the node's local elements. Communication can also be improved by replacing the message passing `alltoall` primitive used in **Steps (2)** and **(4)** with the appropriate SIMPLE primitive.

The h -relation used in the final step of Counting Sort is a permutation routing since $h = \frac{n}{p}$, and was given in the previous section.

Histogramming in **Step (1A)** costs $O\left(\epsilon + \frac{n}{p} \frac{1}{\alpha}\right)$ to read the input and $O\left(\epsilon + R \frac{r}{\alpha}\right)$ for each processor to write its histogram into main memory. Merging in **Step (1B)** uses an SMP reduce with cost $O\left((\log r)\epsilon + R \frac{r}{\alpha}\right)$. SIMPLE `alltoall` in **Step (2)** and the inverse `alltoall` in **Step (4)** take $O\left(\tau + \frac{R}{\beta} + \epsilon + \frac{R}{\alpha}\right)$ time. Computing local prefix-sums in **Step (3)** costs $O\left(\epsilon + \frac{R}{pr} p \frac{r}{\alpha}\right)$. Ranking each element in **Step (5)** takes $O\left(\epsilon + \frac{n}{p} \frac{1}{\alpha}\right)$ time. Finally, the SIMPLE permutation with $h = \frac{n}{p}$ costs $O\left(\tau + \frac{n}{p} \left(\frac{1}{\beta} + \frac{1}{\alpha} + \frac{\epsilon}{r} \right)\right)$, for $\frac{n}{p} \geq r \cdot \max(p, \log r)$. Thus, the total complexity for Counting Sort, assuming that $n \geq p \cdot \max(R, r \cdot \max(p, \log r))$ is

$$O\left(\tau + \frac{n}{p} \left(\frac{1}{\beta} + \frac{r}{\alpha} + \frac{\epsilon}{r} \right)\right). \quad (4)$$

SIMPLE Radix Sort Algorithm

The message passing **Radix Sort** algorithm makes several passes of the previous message passing Counting Sort in order to completely sort integer keys. Counting Sort can be used as the intermediate sorting routine because it provides a stable sort. Let the n integer keys fall in the range $[0, M - 1]$, and $M = 2^b$. Then $\frac{b}{\rho}$ passes of Counting Sort is needed; each pass

works on ρ -bit digits of the input keys, starting from the least significant digit of ρ bits to the most significant digit. Radix Sort easily can be adapted for clusters of SMPs by using the SIMPLE Counting Sort routine. Thus, the total complexity for Radix Sort, assuming that $n \geq p \cdot \max(R, r \cdot \max(p, \log r))$ is

$$O\left(\frac{b}{\rho} \left(\tau + \frac{n}{p} \left(\frac{1}{\beta} + \frac{r}{\alpha} + \frac{\epsilon}{r} \right) \right)\right). \quad (5)$$

Bibliography

1. Bader DA (1996) On the design and analysis of practical parallel algorithms for combinatorial problems with applications to image processing, PhD thesis, Department of Electrical Engineering, University of Maryland, College Park
2. Bader DA, Helman DR, JáJá J (1995) Practical parallel algorithms for personalized communication and integer sorting. Technical Report CS-TR-3548 and UMIACS-TR-95-101, UMIACS and Electrical Engineering, University of Maryland, College Park
3. Bader DA, Helman DR, JáJá J (1996) Practical parallel algorithms for personalized communication and integer sorting. ACM J Exp Algorithms 1(3):1–42. www.jea.acm.org/1996/BaderPersonalized/
4. Bailey D, Barszcz E, Barton J, Browning D, Carter R, Dagum L, Fatoohi R, Fineberg S, Frederickson P, Lasinski T, Schreiber R, Simon H, Venkatakrishnan V, Weeratunga S (1994) The NAS parallel benchmarks. Technical Report RNR-94-007, Numerical Aerodynamic Simulation Facility. NASA Ames Research Center, Moffett Field
5. JáJá J (1992) An Introduction to Parallel Algorithms. Addison-Wesley Publishing Company, New York
6. Knuth DE (1973) The Art of Computer Programming: Sorting and Searching, vol 3. Addison-Wesley Publishing Company, Reading
7. Message Passing Interface Forum. MPI (1995) A message-passing interface standard. Technical report. University of Tennessee, Knoxville. Version 1.1
8. Woo SC, Ohara M, Torrie E, Singh JP, Gupta A (1995) The SPLASH-2 programs: Characterization and methodological considerations. In Proceeding of the 22nd Annual Int'l Symposium Computer Architecture, pp 24–36

Hypercube

- ▶ [Hypercubes and Meshes](#)
- ▶ [Networks, Direct](#)

Hypercubes and Meshes

THOMAS M. STRICKER
Zürich, Switzerland

Synonyms

Distributed computer; Distributed memory computers; Generalized meshes and tori; Hypercube; Interconnection network; k-ary n-cube; Mesh; Multicomputers; Multi-processor networks

Definition

In the specific context of computer architecture, a *hypercube* refers to a parallel computer with a common regular interconnect topology that specifies the layout of processing elements and the wiring in between them. The etymology of the term suggests that a hypercube is an unbounded, higher dimensional cube alike geometric structure, that is scaled beyond (greek “hyper”) the three dimensions of a platonic cube (greek “kubos”). In its broader meaning, the term is also commonly used to denote a genre of supercomputer-prototypes and supercomputer-products, that were designed and built in the time period of 1980–1995, including the *Cosmic Cube*, the *Intel iPSC* hypercube, the *FPS T-Series*, and a similar machine manufactured by *nCUBE* corporation. A mesh-connected parallel computer is using a regular interconnect topology with an array of multiple processing elements in one, two, or three dimensions. In a generalization from hypercubes and meshes to the broader class of *k-ary n-cubes*, the concept extends to many more distributed-memory multi-computers with highly regular, direct networks.

Discussion

Introduction and Technical Background

In a *distributed memory multicomputer* with a *direct network*, the processing elements also serve as switching nodes in the network of wires connecting them. For a mathematical abstraction, the arrangement of processors and wires in a parallel machine is commonly expressed as a graph of nodes of processing elements

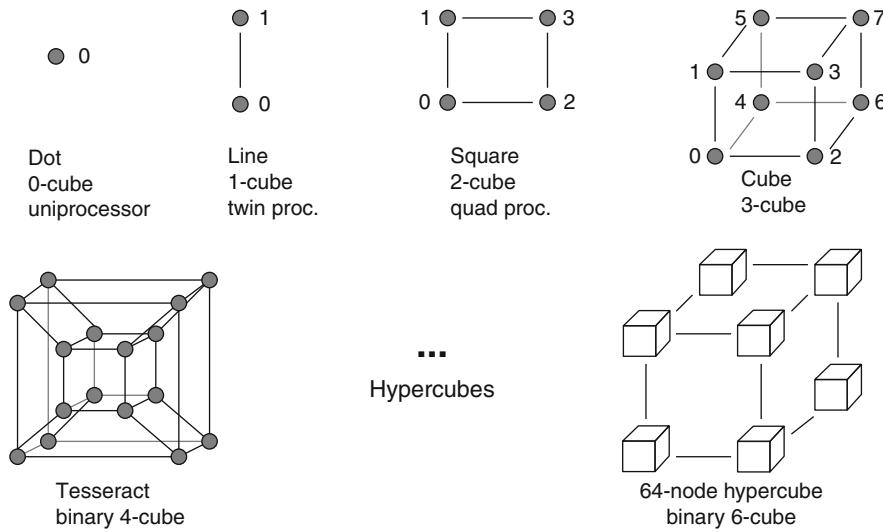
(vertices) and interconnect wires (edges) that are connecting the processors. Since high-speed data transfers normally require an unidirectional point-to-point connection, the resulting interconnection graphs are directed. In addition to the abstract connectivity graph, the optimal layout of processing elements and wires in physical space must be studied. In reference to the branch of mathematics dealing with invariant geometric properties in relation to different spaces, this specification is called the *topology* of a parallel computer system.

In geometry, the classical term *cube* refers to the regular convex hexahedron as one of the five platonic solids in three-dimensional space. Cube alike structures, beneath and beyond the three dimensions of a physical cube can be listed as follows and are drawn in Fig. 1.

- *Zero dimensional*: A dot in geometry or a non-connected uniprocessor in parallel computing.
- *One dimensional*: A geometric line segment or a twin processor system, connected with two unidirectional wires.
- *Two dimensional*: A geometric square or a four processor array, connected with four unidirectional wires in the horizontal and four wires in the vertical direction.
- *Three dimensional*: A geometric cube (hexahedron) or eight processors connected with 24 wires.
- *Four dimensional*: A tesseract in geometry or sixteen processors, arranged as two cubes with the eight corresponding vertices linked by two unidirectional wires each.
- *n-dimensional*: A hypercube as an abstract geometric figure, that becomes increasingly difficult to draw on paper or the arrangement of 2^n processor with each processor having n wires to its n immediate neighbors in all n dimensions.

History

With the evolution of supercomputers from a single vector processor toward a collection of high performance microprocessors during the mid-1980s, a large amount of research work focused on the best possible topology for massively parallel machines. In that time period the majority of parallel systems were built from a large



Hypercubes and Meshes. Fig. 1 Graphical rendering of dot (0 dimensional), line (1D), square (2D), cube (3D), and hypercubes (>3D)

number of identical processing elements, each comprising a standard microprocessor, a dedicated semiconductor random access memory, a network interface and – in some cases – even a dedicated local storage disk in every node. All processing elements also served as switching points in the interconnect fabric. This evolution of technology in highly parallel computers resulted in the class of *distributed memory parallel computers* or *multi-computers*.

The extensive research activity goes back to a special focus on geometry in the theory of parallel algorithms pre-dating the development of the first practical parallel systems. This led to a widespread belief, that the physical topology of parallel computing architectures had to reflect the communication pattern of the most common parallel algorithms known at the time. The obvious mathematical treatment of the question through graph theory resulted in countless theoretical papers that established many new topologies, mappings, emulations, and entire equivalence classes of topologies including meshes, hypercubes, and fat trees. The suggestions for a physical layout to connect the nodes of a parallel multicomputer range from simple one-dimensional arrays of processors to some fully interconnected graphs with direct wires from every node to every other node in the system. Hierarchical rings and busses were also considered. The many results

of the effort are compiled into a comprehensive text book [1].

During the golden age of the practical hypercubes (roughly during the years of 1985–1995), it was assumed that the topology of high-dimensional binary hypercubes would result in the best network for the direct mapping of many well-known parallel algorithms. In a binary hypercube each dimension is populated with just two processing nodes as shown in Fig. 1. The wiring of $P * \log_2 P$ unidirectional wires to connect P processors in hypercube topology seemed to be an optimal compromise between the minimal interconnect of just P wires for P processors in a ring and the complete interconnect with $P * (P-1)$ wires in a fully connected graph (clique).

In a hypercube or a mesh layout for a parallel system, the processors can only communicate data directly to a subset of neighboring processors and therefore require a mechanism for indirect communication through data forwarding at the intermediate processing nodes. Indirect communication can take place along pre-configured virtual channels or through the use of forwarded messages. Accordingly the practical hypercube and mesh systems are primarily designed for *message passing* as their programming model. However, most stream-based programming models can be supported efficiently with channel virtualization. Shared

memory programming models are often supported on top of a low level message passing software layer or by hardware mechanisms like directory-based caches that rely on messages for communication.

Binary hypercube parallel computers like the *Cosmic Cube* [7], the *iPSC* [8, 9] the *FPS T Series* [10] or *NCube* [11] were succeeded around 1992 by a series of distributed memory multi-computers with a two- or three-dimensional mesh/torus topology, such as the *Ametek Warp/iWarp*, *MassPar*, *J-Machine*, the Intel *Paragon*, Intel *ASCI Red*, *Cray T3D*, *T3E*, *Red Storm*, and *XT4/Seastar*. Scaling to larger machines, the early binary hypercubes were at a severe disadvantage, because they required a high number of dimensions resulting in an unbounded number of ports for the switches at every node (due to the *unbounded in/out degree* of the hypercube topology). Something all the hypercubes and the more advanced mesh/torus architectures have in common is that they rely heavily on the message routing technologies developed for the early hypercubes. They are classified as *generalized hypercubes (k -ary n -cubes)* in the literature and therefore included in this discussion. By the turn of the century the concept of hypercubes was definitely superseded by the *network of workstations* (NOWs) and by the *cluster of personal computers* (COPs) that used a more or less irregular interconnect fabric built with dedicated network switches that are readily available from the global working technology of the Internet.

Significance of Parallel Algorithms and Their Communication Patterns

A valid statement about the optimal interconnect layout in a parallel system can only be made under the assumption that there is a best suitable mapping between the M data elements of a large simulation problem and the P processing elements in a parallel computer. For simple, linear mappings (e.g., M/P elements on every processor in block or block/cyclic distribution) the communication patterns of operations between data elements translate into roughly the same communication pattern of messages between the processors in the parallel machine. For simplicity, we also assume $M \gg P$ and that M and P are powers of two. In an algorithm with

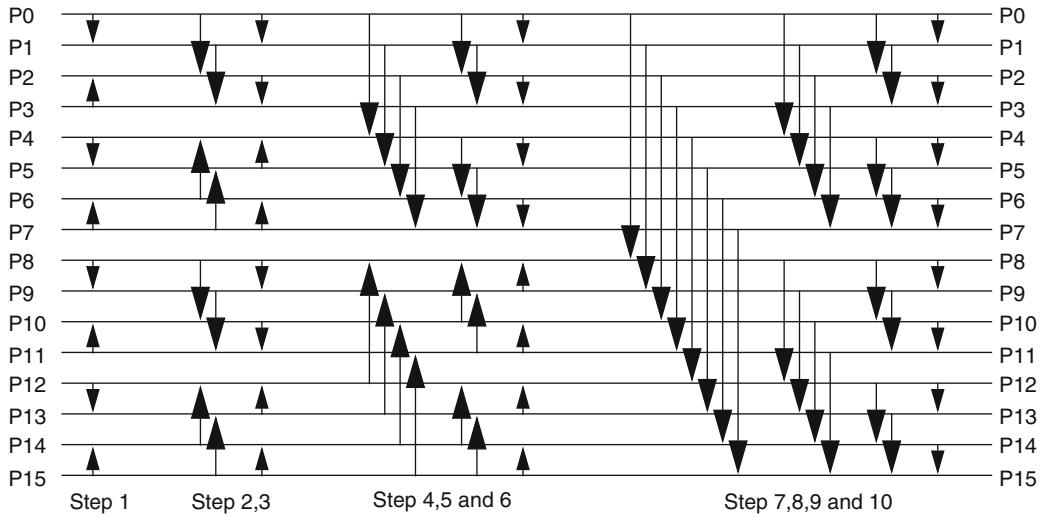
a hypercube communication pattern all communication activities take place between nodes that are direct neighbors in the layout of a hypercube. For the common number scheme of nodes this is between node pairs that differ by exactly one bit in their number. Every communication in a hypercube pattern therefore crosses just one wire in the hypercube machine.

Algorithms with Pure Hypercube Communication Patterns

Several well-known and important algorithms do require a sequence of inter-node communication steps that follow a pure hypercube communication pattern. In most of these algorithms, the communication happens in steps along increasing or decreasing dimensions, e.g., the first communication step between neighbors whose number differs in the least significant bit followed by a next step until a last step with communication to neighbors differing in the most significant bit. Most parallel computers cannot use more than one or two links for communication at the same time due to bandwidth constraints within the processor node. Therefore, a true pipeline with more than one or two overlapping hypercube communication steps is rarely encountered in practical programs.

One of the earliest and best-known algorithms with a pure hypercube communication pattern is the *bitonic sorting algorithm* that goes back to a generalized merge sort algorithm for parallel computers published in 1968 [3]. The communication graph in Fig. 2 shows all the hypercube communication steps that are required to merge locally sorted subsequences into a globally sorted sequence on 16 processors.

The most significant and most popular algorithm communicating between nodes in a pure hypercube pattern is the classic calculation of a *Fast Fourier Transform (FFT)* over an one-dimensional array, with its data distributed among P processors in a simple block partitioning. During the Fourier transformation of a large distributed array a certain number of butterfly calculations can be carried out in local memory up to the point when the distance between the elements in the butterfly becomes large enough that processor boundaries are crossed and some inter-processor communication over



Hypercubes and Meshes. Fig. 2 Batcher bitonic parallel sorting algorithm. A series of hypercube communication steps will merge 16 locally sorted subsequences into one globally sorted array. The arrows indicate the direction of the merge, i.e., one processor gets the upper and the other processor gets the lower half of the merged sequence

the hypercube links is required. The distance of butterfly operations is always a power of two and therefore it maps directly to a hypercube.

In the more advanced and multi-dimensional FFT library routines the communication operations between elements at the larger distances might eventually be reduced by dynamically rearranging the distribution of array elements during the calculation. Such a redistribution is typically performed by global data transpose operation that requires an *all-to-all personalized communication* (AAPC), i.e., individual and distinct data blocks are exchanged among all P processors. In general, fully efficient and scalable AAPC does require a scalable bandwidth interconnect, like the binary hypercube. In practice a talented programmer can optimize such collective communication primitives to run at maximal link speed for sufficiently large mesh- and torus-connected multi-computers (i.e., Cray T3D/T3E tori with up to 512 nodes) [5].

Algorithms with Next Neighbor Communication Pattern

Most simulations in computational physics, chemistry, and biology are calculating some physical interactions (forces) between a large number of model nodes (i.e., particles or grid points) in three-dimensional physical space. Therefore, they only require a limited amount

of communication between the model nodes in at most three dimensions because the longer range forces can be summarized or omitted. Consequently the communication in the parallel code of an equation solver, based on relaxation techniques is also limited to nearby processor in two or three dimensions.

Many calculations in natural science do not require communication in dimensions that are higher than three and there is no need to arrange processing elements of a parallel computer in hypercube topologies. Users in the field of scientific computing have recently introduced some optimizations that require increased connectivity. Modern physical simulations are carried out on *irregular meshes*, that are modeling the physical space more accurately by adapting the density of meshing points to some physical key parameter like the density of the material, the electric field, or a current flow. Irregular meshes grid points and sparse matrices are more difficult to partition into regular parallel machines. Advanced simulation methods also involve the summation of certain particle interactions and forces in Fourier and Laplace spaces to achieve a better accuracy in fewer simulation steps (e.g., the particle-mesh Ewald summation in molecular dynamics). Both improvements result in denser communication patterns that require higher connectivity. In the end it remains fairly hard to judge for the systems architect, whether

these codes alone can justify the more complex wiring of a high-dimensional hypercube within the range of the machine sizes, that are actually purchased by the customers.

Meshes as a Generalization of Binary Hypercubes into k -ary n -Cubes

In the original form of a binary hypercube, the wires connect only two processing nodes in every dimension. In linear arrays and rings a larger number of processors can be connected in every dimension. The combination of multiple dimensions and multiple processors connected along a line in one dimension leads to a natural generalization of the hypercube concept that includes the popular 1D, 2D, and 3D arrays of processing elements. In computer architecture, the term “mesh” is used for a two- or higher-dimensional processor array. A linear array of processing nodes can be wired into a ring by a wrap-around link. A two or higher dimensional mesh, that is equipped with wrap-around links is called a torus.

The generalized form of hypercubes and meshes is characterized by two parameters, the maximal number of elements found along one dimension, k , and the total number of dimensions used, n . The binary hypercubes described in the introduction are classified as a **2-ary n -cubes**. A $k \times k$, two-dimensional torus can be classified as a **k -ary 2-cube**.

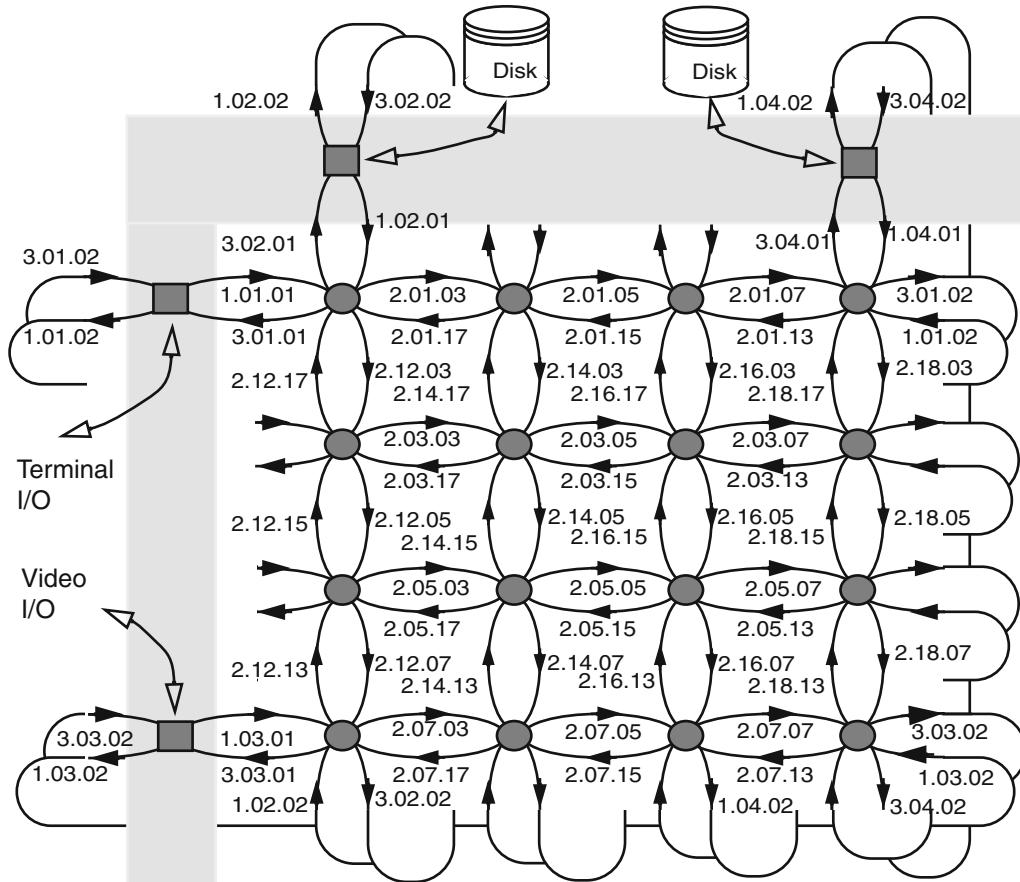
With the popularity of the hypercube topology in the beginning of distributed memory computing, many key technologies for message forwarding and routing were developed in the early hypercube machines. These ideas extend quite easily to the generalized k -ary n -cubes and were incorporated quickly into meshes and tori. In hypercubes, meshes and tori alike, the messages traveling between arbitrary processing nodes have to traverse multiple links to reach their destination. They have to be forwarded in a number of intermediate nodes, raising the important issues of *message forwarding policies* and *deadlock avoidance*.

In a general setting it would be very hard to establish a global guarantee that every message can travel right through without any delay along a given route. Therefore, some low level flow control mechanisms are used to stop and hold the messages, if another message is already occupying a channel along the path. The most

common strategies for forwarding the messages in a hypercube network are *wormhole* or *cut-through routing*. If not done carefully, blocking and delaying messages along the path can lead to tricky deadlock situations, in particular when the messages are indefinitely blocking each other within an interconnect fabric. During the era of the first hypercube multicomputers two key technologies for deadlock avoidance were described [7] and applied to the construction of prototypes and products:

- *Dimension order routing*: In binary hypercubes routes between two nodes must be chosen in a way that the distance in the highest dimension is traveled first, before any distance of a lesser dimension is traveled. Therefore messages can only be stopped due to a busy channel in a dimension lower than the one that they are traveling. The channels in the lowest dimension always lead to a final destination. The messages occupying these channels can make progress immediately and free the resource for other messages blocked at the same and higher dimensions. This is sufficient to prevent a routing deadlock.
- *Dateline-based torus routing*: With the wrap-around link of rings and tori, the case of messages blocking each other in a single dimension around the back-loops has to be addressed. This is done by replicating the physical wires into some parallel virtual channels forming a higher and a lower virtual ring. A dateline is established at a particular position in the ring. The dateline can only be crossed if the message switches from the higher to the lower ring at that position. With this simple trick, all connections along the ring remain possible, but the messages in the ring can no longer block each other in a circular deadlock.

Both *deadlock avoiding* techniques are normally combined to permit deadlock-free routing in the generalized k -ary n -cube topologies. Extending the well-known dimension-order and dateline routing strategies to slightly irregular network topologies is an additional challenge. The two abstract rules for legal message routes can be translated into a simple channel numbering scheme providing a total order of all channels within an interconnect, including all irregular nodes. A simple



Hypercubes and Meshes. Fig. 3 Generalized hypercubes. Channel numbering based on the original hypercube routing rules for the validation of a deadlock-free router in a slightly irregular two-dimensional torus with several IO nodes (an iWarp system)

rule that channels must be followed in a strictly increasing/decreasing order to form a legal route is sufficient to prevent deadlock. With the technique, illustrated in Fig. 3, it becomes possible to code and validate the router code for any k -ary 2-cube configuration that was offered as part of the Intel/CMU *iWarp* product line, including the service nodes that are arbitrarily inserted into the otherwise regular torus network [6].

Mapping Hypercube Algorithms into Meshes and Tori

By design the binary hypercube topology provides a fully scalable *worst case bisection bandwidth*. Regardless of how the machine is scaled or partitioned, the algorithm can count on the same amount of bandwidth per processor for the communication between the two

halves. In meshes and tori the total bisection bandwidth does not scale up linearly with the number of nodes. The global operations, that are usually communication bound, become increasingly costly in larger machines. The concept of the scalable bisection bandwidth was thought to be a key advantage of the hypercube designs that is not present in meshes for a long time.

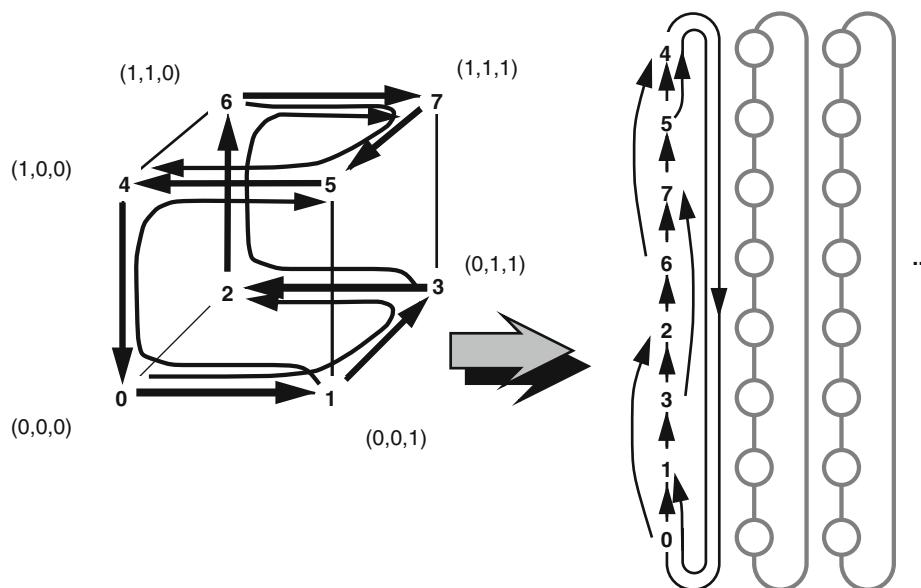
In the theory of parallel algorithms the asymptotic lower bounds on communication in binary hypercubes usually differ from the bounds holding for communication in an one-, two- or three-dimensional mesh/torus. The lower bound on the number of steps (i.e., the time complexity) is determined by the amount of data that has to be moved between the processing elements and by the number of wires that are available for this task (i.e., the bisection bandwidth). The upper bound is

usually determined by the sophistication of the algorithm. One of the most studied algorithms is *parallel sorting* in a 1-1 comparison based model [2, 3]. Similar considerations can be made for *matrix transposes* or *redistributions of block/cyclic arrays* distributed across the nodes of a parallel machine.

In the practice of parallel computing, a mesh-connected machine must be fairly large until an algorithmic limitation due to bisection bandwidth is encountered and a slowdown over a properly hypercube connected machine can be observed [5]. As an example, the bandwidth of an 8-node ring, a 64-node two-dimensional torus or a 512-node three-dimensional torus is sufficient to permit arbitrary array transpose operations without any slowdown over a hypercube of the same size. The comparison assumes that the processing nodes of both machines can transfer data from and to the network at about the speed of a bidirectional link – as this is the case for most parallel machines. Therefore, a few simple mapping techniques are used in practice to implement higher dimensional binary hypercubes on k-ary meshes or tori. A three-dimensional binary hypercube can be mapped into an eight node ring with a small performance penalty using a *Gray code mapping* as illustrated in Fig. 4.

In this simple mapping some links have to be shared or multiplexed by a factor of two and next-neighbor communication is extended to a distance of up to three hops. The mapping of larger virtual hypercube networks comes at the cost of an increasing dilation (increase in link distance) and congestion (a multiplexing factor of each link) for higher dimensions. The congestion factor can be derived using the calculations of bisection bandwidth for both topologies. A binary 6-cube can be mapped into a 2D torus and the 9-cube into a 3D torus by extending the linear scheme to multiple dimensions.

In their VLSI implementation, a two- or three-dimensional mesh/torus-machine is much easier to lay out in an integrated circuit than an eight- or ten-dimensional hypercube with its complicated wiring structure. Therefore its simple next-neighbor links can be built with faster wires and wider channels for multiple wires. The resulting advantage for the implementation is known to offset the congestion and dilation factors in practice. It is therefore highly interesting to study these technology trade-offs in practical machines. For 1-1 sorting, the algorithmic complexity and the measured execution times on a virtual hypercube mapped to mesh match the known complexities and



Hypercubes and Meshes. Fig. 4 Gray code mapping of a binary three-cube into an eight node ring. Virtual network channels were used to implement virtual hypercube network in machines that are actually wired as meshes or tori

running times of the algorithm for direct execution on a mesh fairly closely [2, 4].

Limitations of the Hypercube Machines

The ideal topology of a binary hypercube comes at the high cost of significant disadvantages and restrictions regarding their practical implementation in the VLSI circuits of a parallel computer. Therefore, a number of alternative options to build multicomputers were developed beyond hypercubes, meshes, and their superclass of the k -ary n -cubes.

Among the alternatives studied are: *hierarchical rings*, that differ slightly from multi-dimensional tori, *cube-connected cycles*, that successfully combine the scalable bisection bandwidth of hypercubes with the bounded in/out degrees of lower - dimensional meshes and finally *fat trees* that are provably optimal in their layout of wiring for VLSI. Those interconnect structures have in common, that they are still fairly regular and can leverage of the same practical technical solutions for message forwarding, routing and link level flow control, the way the traditional hypercubes do. None of these regular interconnects requires a spanning routing protocol or TCP/IP connections to function properly.

Towards the end of the golden age of hypercubes, the regular, direct networks were facing an increasing competition by new types of networks built either from physical links with different strength (i.e., different link bandwidths implemented by a multiplication of wires or in a different technology). In 1991, the Thinking Machines Corporation announced the Connection Machine CM5 using a fat tree topology and this announcement contributed to end of the hypercube age. After the year 2000, the physical network topologies became less and less important in supercomputer architecture. The detailed structure of the networks became hidden and de-emphasized due to many higher-level abstractions of parallelizing compilers and message passing libraries available to the programmers.

The availability of high performance commodity networking switches for use in the highly irregular topology of the global Internet accelerated this trend. In most newer designs, a fairly large multistage network of switches (e.g., a Clos network) has replaced the older direct networks of hypercubes and meshes. At this time only a small fraction of PC Clusters is still including

the luxury of a low latency, high performance interconnect. Despite all trends to abstraction and de-emphasis of network structure, the key figures of *message forwarding latency* and the *bisection bandwidth* remain the most important measure of scalability in a parallel computer architecture.

Hypercube Machine Prototypes and Products

One of the earliest practical computer systems using a hypercube topology is the *Cosmic Cube* prototype designed by the Group of C. Seitz at the California Institute of Technology [7]. The system was developed in the first half of the 1980s using the VLSI integration technologies newly available at the time. The first version of the system comprised 64 nodes with an Intel 8086/8087 microprocessor/floating point coprocessors combination running at 5 MHz and using 128 kB of memory in each node. The nodes were connected with point-to-point links running at 2 Mbit/s nominal speed. The original system of 64 nodes was allegedly planned as a $4 \times 4 \times 4$ three-dimensional torus with bidirectional links – but this particular topology is fully equivalent to a binary six-dimensional hypercube under a gray code mapping and became therefore known as a first *hypercube multicomputer*, rather than as a first three-dimensional torus. Subsequently a small number of commercial machines were built in a collaboration with a division of AMETEK Corporation [15].

The Caltech prototypes lead to the development of iPSC at the Intel Supercomputer Systems Division, a commercial product series based on hypercube topology [8, 9]. An example of the typical technical specifications is the iPSC/2 system as released in 1987: 64 nodes, Intel 80386/387 processor/floating point coprocessor running at 16 MHz, 4–16 MB of main memory in each node, the nodes connected with links running 22 Mbit/s. iPSC systems were programmed using the NX message passing library, similar in the functionality, but pre-dating the popular portable message passing systems like PVM or MPI. The development of the product line was strongly supported by (D)ARPA, the Advanced Research Project Agency of the US Dept. of Defense.

During roughly the same time period a hypercube machine was also offered by NCube Corporation, a super-computing vendor that was fully dedicated to the development of hypercube systems at the time. The

NCube 6,400 model, available to Nasa AMES in 1991 was a 64-node system with a 20 MHz full custom 64 bit CPU/FPU with up to 64 MB main memory in every node and 20 Mbit/s interconnects. The largest configuration commercially offered was a binary 10-cube with 1,024 processing nodes [11, 12].

The *FPS (Floating Point Systems)* T Series Parallel Vector Supercomputer was also made of multiple processing nodes interconnected as a binary n-cube. Each node is implemented on a single printed circuit board, contains 1 MB of data storage memory, and has a peak vector computational speed of 12 MFLOPS. Eight vector nodes are grouped around a system support node and a disk storage system to form a module. The module is the basic building block for larger system configurations in hypercube topology. The T series was named after the tesseract, a geometric cube in four-dimensional space [10].

The *Thinking Machine* CM2 also included a hypercube communication facility for data exchanges between the nodes as well as a mesh (called NEWS grid) [13]. As an SIMD machine with a large number of bit slice processors, its architecture differed significantly from the classical hypercube multicomputer. In its successor, the *Thinking Machine* CM5, the direct networks were given up in favor of a fat tree using a variable number of (typically N-1) of additional switches to form a network in fat tree topology. A similar trend was followed in the machines of *Meiko/Quadrics*. The delay of the T9000 transputer chips with processing and communication capabilities on a single chip forced the designers of the CS2 system to build their communication system with a multi-stage fabric of switches instead of a hypercube interconnect. The multicomputer line by IBM, the IBM SP1 and its follow-on products also used a multi-stage switch fabric. Therefore, these architectures do no longer qualify as generalized hypercubes with direct networks.

It is worth mentioning that the first *Beowulf* clusters of commodity PCs were equipped with three to five network interfaces that could be wired directly in hypercube topology. Routing of messages between the links was achieved by system software in the ethernet drivers of the LINUX operating system. The communication benchmark presented in the first beowulf paper were measured on an 8-node system, wired as 2^3 cube [14]. The author also remembers encountering a

Beowulf system on display at the Supercomputing trade show wired as a 32-node binary 5-cube using multiple 100BaseT network interface cards and simple CAT5 crossover cables.

But as mentioned before, the designs of processors with their own communication hardware on board and direct networks were quickly replaced by commodity switches manufactured by a large number of network equipment vendors in the Internet world. So the topology of the parallel system suddenly became a secondary issue. A large variety of different network configurations rapidly succeeded the regular Beowulf systems in the growing market for clusters of commodity PCs.

H

Research Conferences on Hypercubes

The popularity of hypercube architectures in the second half of the 1980s led to several instances of a computer architecture conference dedicated solely to distributed memory multicomputers with hypercube and mesh topologies. The locations and dates of the conferences are as remembered or collected from citations in subsequent papers:

- First Hypercube Conference in Knoxville, August 26–27, 1985, with proceedings published by SIAM after the conference.
- Second Conference on Hypercube Multiprocessors, Knoxville, TN, September 29–October 1, 1986, with proceedings published by SIAM after the conference.
- Third Conference on Hypercube Concurrent Computers and Applications, Pasadena, CA, January 19–20, 1988 with proceedings published by the ACM, New York.
- Fourth Conference on Hypercube Concurrent Computers and Applications, Monterrey, CA, March 6–9, 1989, proceedings publisher unknown.

and later giving in to the trend that the architecture as distributed memory computer is more important than the hypercube layout:

- Fifth Distributed Memory Computing Conference, Charleston, SC, April 8–12, 1990 with proceedings published by IEEE Computer Society Press.
- Sixth Distributed Memory Computing Conference, Portland, OR, April 29–May 1, 1991 with proceedings published by IEEE Computer Society Press.

- First European Distributed Memory Computing, EDMCC, held in Rennes, France.
- Second European Distributed Memory Computing, EDMCC2, Munich, FRG, April 22–24, 1991 with Proceedings by Springer, Lecture Notes in Computer Science.

After these rather successful meetings of computer architects and application programmers, the conference series on hypercubes and distributed memory parallel systems lost its momentum and in 1992, the specialized venues were unfortunately discontinued.

Related Entries

- [Beowulf clusters](#)
- [Bitonic Sort](#)
- [Clusters](#)
- [Connection Machine](#)
- [Cray T3E](#)
- [Cray Vector Computers](#)
- [Cray XT4 and Seastar 3-D Torus Interconnect](#)
- [Distributed-Memory Multiprocessor](#)
- [Fast Fourier Transform \(FFT\)](#)
- [IBM RS/6000 SP](#)
- [Interconnection Networks](#)
- [MasPar](#)
- [MPI \(Message Passing Interface\)](#)
- [MPP](#)
- [nCUBE](#)
- [Networks, Direct](#)
- [Routing \(including Deadlock Avoidance\)](#)
- [Sorting](#)
- [Warp and IWarp](#)

Bibliographic Notes and Further Reading

A most detailed description of the algorithms optimally suited for hypercubes, the classes of hypercube equivalent networks, and the emulation of different topologies with respect to algorithmic models is given in an 835 page textbook by F.T. Leighton that appeared in 1991 [1]. A detailed description of the network topologies and the technical data of all practical hypercube prototypes built and machines commercially sold between 1985 and 1995 can be found through Google Scholar in the numerous papers written by the architects working for the computer manufacturers or by researches at the

different US national labs evaluating and benchmarking these distributed memory multicomputers. The most interesting study dedicated entirely to binary hypercubes appeared in 1991 in Parallel Computing [12]. The visit to the trade show of “Supercomputing” during the “hypercube” years was a memorable experience, because countless new distributed memory system vendors surprised the audience with a new parallel computer architecture every year. Most of them contained some important innovation and can still be admired in the permanent collections of the computer museums in Boston (MA), Mountain View (CA), and Paderborn (Germany).

Bibliography

1. Leighton FT (1991) Introduction to parallel algorithms and architectures: array, trees, hypercubes. Morgan Kaufmann Publishers, San Francisco 837p, ISBN:1-55860-117-1
2. Stricker T (1992) Supporting the hypercube programming model on meshes (a fast parallel sorter for iwarp). In: Proceedings of the symposium for parallel algorithms and architectures, pp 148–157, San Diego, June 1992
3. Batcher KE (1968) Sorting networks and their applications. In: Proceedings of the american federation of information processing societies spring joint computer conference, vol 32. AFIPS Press, Montvale, pp 307–314
4. Nassimi D, Sahni S (1979) Bitonic sort on a mesh-connected parallel computer. In: IEEE TransComput 28(1):2–7
5. Hinrichs S, Kosak C, O'Hallaron D, Stricker T, Take R (1994) Optimal all-to-all personalized communication in meshes and tori. In: Proceedings of the symposium of parallel algorithms and architectures, ACM SPAA'94, Cape May, Jan 1994
6. Stricker T (1991) Message routing in irregular meshes and tori. In: Proceedings of the 6th IEEE distributed memory computing conference, DMCC, Portland, May 1991
7. Seitz CL (1985) The cosmic cube. Commun ACM 28(1):22–33
8. Close P (1988) The iPSC/2 node architecture. In: Proceedings of the third conference on hypercube concurrent computers and applications, Pasadena, 19–20 Jan 1988, pp 43–50
9. Nugent SF (1988) The iPSC/2 direct-connect communications technology. In: Proceedings of the third conference on hypercube concurrent computers and applications, Pasadena, 19–20 Jan 1988, pp 51–60
10. Hawkinson S (1988) The FPS T series supercomputer, system modelling and optimization. In: Lecture notes in control and information sciences, vol 113/1988. Springer, Berlin/Heidelberg, pp 766–785
11. The nCUBE Handbook, Beaverton OR, 1986 and the nCUBE 6400 processor, user manual, Beaverton
12. Dunigan TH (1991) Performance of the Intel iPSC 1/2/860 and Ncube 3200/6400 hypercubes, parallel computing 17. North Holland, Elsevier, pp 1285–1302

13. Tucker LW, Robertson GG (1988) Architecture and applications of the connection machines. *IEEE Comput* 21(8):26–38
14. Becker J, Sterling D, Savarese T, Dorband JE, Ranawake UA, Packer CV (1995) Beowulf: a parallel workstation for scientific computation. In: Proceedings of ICPP workshop on challenges for parallel processing, CRC Press, Oconomowc, August 1995
15. Seitz CL, Athas W, Flraig C, Martin A, Seieovic J, Steele CS, Su WK (1988) The architecture and programming of the ametek series 2010 multicomputer. In: Proceedings of the third conference on hypercube concurrent computers and applications, Pasadena, 19–20 Jan 1988, pp 31–36

Formal Definition of Hypergraph Partitioning

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is defined as a set of vertices (cells) \mathcal{V} and a set of nets (hyperedges) \mathcal{N} among those vertices. Every net $n \in \mathcal{N}$ is a subset of vertices, that is, $n \subseteq \mathcal{V}$. The vertices in a net n are called its *pins*. The size of a net is equal to the number of pins. The *degree* of a vertex is equal to the number of nets it is connected to. Graph is a special instance of hypergraph such that each net has exactly two pins. Vertices can be associated with weights, denoted with $w[\cdot]$, and nets can be associated with costs, denoted with $c[\cdot]$.

$\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \dots, \mathcal{V}_K\}$ is a *K-way partition* of \mathcal{H} if the following conditions hold:

- Each part \mathcal{V}_k is a nonempty subset of \mathcal{V} , that is, $\mathcal{V}_k \subseteq \mathcal{V}$ and $\mathcal{V}_k \neq \emptyset$ for $1 \leq k \leq K$
- Parts are pairwise disjoint, that is, $\mathcal{V}_k \cap \mathcal{V}_\ell = \emptyset$ for all $1 \leq k < \ell \leq K$
- Union of K parts is equal to \mathcal{V} , i.e., $\bigcup_{k=1}^K \mathcal{V}_k = \mathcal{V}$

In a partition Π of \mathcal{H} , a net that has at least one pin (vertex) in a part is said to *connect* that part. *Connectivity* λ_n of a net n denotes the number of parts connected by n . A net n is said to be *cut (external)* if it connects more than one part (i.e., $\lambda_n > 1$), and *uncut (internal)* otherwise (i.e., $\lambda_n = 1$). A partition is said to be balanced if each part \mathcal{V}_k satisfies the *balance criterion*:

$$W_k \leq W_{avg}(1 + \varepsilon), \quad \text{for } k = 1, 2, \dots, K. \quad (1)$$

In (1), weight W_k of a part \mathcal{V}_k is defined as the sum of the weights of the vertices in that part (i.e., $W_k = \sum_{v \in \mathcal{V}_k} w[v]$), W_{avg} denotes the weight of each part under the perfect load balance condition (i.e., $W_{avg} = (\sum_{v \in \mathcal{V}} w[v])/K$), and ε represents the predetermined maximum imbalance ratio allowed.

The set of external nets of a partition Π is denoted as \mathcal{N}_E . There are various [20] *cutsizes* definitions for representing the cost $\chi(\Pi)$ of a partition Π . Two relevant definitions are:

$$\chi(\Pi) = \sum_{n \in \mathcal{N}_E} c[n] \quad (2)$$

$$\chi(\Pi) = \sum_{n \in \mathcal{N}_E} c[n](\lambda_n - 1). \quad (3)$$

In (2), the cutsize is equal to the sum of the costs of the cut nets. In (3), each cut net n contributes $c[n](\lambda_n - 1)$ to the cutsize. The cutsize metrics given in (2) and (3)

Hypergraph Partitioning

ÜMIT V. ÇATALYÜREK¹, BORA UÇAR², CEVDET AYKANAT³

¹The Ohio State University, Columbus, OH, USA

²ENS Lyon, Lyon, France

³Bilkent University, Ankara, Turkey

Definition

Hypergraphs are generalization of graphs where each edge (hyperedge) can connect more than two vertices. In simple terms, the hypergraph partitioning problem can be defined as the task of dividing a hypergraph into two or more roughly equal-sized parts such that a cost function on the hyperedges connecting vertices in different parts is minimized.

Discussion

Introduction

During the last decade, hypergraph-based models gained wide acceptance in the parallel computing community for modeling various problems. By providing natural way to represent multiway interactions and unsymmetric dependencies, hypergraph can be used to elegantly model complex computational structures in parallel computing. Here, some concrete applications will be presented to show how hypergraph models can be used to cast a suitable scientific problem as an hypergraph partitioning problem. Some insights and general guidelines for using hypergraph partitioning methods in some general classes of problems are also given.

will be referred to here as *cut-net* and *connectivity* metrics, respectively. The hypergraph partitioning problem can be defined as the task of dividing a hypergraph into two or more parts such that the cutsize is minimized, while a given balance criterion (1) among part weights is maintained.

A recent variant of the above problem is the *multi-constraint hypergraph* partitioning [9, 18] in which each vertex has a vector of weights associated with it. The partitioning objective is the same as above, and the partitioning constraint is to satisfy a balancing constraint associated with each weight. Here, $w[v, i]$ denotes the C weights of a vertex v for $i = 1, \dots, C$. Hence, the balance criterion (1) can be rewritten as

$$W_{k,i} \leq W_{avg,i} (1 + \varepsilon) \text{ for } k = 1, \dots, K \text{ and } i = 1, \dots, C, \quad (4)$$

where the i th weight $W_{k,i}$ of a part \mathcal{V}_k is defined as the sum of the i th weights of the vertices in that part (i.e., $W_{k,i} = \sum_{v \in \mathcal{V}_k} w[v, i]$), and $W_{avg,i}$ is the average part weight for the i th weight (i.e., $W_{avg,i} = (\sum_{v \in \mathcal{V}} w[v, i]) / K$), and ε again represents the allowed imbalance ratio.

Another variant is the *hypergraph partitioning with fixed vertices*, in which some of the vertices are fixed in some parts before partitioning. In other words, in this problem, a *fixed-part* function is provided as an input to the problem. A vertex is said to be *free* if it is allowed to be in any part in the final partition, and it is said to be *fixed* in part k if it is required to be in \mathcal{V}_k in the final partition Π .

Yet another variant is *multi-objective hypergraph partitioning* in which there are several objectives to be minimized [1, 21]. Specifically, a given net contributes different costs to different objectives.

Sparse Matrix Partitioning

One of the most elaborated applications of hypergraph partitioning (HP) method in the parallel scientific computing domain is the parallelization of sparse matrix-vector multiply (SpMxV) operation. Repeated matrix-vector and matrix-transpose-vector multiplies that involve the same large, sparse matrix are the kernel operations in various iterative algorithms involving sparse linear systems. Such iterative algorithms include solvers for linear systems, eigenvalues, and linear programs. The pervasive use of such solvers motivates the

development of HP models and methods for efficient parallelization of SpMxV operations.

Before discussing the HP models and methods for parallelizing SpMxV operations, it is favorable to discuss parallel algorithms for SpMxV. Consider the matrix-vector multiply of the form $y \leftarrow Ax$, where the nonzeros of the sparse matrix A as well as the entries of the input and output vectors x and y are partitioned arbitrarily among the processors. Let $map(\cdot)$ denote the nonzero-to-processor and vector-entry-to-processor assignments induced by this partitioning. A parallel algorithm would execute the following steps at each processor P_k .

1. Send the local input-vector entries x_j , for all j with $map(x_j) = P_k$, to those processors that have at least one nonzero in column j .
2. Compute the scalar products $a_{ij}x_j$ for the local nonzeros, that is, the nonzeros for which $map(a_{ij}) = P_k$ and accumulate the results y_i^k for the same row index i .
3. Send local nonzero partial results y_i^k to the processor $map(y_i) \neq P_k$, for all nonzero y_i^k .
4. Add the partial y_i^k results received to compute the final results $y_i = \sum y_i^k$ for each i with $map(y_i) = P_k$.

As seen in the algorithm, it is necessary to have partitions on the matrix A and the input- and output-vectors x and y of the matrix-vector multiply operation. Finding a partition on the vectors x and y is referred to as the vector partitioning operation, and it can be performed in three different ways: by decoding the partition given on A ; in a post-processing step using the partition on the matrix; or explicitly partitioning the vectors during partitioning the matrix. In any of these cases, the vector partitioning for matrix-vector operations is called *symmetric* if x and y have the same partition, and *non-symmetric* otherwise. A vector partitioning is said to be *consistent*, if each vector entry is assigned to a processor that has at least one nonzero in the respective row or column of the matrix. The consistency is easy to achieve for the nonsymmetric vector partitioning; x_j can be assigned to any of the processors that has a nonzero in the column j , and y_i can be assigned to any of the processors that has a nonzero in the row i . If a symmetric vector partitioning is sought, then special care must be taken to assign a pair of matching input- and output-vector entries, e.g., x_i and

y_i , to a processor having nonzeros in both row and column i . In order to have such a processor for all vector entry pairs, the sparsity pattern of the matrix \mathbf{A} can be modified to have a zero-free diagonal. In such cases, a consistent vector partition is guaranteed to exist, because the processors that own the diagonal entries can also own the corresponding input- and output-vector entries; x_i and y_i can be assigned to the processor that holds the diagonal entry a_{ii} .

In order to achieve an efficient parallelism, the processors should have balanced computational load and the inter-processor communication cost should have been minimized. In order to have balanced computational load, it suffices to have almost equal number of nonzeros per processor so that each processor will perform almost equal number of scalar products, for example, $a_{ij}x_j$, in any given parallel system. The communication cost, however, has many components (the total volume of messages, the total number of messages, maximum volume/number of messages in a single processor, either in terms of sends or receives or both) each of which can be of utmost importance for a given matrix in a given parallel system. Although there are alternatives and more elaborate proposals, the most common communication cost metric addressed in hypergraph partitioning-based methods is the total volume of communication.

Loosely speaking, hypergraph partitioning-based methods for efficient parallelization of SpMxV model the data of the SpMxV (i.e., matrix and vector entries) with the vertices of a hypergraph. A partition on the vertices of the hypergraph is then interpreted in such a way that the data corresponding to a set of vertices in a part are assigned to a single processor. More accurately, there are two classes of hypergraph partitioning-based methods to parallelizing SpMxV. The methods in the first class build a hypergraph model representing the data and invoke a partitioning heuristic on the so-built hypergraph. The methods in this class can be said to be models rather than being algorithms. There are currently three main hypergraph models for representing sparse matrices, and hence there are three methods in this first class. These three main models are described below in the next section. Essential property of these models is that the cutsizes (3) of any given partition is equal to the total communication volume to be incurred under a consistent vector partitioning when the matrix

elements are distributed according to the vertex partition. The methods in the second class follow a mix-and-match approach and use the three main models, perhaps, along with multi-constraint and fixed-vertex variations in an algorithmic form. There are a number of methods in this second class, and one can develop many others according to application needs and matrix characteristics. Three common methods belonging to this class are described later, after the three main models. The main property of these algorithms is that the sum of the cutsizes of each application of hypergraph partitioning amounts to the total communication volume to be incurred under a consistent vector partitioning (currently these methods compute a vector partitioning after having found a matrix partitioning) when the matrix elements are distributed according to the vertex partitions found at the end.

Three Main Models for Matrix Partitioning

In the *column-net hypergraph model* [11] used for 1D rowwise partitioning, an $M \times N$ matrix \mathbf{A} with Z nonzeros is represented as a unit-cost hypergraph $\mathcal{H}_{\mathcal{R}} = (\mathcal{V}_{\mathcal{R}}, \mathcal{N}_{\mathcal{C}})$ with $|\mathcal{V}_{\mathcal{R}}| = M$ vertices, $|\mathcal{N}_{\mathcal{C}}| = N$ nets, and Z pins. In $\mathcal{H}_{\mathcal{R}}$, there exists one vertex $v_i \in \mathcal{V}_{\mathcal{R}}$ for each row i of matrix \mathbf{A} . Weight $w[v_i]$ of a vertex v_i is equal to the number of nonzeros in row i . The name of the model comes from the fact that columns are represented as nets. That is, there exists one unit-cost net $n_j \in \mathcal{N}_{\mathcal{C}}$ for each column j of matrix \mathbf{A} . Net n_j connects the vertices corresponding to the rows that have a nonzero in column j . That is, $v_i \in n_j$ if and only if $a_{ij} \neq 0$.

In the *row-net hypergraph model* [11] used for 1D columnwise partitioning, an $M \times N$ matrix \mathbf{A} with Z nonzeros is represented as a unit-cost hypergraph $\mathcal{H}_{\mathcal{C}} = (\mathcal{V}_{\mathcal{C}}, \mathcal{N}_{\mathcal{R}})$ with $|\mathcal{V}_{\mathcal{C}}| = N$ vertices, $|\mathcal{N}_{\mathcal{R}}| = M$ nets, and Z pins. In $\mathcal{H}_{\mathcal{C}}$, there exists one vertex $v_j \in \mathcal{V}_{\mathcal{C}}$ for each column j of matrix \mathbf{A} . Weight $w[v_j]$ of a vertex $v_j \in \mathcal{V}_{\mathcal{R}}$ is equal to the number of nonzeros in column j . The name of the model comes from the fact that rows are represented as nets. That is, there exists one unit-cost net $n_i \in \mathcal{N}_{\mathcal{R}}$ for each row i of matrix \mathbf{A} . Net n_i connects the vertices corresponding to the columns that have a nonzero in row i . That is, $v_j \in n_i$ if and only if $a_{ij} \neq 0$.

In the *column-row-net hypergraph model*, otherwise known as the *fine-grain model* [13], used for 2D

nonzero-based fine-grain partitioning, an $M \times N$ matrix \mathbf{A} with Z nonzeros is represented as a unit-weight and unit-cost hypergraph $\mathcal{H}_{\mathcal{Z}} = (\mathcal{V}_{\mathcal{Z}}, \mathcal{N}_{\mathcal{RC}})$ with $|\mathcal{V}_{\mathcal{Z}}| = Z$ vertices, $|\mathcal{N}_{\mathcal{RC}}| = M + N$ nets and $2Z$ pins. In $\mathcal{V}_{\mathcal{Z}}$, there exists one unit-weight vertex v_{ij} for each nonzero a_{ij} of matrix \mathbf{A} . The name of the model comes from the fact that both rows and columns are represented as nets. That is, in $\mathcal{N}_{\mathcal{RC}}$, there exist one unit-cost row-net r_i for each row i of matrix \mathbf{A} and one unit-cost column-net c_j for each column j of matrix \mathbf{A} . The row-net r_i connects the vertices corresponding to the nonzeros in row i of matrix \mathbf{A} , and the column-net c_j connects the vertices corresponding to the nonzeros in column j of matrix \mathbf{A} . That is, $v_{ij} \in r_i$ and $v_{ij} \in c_j$ if and only if $a_{ij} \neq 0$. Note that each vertex v_{ij} is in exactly two nets.

Some Other Methods for Matrix Partitioning

The *jagged-like partitioning method* [16] uses the row-net and column-net hypergraph models. It is an algorithm with two steps, in which each step models either the *expand phase* (the 1st line) or the *fold phase* (the 3rd line) of the parallel SpMxV algorithm given above. Therefore, there are two alternative schemes for this partitioning method. The one which models the expands in the first step and the folds in the second step is described below.

Given an $M \times N$ matrix \mathbf{A} and the number K of processors organized as a $P \times Q$ mesh, the jagged-like partitioning model proceeds as shown in Fig. 1. The algorithm has two main steps. First, \mathbf{A} is partitioned rowwise into P parts using the column-net hypergraph model $\mathcal{H}_{\mathcal{R}}$ (lines 1 and 2 of Fig. 1). Consider a P -way partition $\Pi_{\mathcal{R}}$ of $\mathcal{H}_{\mathcal{R}}$. From the partition $\Pi_{\mathcal{R}}$, one obtains P submatrices \mathbf{A}_p , for $p = 1, \dots, P$ each having roughly equal number of nonzeros. For each p , the rows of the submatrix \mathbf{A}_p correspond to the vertices in \mathcal{R}_p (lines 6 and 7 of Fig. 1). The submatrix \mathbf{A}_p is assigned to the p th row of the processor mesh. Second, each submatrix \mathbf{A}_p for $1 \leq p \leq P$ is independently partitioned columnwise into Q parts using the row-net hypergraph \mathcal{H}_p (lines 8 and 9 of Fig. 1). The nonzeros in the i th row of \mathbf{A} are partitioned among the Q processors in a row of the processor mesh. In particular, if $v_i \in \mathcal{R}_p$ at the end of line 2 of the algorithm, then the nonzeros in the i th row of \mathbf{A} are partitioned among the processors in the p th row of the processor mesh. After partitioning the submatrix \mathbf{A}_p columnwise, the *map* array contains the partition information for the nonzeros residing in \mathbf{A}_p .

For each i , the volume of communication required to fold the vector entry y_i is accurately represented as a part of “*foldVolume*” in the algorithm. For each j , the volume of communication regarding the vector entry x_j

JAGGED-LIKE-PARTITIONING ($\mathbf{A}, K = P \times Q, \varepsilon_1, \varepsilon_2$)

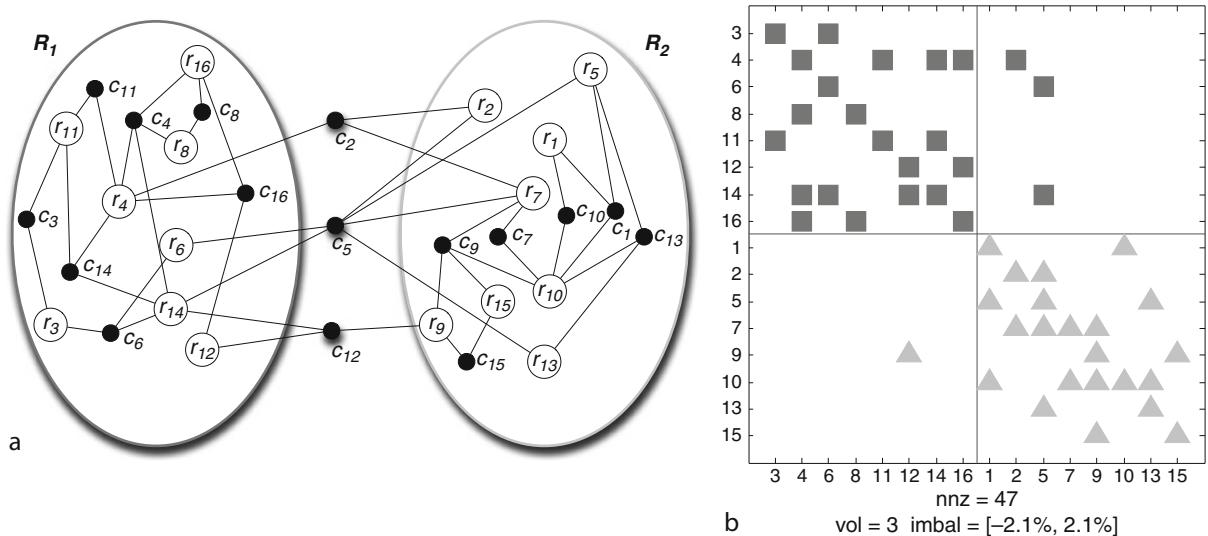
Input : a matrix \mathbf{A} , the number of processors $K = P \times Q$, and the imbalance ratios $\varepsilon_1, \varepsilon_2$.

Output: $map(a_{ij})$ for all $a_{ij} \neq 0$ and total Volume.

- ```

1: $\mathcal{H}_{\mathcal{R}} = (\mathcal{V}_{\mathcal{R}}, \mathcal{N}_{\mathcal{C}}) \leftarrow \text{columnNet}(\mathbf{A})$
2: $\Pi_{\mathcal{R}} = \{\mathcal{R}_1, \dots, \mathcal{R}_P\} \leftarrow \text{partition}(\mathcal{H}_{\mathcal{R}}, P, \varepsilon_1)$ ▷ rowwise partitioning of \mathbf{A}
3: expand Volume $\leftarrow \text{cutsize}(\Pi_{\mathcal{R}})$
4: foldVolume $\leftarrow 0$
5: for $p = 1$ to P do
6: $R_p = \{r_i : v_i \in \mathcal{R}_p\}$
7: $\mathbf{A}_p \leftarrow \mathbf{A}(R_p, :)$ ▷ submatrix indexed by rows R_p
8: $\mathcal{H}_p = (\mathcal{V}_p, \mathcal{N}_p) \leftarrow \text{rowNet}(\mathbf{A}_p)$
9: $\Pi_p^C = \{C_p^1, \dots, C_p^Q\} \leftarrow \text{partition}(\mathcal{H}_p, Q, \varepsilon_2)$ ▷ columnwise partitioning of \mathbf{A}_p
10: foldVolume $\leftarrow \text{foldVolume} + \text{cutsize}(\Pi_p^C)$
11: for all $a_{ij} \neq 0$ of \mathbf{A}_p do
12: $map(a_{ij}) = P_{p,q} \Leftrightarrow c_j \in C_p^q$
13: return totalVolume $\leftarrow \text{expandVolume} + \text{foldVolume}$

```
-



**Hypergraph Partitioning.** Fig. 2 First step of four-way jagged-like partitioning of a matrix; (a) two-way partitioning  $\Pi_R$  of column-net hypergraph representation  $\mathcal{H}_R$  of  $\mathbf{A}$ , (b) two-way rowwise partitioning of matrix  $\mathbf{A}^\Pi$  obtained by permuting  $\mathbf{A}$  according to the partitioning induced by  $\Pi$ ; the nonzeros in the same partition are shown with the same shape and color; the deviation of the minimum and maximum numbers of nonzeros of a part from the average are displayed as an interval  $\text{imbal}$ ;  $\text{vol}$  denotes the number of nonzeros and the total communication volume

is accurately represented as a part of “expandVolume” in the algorithm.

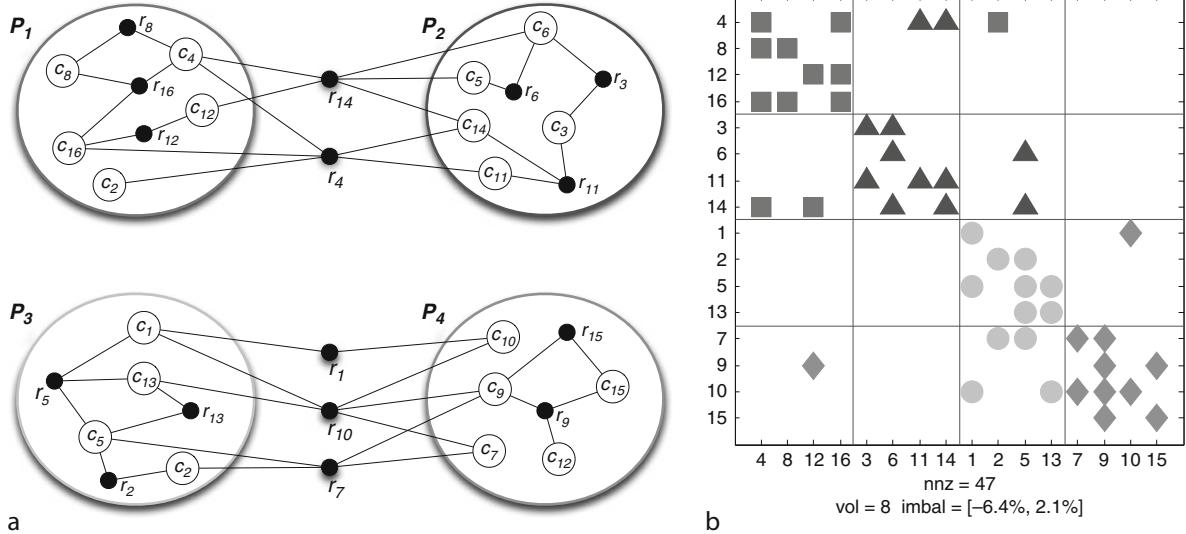
Figure 2a illustrates the column-net representation of a sample matrix to be partitioned among the processors of a  $2 \times 2$  mesh. For simplicity of the presentation, the vertices and the nets of the hypergraphs are labeled with letters “r” and “c” to denote the rows and columns of the matrix. The matrix is first partitioned rowwise into two parts, and each part is assigned to a row of the processor mesh, namely to processors  $\{P_1, P_2\}$  and  $\{P_3, P_4\}$ . The resulting permuted matrix is displayed in Fig. 2b. Figure 3a displays the two row-net hypergraphs corresponding to each submatrix  $\mathbf{A}_p$  for  $p = 1, 2$ . Each hypergraph is partitioned independently; sample partitions of these hypergraphs are also presented in this figure. As seen in the final symmetric permutation in Fig. 3b, the nonzeros of columns 2 and 5 are assigned to different parts, resulting  $P_3$  to communicate with both  $P_1$  and  $P_2$  in the expand phase.

The *checkerboard partitioning method* [14] is also a two-step method, in which each step models either the expand phase or the fold phase of the parallel SpMxV. Similar to jagged-like partitioning, there are two alternative schemes for this partitioning method. The one

which models the expands in the first step and the folds in the second step is presented below.

Given an  $M \times N$  matrix  $\mathbf{A}$  and the number  $K$  of processors organized as a  $P \times Q$  mesh, the checkerboard partitioning method proceeds as shown in Fig. 4. First,  $\mathbf{A}$  is partitioned rowwise into  $P$  parts using the column-net model (lines 1 and 2 of Fig. 4), producing  $\Pi_R = \{\mathcal{R}_1, \dots, \mathcal{R}_P\}$ . Note that this first step is exactly the same as that of the jagged-like partitioning. In the second step, the matrix  $\mathbf{A}$  is partitioned columnwise into  $Q$  parts by using the multi-constraint partitioning to obtain  $\Pi_C = \{\mathcal{C}_1, \dots, \mathcal{C}_Q\}$ . In comparison to the jagged-like method, in this second step the whole matrix  $\mathbf{A}$  is partitioned (lines 4 and 8 of Fig. 4), not the submatrices defined by  $\Pi_R$ . The rowwise and columnwise partitions  $\Pi_R$  and  $\Pi_C$  together define a 2D partition on the matrix  $\mathbf{A}$ , where  $\text{map}(a_{ij}) = P_{p,q} \Leftrightarrow r_i \in \mathcal{R}_p$  and  $c_j \in \mathcal{C}_q$ .

In order to achieve a load balance among processors, a multi-constraint partitioning formulation is used (line 8 of the algorithm). Each vertex  $v_i$  of  $\mathcal{H}_C$  is assigned  $P$  weights:  $w[i, p]$ , for  $p = 1, \dots, P$ . Here,  $w[i, p]$  is equal to the number of nonzeros of column  $c_i$  in rows  $\mathcal{R}_p$  (line 7 of Fig. 4). Consider a  $Q$ -way partitioning of  $\mathcal{H}_C$  with  $P$  constraints using the vertex weight definition



**Hypergraph Partitioning.** Fig. 3 Second step of four-way jagged-like partitioning: (a) Row-net representations of submatrices of  $\mathbf{A}$  and two-way partitionings, (b) Final permuted matrix; the nonzeros in the same partition are shown with the same shape and color; the deviation of the minimum and maximum numbers of nonzeros of a part from the average are displayed as an interval  $\text{imbal}$ ;  $\text{nnz}$  and  $\text{vol}$  denote, respectively, the number of nonzeros and the total communication volume

---

CHECKERBOARD-PARTITIONING( $\mathbf{A}$ ,  $K = P \times Q$ ,  $\varepsilon_1, \varepsilon_2$ )

Input: a matrix  $\mathbf{A}$ , the number of processors  $K = P \times Q$ , and the imbalance ratios  $\varepsilon_1, \varepsilon_2$ .

Output:  $\text{map}(a_{ij})$  for all  $a_{ij} \neq 0$  and  $\text{totalVolume}$ .

```

1: $\mathcal{H}_{\mathcal{R}} = (\mathcal{V}_{\mathcal{R}}, \mathcal{N}_{\mathcal{C}}) \leftarrow \text{columnNet}(\mathbf{A})$
2: $\Pi_{\mathcal{R}} = \{\mathcal{R}_1, \dots, \mathcal{R}_P\} \leftarrow \text{partition}(\mathcal{H}_{\mathcal{R}}, P, \varepsilon_1)$ \triangleright rowwise partitioning of \mathbf{A}
3: $\text{expandVolume} \leftarrow \text{cutsize}(\Pi_{\mathcal{R}})$
4: $\mathcal{H}_{\mathcal{C}} = (\mathcal{V}_{\mathcal{C}}, \mathcal{N}_{\mathcal{C}}) \leftarrow \text{rowNet}(\mathbf{A})$
5: for $j = 1$ to $|\mathcal{V}_{\mathcal{C}}|$ do
6: for $p = 1$ to P do
7: $w_{j,p} = |\{n_j \cap \mathcal{R}_p\}|$
8: $\Pi_{\mathcal{C}} = \{\mathcal{C}_1, \dots, \mathcal{C}_Q\} \leftarrow \text{MCPartition}(\mathcal{H}_{\mathcal{C}}, Q, \varepsilon_2)$ \triangleright columnwise partitioning of \mathbf{A}
9: $\text{foldVolume} \leftarrow \text{cutsize}(\Pi_{\mathcal{C}})$
10: for all $a_{ij} \neq 0$ of \mathbf{A} do
11: $\text{map}(a_{ij}) = P_{p,q} \Leftrightarrow r_i \in \mathcal{R}_p$ and $c_j \in \mathcal{C}_q$
12: $\text{totalVolume} \leftarrow \text{expandVolume} + \text{foldVolume}$
```

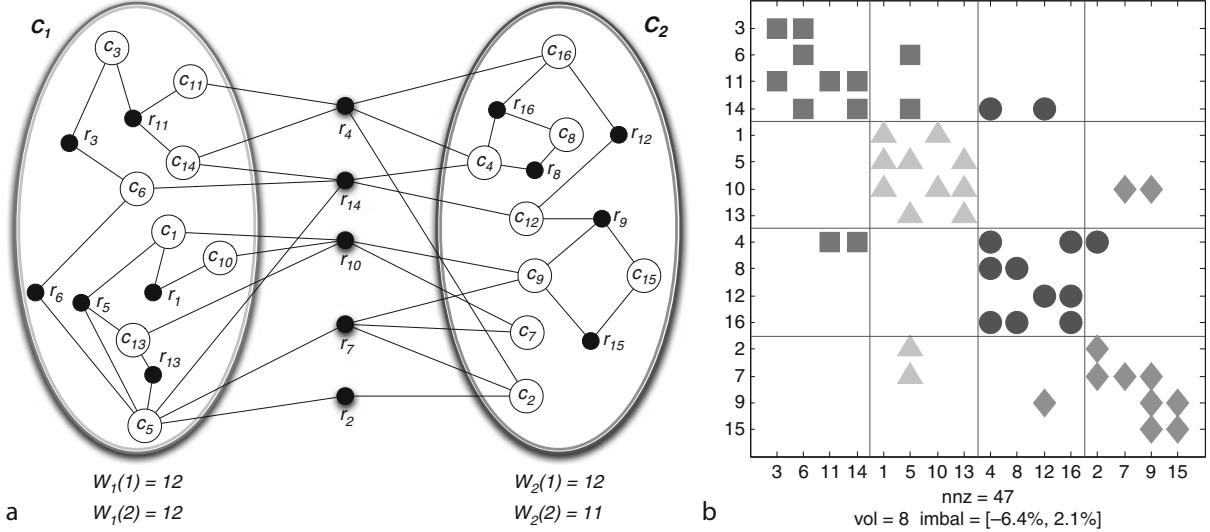
---

**Hypergraph Partitioning.** Fig. 4 Checkerboard partitioning

above. Maintaining the  $P$  balance constraints (4) corresponds to maintaining computational load balance on the processors of each row of the processor mesh.

Establishing the equivalence between the total communication volume and the sum of the cutsizes of the

two partitions is fairly straightforward. The volume of communication for the fold operations corresponds exactly to the  $\text{cutsize}(\Pi_{\mathcal{C}})$ . The volume of communication for the expand operations corresponds exactly to the  $\text{cutsize}(\Pi_{\mathcal{R}})$ .



**Hypergraph Partitioning, Fig. 5** Second step of four-way checkerboard partitioning: **(a)** two-way multi-constraint partitioning  $\Pi_C$  of row-net hypergraph representation  $\mathcal{H}_C$  of  $\mathbf{A}$ , **(b)** Final checkerboard partitioning of  $\mathbf{A}$  induced by  $(\Pi_R, \Pi_C)$ ; the nonzeros in the same partition are shown with the same shape and color; the deviation of the minimum and maximum numbers of nonzeros of a part from the average are displayed as an interval `imbal`; `nnz` and `vol` denote, respectively, the number of nonzeros and the total communication volume

**Figure 5b** displays the  $2 \times 2$  checkerboard partition induced by  $(\Pi_{\mathcal{R}}, \Pi_{\mathcal{C}})$ . Here,  $\Pi_{\mathcal{R}}$  is a rowwise two-way partition giving the same figure as shown in [Fig. 2](#), and  $\Pi_{\mathcal{C}}$  is a two-way multi-constraint partition  $\Pi_{\mathcal{C}}$  of the row-net hypergraph model  $\mathcal{H}_{\mathcal{C}}$  of  $\mathbf{A}$  shown in [Fig. 5a](#). In [Fig. 5a](#),  $w[9, 1] = 0$  and  $w[9, 2] = 4$  for internal column  $c_9$  of row stripe  $\mathcal{R}_2$ , whereas  $w[5, 1] = 2$  and  $w[5, 2] = 4$  for external column  $c_5$ .

Another common method of matrix partitioning is the *orthogonal recursive bisection* (ORB) [27]. In this approach, the matrix is first partitioned rowwise into two submatrices using the column-net hypergraph model, and then each part is further partitioned columnwise into two parts using the row-net hypergraph model. The process is continued recursively until the desired number of parts is obtained. The algorithm is shown in Fig. 6. In this algorithm,  $\text{dim}$  represents either rowwise or columnwise partitioning, where  $-\text{dim}$  switches the partitioning dimension.

In the ORB method shown above, the step *bisect* ( $A, dim, \epsilon$ ) corresponds to partitioning the given matrix either along the rows or columns with, respectively, the column-net or the row-net hypergraph models into two. The total sum of the cutsizes (3) of each bisection

step corresponds to the total communication volume. It is possible to dynamically adjust the  $\epsilon$  at each recursive call by allowing larger imbalance ratio for the recursive call on the submatrix  $A_1$  or  $A_2$ .

## Some Other Applications of Hypergraph Partitioning

As said before, the initial motivations for hypergraph models were accurate modeling of the nonzero structure of unsymmetric and rectangular sparse matrices to minimize the communication volume for iterative solvers. There are other applications that can make use of hypergraph partitioning formulation. Here, a brief overview of general classes of applications is given along with the names of some specific problems. Further application classes are given in bibliographic notes.

*Parallel reduction* or aggregation operations form a significant class of such applications, including the MapReduce model. The reduction operation consists of computing  $M$  output elements from  $N$  input elements. An output element may depend on multiple input elements, and an input element may contribute to multiple output elements. Assume that the operation on which

---

ORB-PARTITIONING( $\mathbf{A}$ ,  $dim$ ,  $K_{min}$ ,  $K_{max}$ ,  $\epsilon$ )

Input: a matrix  $\mathbf{A}$ , the part numbers  $K_{min}$  (at initial call, it is equal to 1) and  $K_{max}$  (at initial call it is equal to  $K$ , the desired number of parts), and the imbalance ratio  $\epsilon$ .

Output:  $map(a_{ij})$  for all  $a_{ij} \neq 0$ .

```

1: if $K_{max} - K_{min} > 0$ then
2: $mid \leftarrow (K_{max} - K_{min} + 1)/2$
3: $\Pi = \langle \mathbf{A}_1, \mathbf{A}_2 \rangle \leftarrow \text{bisect}(\mathbf{A}, dim, \epsilon)$ ▷ Partition \mathbf{A} along dim into two, producing two submatrices
4: $\text{totalVolume} \leftarrow \text{totalVolume} + \text{cutsizes}(\Pi)$
 ▷ Recursively partition each submatrix along the orthogonal direction
5: $map1(\mathbf{A}_1) \leftarrow \text{ORB-PARTITIONING}(\mathbf{A}_1, -dim, K_{min}, K_{min} + mid - 1, \epsilon)$
6: $map2(\mathbf{A}_2) \leftarrow \text{ORB-PARTITIONING}(\mathbf{A}_2, -dim, K_{min} + mid, K_{max}, \epsilon)$
7: $map(\mathbf{A}) \leftarrow map1(\mathbf{A}_1) \cup map2(\mathbf{A}_2)$
8: else
9: $map(\mathbf{A}) \leftarrow K_{min}$
```

---

**Hypergraph Partitioning. Fig. 6** Orthogonal recursive bisection (ORB)

reduction is performed is commutative and associative. Then, the inherent computational structure can be represented with an  $M \times N$  dependency matrix, where each row and column of the matrix represents an output element and an input element, respectively. For an input element  $x_j$  and an output element  $y_i$ , if  $y_i$  depends on  $x_j$ ,  $a_{ij}$  is set to 1 (otherwise zero). Using this representation, the problem of partitioning the workload for the reduction operation is equivalent to the problem of partitioning the dependency matrix for efficient SpMxV.

In some other reduction problems, the input and output elements may be preassigned to parts. The proposed hypergraph model can be accommodated to those problems by adding  $K$  part vertices and connecting those vertices to the nets which correspond to the preassigned input and output elements. Obviously, those part vertices must be fixed to the corresponding parts during the partitioning. Since the required property is already included in the existing hypergraph partitioners [6, 12, 19], this does not add extra complexity to the partitioning methods.

*Iterative methods for solving linear systems* usually employ *preconditioning* techniques. Roughly speaking, preconditioning techniques modify the given linear system to accelerate convergence. Applications of explicit preconditioners in the form of approximate inverses or factored approximate inverses are amenable to parallelization. Because, these techniques require SpMxV operations with the approximate inverse or factors of the approximate inverse at each step. In

other words, preconditioned iterative methods perform SpMxV operations with both coefficient and preconditioner matrices in a step. Therefore, parallelizing a full step of these methods requires the coefficient and preconditioner matrices to be well partitioned, for example, processors' loads are balanced and communication costs are low in both multiply operations. To meet this requirement, the coefficient and preconditioner matrices should be partitioned simultaneously. One can accomplish such a simultaneous partitioning by building a single hypergraph and then partitioning that hypergraph. Roughly speaking, one follows a four-step approach: (i) build a hypergraph for each matrix, (ii) determine which vertices of the two hypergraphs need to be in the same part (according to the computations forming the iterative method), (iii) amalgamate those vertices coming from different hypergraphs, (iv) if the computations represented by the two hypergraphs of the first step are separated by synchronization points then assign multiple weights to vertices (the weights of the vertices of the hypergraphs of the first step are kept), otherwise assign a single weight to vertices (the weights of the vertices of the hypergraphs of the first step are summed up for each amalgamation).

The computational structure of the preconditioned iterative methods is similar to that of a more general class of scientific computations including multiphase, multiphysics, and multi-mesh simulations.

In *multiphase simulations*, there are a number of computational phases separated by global synchronization points. The existence of the global synchronizations

necessitates each phase to be load balanced individually. Multi-constraint formulation of hypergraph partitioning can be used to achieve this goal.

In *multi-physics simulations*, a variety of materials and processes are analyzed using different physics procedures. In these types of simulations, computational as well as the memory requirements are not uniform across the mesh. For scalability issues, processor loads should be balanced in terms of these two components. The multi-constraint partitioning framework also addresses these problems.

In *multi-mesh simulations*, a number of grids with different discretization schemes and with arbitrary overlaps are used. The existence of overlapping grid points necessitates a simultaneous partitioning of the grids. Such a simultaneous partitioning scheme should balance the computational loads of the processors and minimize the communication cost due to interactions within a grid as well as the interactions among different grids. With a particular transformation (the vertex amalgamation operation, also mentioned above), hypergraphs can be used to model the interactions between different grids. With the use of multi-constraint formulation, the partitioning problem in the multi-mesh computations can also be formulated as a hypergraph partitioning problem.

In obtaining partitions for two or more computation phases interleaved with synchronization points, the hypergraph models lead to the minimization of the overall sum of the total volume of communication in all phases (assuming that a single hypergraph is built as suggested in the previous paragraphs). In some sophisticated simulations, the magnitude of the interactions in one phase may be different than that of the interactions in another one. In such settings, minimizing the total volume of communication in each phase separately may be advantageous. This problem can be formulated as a multi-objective hypergraph partitioning problem on the so-built hypergraphs.

There are certain limitations in applying hypergraph partitioning to the multiphase, multiphysics, and multi-mesh-like computations. The dependencies must remain the same throughout the computations, otherwise the cutsize may not represent the communication volume requirements as precisely as before. The weights assigned to the vertices, for load balancing issues, should be static and available prior to the

partitioning; the hypergraph models cannot be used as naturally for applications whose computational requirements vary drastically in time. If, however, the computational requirements change gradually in time, then the models can be used to re-partition the load at certain time intervals (while also minimizing the redistribution or migration costs associated with the new partition).

*Ordering methods* are quite common techniques to permute matrices in special forms in order to reduce the memory and running time requirements, as well as to achieve increased parallelism in direct methods (such as LU and Cholesky decompositions) used for solving systems of linear equations. Nested-dissection is a well-known ordering method that has been used quite efficiently and successfully. In the current state-of-the-art variations of the nested-dissection approach, a matrix is symmetrically permuted with a permutation matrix  $\mathbf{P}$  into doubly bordered block diagonal form

$$\mathbf{A}_{DB} = \mathbf{P} \mathbf{A} \mathbf{P}^T \begin{bmatrix} A_{11} & & & A_{1S} \\ & A_{22} & & A_{2S} \\ & & \ddots & \vdots \\ & & & A_{KK} & A_{KS} \\ A_{S1} & A_{S2} & \cdots & A_{SK} & A_{SS} \end{bmatrix},$$

where the nonzeros are only in the marked blocks (the blocks on the diagonal and the row and column borders). The aim in such a permutation is to have reduced numbers of rows/columns in the borders and to have equal-sized square blocks in the diagonal. One way to achieve such a permutation when  $\mathbf{A}$  has symmetric pattern is as follows. Suppose a matrix  $\mathbf{B}$  is given (if not, it is possible to find one) where the sparsity pattern of  $\mathbf{B}^T \mathbf{B}$  equals to that of  $\mathbf{A}$  (here arithmetic cancellations are ignored). Then, one can permute  $\mathbf{B}$  nonsymmetrically into the singly bordered form

$$\mathbf{B}_{SB} = \mathbf{Q} \mathbf{B} \mathbf{P}^T \begin{bmatrix} B_{11} & & & B_{1S} \\ & B_{22} & & B_{2S} \\ & & \ddots & \vdots \\ & & & B_{KK} & B_{KS} \end{bmatrix},$$

so that  $\mathbf{B}_{SB}^T \mathbf{B}_{SB} = \mathbf{P} \mathbf{A} \mathbf{P}^T$ ; that is, one can use the column permutation of  $\mathbf{B}$  resulting in  $\mathbf{B}_{SB}$  to obtain a symmetric permutation for  $\mathbf{A}$  which results in  $\mathbf{A}_{DB}$ . Clearly, the column dimension of  $B_{kk}$  will be the size of the square matrix  $A_{kk}$  and the number of rows and columns in the border will be equal to the number of columns in the column border of  $\mathbf{B}_{SB}$ . One can achieve such a permutation of  $\mathbf{B}$  by partitioning the column-net model of  $\mathbf{B}$  while reducing the cutsize according to the cut-net metric (2), with unit net costs, to obtain the permutation  $\mathbf{P}$  as follows. First, the permutation  $\mathbf{Q}$  is defined to be able to define  $\mathbf{P}$ . Permute all rows corresponding to the vertices in part  $k$  before those in a part  $\ell$ , for  $1 \leq k < \ell \leq K$ . Then, permute all columns corresponding to the nets that are internal to a part  $k$  before those that are internal to a part  $\ell$ , for  $1 \leq k < \ell \leq K$ , yielding the diagonal blocks, and then permute all columns corresponding to the cut nets to the end, yielding the column border (the order of column defining a diagonal block). Clearly the correspondence between the size of the column border of  $\mathbf{B}_{SB}$  and the doubly border of  $\mathbf{A}_{DB}$  is exact, and hence the cutsize according to the cut-net metric is an exact measure. The requirement to have almost equal sized square blocks  $A_{kk}$  decoded as the requirement that each part should have an almost equal number of internal nets in the partition of the column-net model of  $\mathbf{B}$ . Although such a requirement is neither the objective nor the constraint of the hypergraph partitioning problem, the common hypergraph-partitioning heuristics easily accommodate such requirements.

## Related Entries

- ▶ [Data Distribution](#)
- ▶ [Graph Algorithms](#)
- ▶ [Graph Partitioning](#)
- ▶ [Linear Algebra, Numerical](#)
- ▶ [PaToH \(Partitioning Tool for Hypergraphs\)](#)
- ▶ [Preconditioners for Sparse Iterative Methods](#)
- ▶ [Sparse Direct Methods](#)

## Bibliographic Notes and Further Reading

The first use of the hypergraph partitioning methods for efficient parallel sparse matrix-vector multiply

operations is seen in [10]. A more comprehensive study [11] describes the use of the row-net and column-net hypergraph models in 1D sparse matrix partitioning. For different views and alternatives on vector partitioning see [5, 22, 24].

A fair treatment of parallel sparse matrix-vector multiplication, analysis, and investigations on certain matrix types along with the use of hypergraph partitioning is given in [4, Chapter 4]. Further analysis of hypergraph partitioning on some model problems is given in [25].

Hypergraph partitioning schemes for preconditioned iterative methods are given in [23], where vertex amalgamation and multi-constraint weighting to represent different phases of computations are given. Discussions on application of such methodology for multiphase, multiphysics, and multi-mesh simulations are also discussed in the same paper.

Some different methods for sparse matrix partitioning using hypergraphs can be found in [26], including jagged-like and checkerboard partitioning methods, and in [27], the orthogonal recursive bisection approach. A recipe to choose a partitioning method for a given matrix is given in [16].

The use of hypergraph models for permuting matrices into special forms such as singly bordered block-diagonal form can be found in [3]. This permutation can be leveraged to develop hypergraph partitioning-based symmetric [9, 15] and nonsymmetric [17] nested-dissection orderings.

The standard hypergraph partitioning and the hypergraph partitioning with fixed vertices formulation, respectively, is used for static and dynamic load balancing for some scientific applications in [7, 8].

Some other applications of hypergraph partitioning are briefly summarized in [2]. These include image-space parallel direct volume rendering, parallel mixed integer linear programming, data declustering for multi-disk databases, scheduling file-sharing tasks in heterogeneous master-slave computing environments, and work-stealing scheduling, road network clustering methods for efficient query processing, pattern-based data clustering, reducing software development and maintenance costs, processing spatial join operations, and improving locality in memory or cache performance.

## Bibliography

1. Ababei C, Selvakkumaran N, Bazargan K, Karypis G (2002) Multi-objective circuit partitioning for cutszie and path-based delay minimization. In: Proceedings of ICCAD 2002, San Jose, CA, November 2002
2. Aykanat C, Cambazoglu BB, Uçar B (2008) Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices. *J Parallel Distr Comput* 68(5): 609–625
3. Aykanat C, Pinar A, Çatalyürek UV (2004) Permuting sparse rectangular matrices into block-diagonal form. *SIAM J Sci Comput* 26(6):1860–1879
4. Bisseling RH (2004) Parallel scientific computation: a structured approach using BSP and MPI. Oxford University Press, Oxford, UK
5. Bisseling RH, Meesen W (2005) Communication balancing in parallel sparse matrix-vector multiplication. *Electron Trans Numer Anal* 21:47–65
6. Boman E, Devine K, Heaphy R, Hendrickson B, Leung V, Riesen LA, Vaughan C, Catalyürek U, Bozdag D, Mitchell W, Teresco J (2007) Zoltan 3.0: parallel partitioning, load balancing, and data-management services; user's guide. Sandia National Laboratories, Albuquerque, NM, 2007. Technical Report SAND2007-4748W [http://www.cs.sandia.gov/Zoltan/ug\\_html/ug.html](http://www.cs.sandia.gov/Zoltan/ug_html/ug.html)
7. Cambazoglu BB, Aykanat C (2007) Hypergraph-partitioning-based remapping models for image-space-parallel direct volume rendering of unstructured grids. *IEEE Trans Parallel Distr Syst* 18(1):3–16
8. Catalyürek U, Boman E, Devine K, Bozdag D, Heaphy R, Riesen L (2009) A repartitioning hypergraph model for dynamic load balancing. *J Parallel Distr Comput* 69(8):711–724
9. Çatalyürek UV (1999) Hypergraph models for sparse matrix partitioning and reordering. Ph.D. thesis, Computer Engineering and Information Science, Bilkent University. Available at <http://www.cs.bilkent.edu.tr/tech-reports/1999/ABSTRACTS.1999.html>
10. Çatalyürek UV, Aykanat C (1995) A hypergraph model for mapping repeated sparse matrixvector product computations onto multicomputers. In: Proceedings of International Conference on High Performance Computing (HiPC'95), Goa, India, December 1995
11. Çatalyürek UV, Aykanat C (1999) Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans Parallel Distr Syst* 10(7):673–693
12. Çatalyürek UV, Aykanat C (1999) PaToH: a multilevel hypergraph partitioning tool, version 3.0. Department of Computer Engineering, Bilkent University, Ankara, 06533 Turkey. PaToH is available at <http://bmi.osu.edu/~umit/software.htm>
13. Çatalyürek UV, Aykanat C (2001) A fine-grain hypergraph model for 2D decomposition of sparse matrices. In: Proceedings of 15th International Parallel and Distributed Processing Symposium (IPDPS), San Francisco, CA, April 2001
14. Çatalyürek UV, Aykanat C (2001) A hypergraph-partitioning approach for coarse-grain decomposition. In: ACM/IEEE SC2001, Denver, CO, November 2001
15. Çatalyürek UV, Aykanat C, Kayaaslan E (2009) Hypergraph partitioning-based fill-reducing ordering. Technical Report OSUBMI-TR-2009-n02 and BU-CE-0904, Department of Biomedical Informatics, The Ohio State University and Computer Engineering Department, Bilkent University (Submitted)
16. Çatalyürek UV, Aykanat C, Uçar B (2010) On two-dimensional sparse matrix partitioning: models, methods, and a recipe. *SIAM J Sci Comput* 32(2):656–683
17. Grigori L, Boman E, Donfack S, Davis T (2008) Hypergraph unsymmetric nested dissection ordering for sparse LU factorization. Technical Report 2008-1290J, Sandia National Labs, Submitted to SIAM J Sci Comp
18. Karypis G, Kumar V (1998) Multilevel algorithms for multi-constraint hypergraph partitioning. Technical Report 99-034, Department of Computer Science, University of Minnesota/Army HPC Research Center, Minneapolis, MN 55455
19. Karypis G, Kumar V, Aggarwal R, Shekhar S (1998) hMeTiS a hypergraph partitioning package, version 1.0.1. Department of Computer Science, University of Minnesota/Army HPC Research Center, Minneapolis
20. Lengauer T (1990) Combinatorial algorithms for integrated circuit layout. Wiley–Teubner, Chichester
21. Selvakkumaran N, Karypis G (2003) Multi-objective hypergraph partitioning algorithms for cut and maximum subdomain degree minimization. In: Proceedings of ICCAD 2003, San Jose, CA, November 2003
22. Uçar B, Aykanat C (2004) Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies. *SIAM J Sci Comput* 25(6):1837–1859
23. Uçar B, Aykanat C (2007) Partitioning sparse matrices for parallel preconditioned iterative methods. *SIAM J Sci Comput* 29(4):1683–1709
24. Uçar B, Aykanat C (2007) Revisiting hypergraph models for sparse matrix partitioning. *SIAM Review* 49(4):595–603
25. Uçar B, Çatalyürek UV (2010) On the scalability of hypergraph models for sparse matrix partitioning. In: Danelutto M, Bourgeois J, Gross T (eds), Proceedings of the 18th Euromicro Conference on Parallel, Distributed, and Network-based Processing, IEEE Computer Society, Conference Publishing Services, pp 593–600
26. Uçar B, Çatalyürek UV, Aykanat C (2010) A matrix partitioning interface to PaToH in MATLAB. *Parallel Computing* 36(5–6):254–272
27. Vastenhoudt B, Bisseling RH (2005) A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review* 47(1):67–95

## Hyperplane Partitioning

### ► Tiling

## HyperTransport

FEDERICO SILLA

Universidad Politécnica de Valencia, Valencia, Spain

### Synonyms

HT; HT3.10

### Definition

HyperTransport is a scalable packet-based, high-bandwidth, and low-latency point-to-point interconnect technology intended to interconnect processors and also link them to I/O peripheral devices. HyperTransport was initially devised as an efficient replacement for traditional system buses for on-board communications. Nevertheless, the last extension to the standard, referred to as High Node Count HyperTransport, as well as the recent standardization of new cables and connectors, allow HyperTransport to efficiently extend its interconnection capabilities beyond a single motherboard and become a very efficient technology to interconnect processors and I/O devices in a cluster. HyperTransport is an open standard managed by the HyperTransport Consortium.

### Discussion

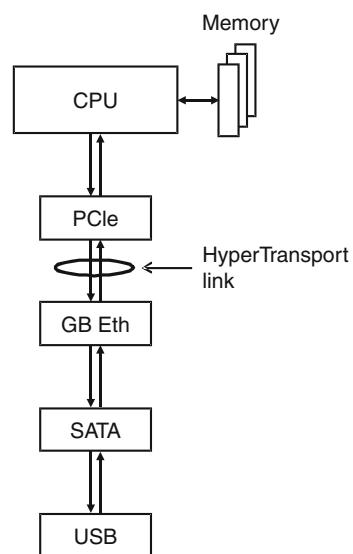
A complete description of the HyperTransport technology should include both an introduction to the protocol used by communicating devices to exchange data and also a description of the electrical interface that this technology makes use of in order to achieve its tremendous performance. However, describing the electrical interface seems to be less interesting than the protocol itself and, additionally, requires the reader to have a large electrical background. Therefore, the following description will be focused on the protocol used by the HyperTransport technology, leaving aside the electrical details. On the other hand, the protocol used by HyperTransport is a quite complex piece of technology, thus requiring an extensive explanation, which may be out of the scope of this encyclopedia. For this reason, the following description just tries to be a brief introduction to HyperTransport. Finally, the reader should note that AMD uses an extended version of the HyperTransport protocol in order to provide cache coherency among the processors in a system. Such extended protocol, usually

referred to as coherent HyperTransport (cHT), is proprietary to AMD, and therefore it is not described in this document. Nevertheless, the main difference between both protocols is that the coherent one includes some additional types of packets.

### HyperTransport Links

The HyperTransport technology is a point-to-point communication standard, meaning that each of the HyperTransport links in the system connects exactly two devices. [Figure 1](#) shows a simplified diagram of a system that deploys HyperTransport in order to interconnect the devices it is composed of. As can be seen in that figure, the main processor is connected to a PCIe device by means of a HyperTransport link. That PCIe device is, at the same time, connected to a GigaByte Ethernet device, which is, additionally, connected to a SATA device. This device connects to a USB device. All of these devices make up a HyperTransport daisy chain. Nevertheless, devices can implement multiple HyperTransport links in order to build larger HyperTransport fabrics.

Each of the links depicted in [Fig. 1](#) consists of two unidirectional and independent sets of signals. Each of these sets includes its own CAD signal, as well as CTL and CLK signals.



**HyperTransport. Fig. 1** System deploying HyperTransport

The CAD signal (named after Command, Address, and Data) carries control and data packets. The CTL signal (named after ConTroL) is intended to differentiate control from data packets in the CAD signal. Finally, the CLK signal (named after CLock) carries the clock for the CAD and CTL signals. [Figure 2](#) shows a diagram presenting all these signals.

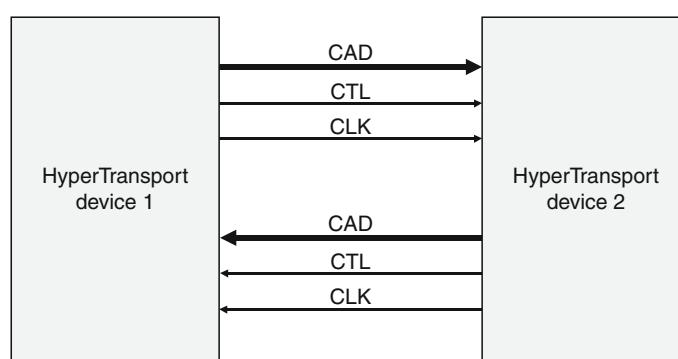
The width of the CAD signal is variable from 2 to 32 bits. Actually, not all values are possible. Only widths of 2-, 4-, 8-, 16-, or 32-bits are allowed. Nevertheless, note that the HyperTransport protocol remains the same independently of the exact link width. More precisely, the format of the packets exchanged among HyperTransport devices does not depend on link width. However, a given packet will require more time to be transmitted in a 2-bit link than in a 32-bit one. The reason for having several widths for the CAD signal is allowing system developers to tune their system for a given performance/cost design point. Narrower links will be cheaper to implement, but they will also provide lower performance.

On the other hand, the width of the CAD signal in each of the unidirectional portions of the link may be different, that is, HyperTransport allows asymmetrical link widths. Therefore, a given device may implement a 32-bit wide link in one direction while deploying a 4-bit link in the other, for example. The usefulness of such asymmetry is based on the fact that, if such a device sends most of its data in one direction and receives a limited amount of data in the other direction, then the system designer can reduce manufacturing cost by providing a wide link in the direction that requires higher

bandwidth and a narrow link for the opposite direction, which requires much less bandwidth.

In addition to the variable link width, HyperTransport also supports variable clock speeds, thus increasing even more the possibilities the system designer has for tuning the bandwidth of the links. The clock speeds currently supported by the HyperTransport specification are 200 MHz, 300 MHz, 400 MHz, 500 MHz, 600 MHz, 800 MHz, 1 GHz, 1.2 GHz, 1.6 GHz, 2 GHz, 2.4 GHz, 2.6 GHz, 2.8 GHz, 3 GHz, and 3.2 GHz. Moreover, clock frequency in both directions of a link does not need to be the same, thus introducing an additional degree of asymmetry. On the other hand, the clock mechanism used in HyperTransport is referred to as double data rate (DDR), which means that rising and falling edges of the clock signal are used to latch data, thus achieving an effective clock frequency that doubles the actual clock rate.

In summary, when variable link width is combined with variable clock frequency, HyperTransport links present an extraordinary scalability in terms of performance. For example, when both directions of a link are 2-bit wide working at 200 MHz, link bandwidth is 200 MB/s. This would be the lowest performance/lowest cost configuration. On the opposite, when 32-bit wide links are used at 3.2 GHz, overall link performance rises up to 51.2 GB/s. This implementation represents the highest bandwidth configuration, also presenting the highest manufacturing cost. This link configuration could be interesting for extreme performance systems, for example, which usually require as much bandwidth as possible. On the other hand, slow devices not



HyperTransport. [Fig. 2](#) Signals in a HyperTransport link

requiring high bandwidth could reduce cost by using narrow links. If these links are additionally clocked at low frequencies, then power consumption is further reduced.

In the same way that the CAD signal can be implemented with a variable width, the CTL and CLK signals can also be implemented with several widths. However, their width does not depend on the implementer's criterion to choose a performance/cost design point, but on the width of the CAD signal. Additionally, as the CAD signal can present a different width in each direction of the link, then the width of the CTL and CLK signals can also be different in each direction, depending on the corresponding CAD signal in that direction.

In the case of the CTL signal, there is an individual CTL bit for each set of 8, or fewer, CAD bits. Therefore, 1, 2, or 4 CTL bits can be found in a HyperTransport link. Moreover, the CTL bits are encoded in such a way that four CTL bits are transferred every 32 CAD bits. These four bits are intended to denote different flavors of the information being transmitted in the CAD signal. These different flavors may include, for example, that a command is being transmitted, that a CRC for a command without data is in the CAD signal, that the CAD signal is being used by a data packet, etc.

In the case for the CLK signal, a HyperTransport link has an individual CLK bit for every set of 8, or fewer, CAD bits. Thus, the number of CTL and CLK bits in a given link is the same. The reason for having a CLK bit for every 8 bits of the CAD signal is because link implementation is made easier. Effectively, the HyperTransport clocking scheme requires that the skew between clock and data signals must be minimized in order to achieve high transmission rates. Therefore, having a CLK bit for every 8 CAD bits allows that differences in trace lengths in the board layout are much lower than just having a CLK bit for the entire set of CAD bits.

In addition to the signals mentioned above, all HyperTransport devices share one PWROK and one RESET# signals for initialization and reset purposes. Moreover, if devices require power management, they should additionally include LDTSTOP# and LDTREQ# signals.

## HyperTransport Packets

Once described the links that connect devices in a HyperTransport fabric, this section presents the packets

that are forwarded along those links. Packets in HyperTransport are multiples of 4-bytes long and carry the command, address, and data associated with each transaction among devices. Packets can be classified into control and data packets.

Control packets are used to initiate and finalize transactions as well as to manage several HyperTransport features, and consist of 4 or 8 bytes. Control packets can be classified into information, request, and response packets.

Information packets are used for several link management purposes, like link synchronization, error condition signaling, and updating flow control information. Information packets are always 4 bytes long and can only be exchanged among adjacent devices directly interconnected by a link.

On the other hand, request and response control packets are used to build HyperTransport transactions. Request packets, which are 4- or 8-bytes long, are used to initiate HyperTransport transactions. On the other hand, response packets, which are always 4-bytes long, are used in the response phase of transactions to reply to a previous request. [Table 1](#) shows the different request and response types of packets. As can be seen, there are two different types of sized writes: posted and non-posted. Although both types write data to the target device of the request packet, their semantics are different. Non-posted writes require a response packet to be sent back to the requesting device in order to confirm that the operation has completed. On the other hand, posted writes do not require such confirmation.

**HyperTransport. Table 1** Types of request and response control packets

|                 | Packet type                 |
|-----------------|-----------------------------|
| Request packet  | Sized read                  |
|                 | Sized write (non-posted)    |
|                 | Sized write (posted)        |
|                 | Atomic read-modify-write    |
|                 | Broadcast                   |
|                 | Flush                       |
|                 | Fence                       |
|                 | Address extension           |
|                 | Source identifier extension |
| Response packet | Read response               |
|                 | Target done                 |

All the packet types will be further described in next section, except the extension packets. These packets are 4-bytes long extensions that prepend some of the other packets, when required. Their purpose is to allow 64-bit addressing instead of the 40-bit one used by default, in the case of the Address Extension, or 16-bit source identifiers in bus-device-function format in the case of the Source Identifier Extension.

With respect to data packets, they carry the actual data transferred among devices. Data packets only include data, with no associated control field. Therefore, data packets immediately follow the associated control packet. For example, a read response control packet, which includes no data, will be followed by the associated data packet carrying the read data. In the same way, a write request control packet will be followed by the data to be written.

Data packet length depends on the command that generated that data packet. Nevertheless, the maximum length is 64 bytes.

## HyperTransport Transactions

HyperTransport devices transfer data by means of transactions. For example, every time a device requests some data from another device, the former initiates a read transaction targeted to the latter. Write transactions happen in a similar way. For example, when a program writes some data to disk, a write transaction is initiated among the processor, which reads the data from main memory, and the SATA device depicted in Fig. 1. Other transactions include broadcasting a message or explicitly taking control of the order of pending transactions.

Every transaction has a request phase. Request control packets are used in this phase. The exact request control packet to be used depends on the particular transaction taking place. On the other hand, many transactions require a response stage. Response control packets are used in this case, for example, to return read data from the target of the transaction, or to confirm its completion.

There are six basic transaction types:

- Sized read transaction
- Sized write transaction
- Atomic read-modify-write transaction
- Broadcast transaction

- Flush transaction
- Fence transaction

### Sized Read Transaction

Sized read transactions are used by devices when they request data located in the address space of another device. For example, when a device wants to read data from main memory, it starts a read transaction targeted to the main processor. Also, when the processor requires some data from the USB device in Fig. 1, it will issue a read transaction destined to that device.

Read transactions begin with a sized read request control packet being issued by the device requesting the data. Once this packet reaches the destination device, it accesses the requested data and generates a response. This response will be composed of a read response control packet followed by the read data, included in a data packet. Once these two packets arrive at the device that initiated the process, the transaction is completed.

It is noteworthy mentioning that during the time elapsed since the requestor delivered the read request on the link until it receives the corresponding response, the HyperTransport chain is not idle. On the opposite, as HyperTransport is a split transaction protocol, other transactions can be issued (even finalized) before our requestor receives the required data.

### Sized Write Transaction

Sized write transactions are similar to sized read ones, with the difference that the requestor device writes data to the target device instead of requesting data from it. A write transaction may happen when a device sends data to memory, or when the processor writes back data from memory to disk, for example.

There are two different sized write transactions: posted and non-posted. Posted write transactions start when the requestor sends to the target a posted write request control packet followed by a data packet containing the data to be written. In this case, because of the posted nature of the transaction, no response packet is sent back to the requestor. On the other hand, non-posted write transactions begin with a non-posted write control packet being issued by the requestor (followed by a data packet). In this case, when both packets reach the destination and the data are written, the target issues back a target-done response control packet to the requestor. When the requestor receives this response, the transaction is finished.

## Atomic Read-Modify-Write Transaction

Atomic read-modify-write transactions are intended to atomically access a memory location and modify it. This means that no other device in the system may access the same memory location during the time required to read and modify it. This is useful to avoid race conditions among devices while performing the mutual exclusion required to access a critical section, for example.

Two different types of atomic operations are allowed:

- Fetch and Add
- Compare and Swap

The Fetch and Add operation is:

```
Fetch_and_Add(Out, Addr, In) {
 Out = Mem[Addr];
 Mem[Addr] = Mem[Addr] + In;
}
```

The Compare and Swap operation is:

```
Compare_and_Swap(Out, Addr, Compare,
 In) {
 Out = Mem[Addr];
 If (Mem[Addr] == Compare) Mem[Addr]
 = In;
}
```

The atomic transaction begins when the requestor issues an atomic read-modify-write request control packet on the link followed by a data packet containing the argument of the atomic operation. Once both packets are received at the target and it performs the requested atomic operation, it will send back a read response control packet followed by a data packet containing the original data read from the memory location.

## Broadcast Transaction

This transaction is used by the processor to communicate information to all HyperTransport devices in the system. This transaction is started with a broadcast request control packet, which can only be issued by the processor. All the other devices accept that packet and forward it to the rest of devices in the system. Broadcast requests include halt, shutdown, and End-Of-Interrupt commands.

## Flush Transaction

Posted writes do not generate any response once completed. Therefore, when a device issues one or more posted writes targeted to the main processor in order to write some data to main memory, the issuing device has no way to know that those writes have effectively completed their way to memory. Thus, if some of the posted writes have not been completely written to memory, that data will not be visible to other devices in the system. In this scenario, flush transactions allow the device that issued the posted writes to make sure that data has reached main memory by flushing all pending posted writes to memory.

Flush transactions begin when a device issues a flush control packet targeted to the processor. Once this control packet reaches the destination, all pending transactions will be flushed to memory and then the processor will generate a target-done response control packet back to the requestor. Once this packet is received, the transaction is completed.

## Fence Transaction

The fence transaction is intended to provide a barrier among posted writes. When a processor (the only possible target of a fence transaction) receives a fence command, it will make sure that no posted write received after the fence command is written to memory before any of the posted writes received earlier than the fence command.

The fence transaction begins when a device issues a fence control packet. No response is generated for this transaction.

## Virtual Channels and Flow Control in HyperTransport

All the different types of control and data packets are multiplexed on the same link and stored in input buffers when they reach the receiving end of the link. Then, they are either accepted by that device in case they are targeted to it or forwarded to the next link in the chain.

If packets are not carefully managed, a protocol deadlock may occur. For example, if many devices in the system issue a large number of non-posted requests, those requests may fill up all the available buffers and hinder responses to make forward progress back to the initial requestors. In this case, requestors would stop

forever because they are waiting for responses that will never arrive because the interconnect is full of requests that avoid responses to advance. Additionally, requests stored in the intermediate buffers will not be able to advance toward their destination because output buffers at the targets will be filled with responses that cannot enter the link, thus hindering targets to accept new requests from the link because they have no free space neither to store them nor to store the responses they would generate. As can be seen, in this situation, no packet can advance because of lack of available free buffers. The result is that the system freezes.

In order to avoid such deadlocks, HyperTransport splits traffic into virtual channels and stores different types of packets in buffers belonging to different virtual channels. Additionally, HyperTransport does not allow that packets traveling in one virtual channel move to another virtual channel. In this way, if non-posted requests use a virtual channel different from the one used by responses, the deadlock described above can be avoided.

HyperTransport defines a base set of virtual channels that must be supported by all HyperTransport devices. Moreover, some additional virtual channel sets are also defined, although support for them is optional. Regarding the base set, it includes three different virtual channels:

- The posted request virtual channel, which carries posted write transactions
- The non-posted request virtual channel, which includes reads, non-posted writes, and flush packets
- The response virtual channel, which is responsible for read responses and target-done control packets

In addition to separate traffic into the three virtual channels mentioned above, each device must implement separate control and data buffers for each of the virtual channels. Therefore, there are six types of buffers:

- Non-posted request control buffer
- Posted request control buffer
- Response control buffer
- Non-posted request data buffer
- Posted request data buffer
- Response data buffer

Figure 3 shows the basic buffer configuration for a HyperTransport link. The exact number of packets that

can be stored in each buffer depends on the implementation. Nevertheless, request and response buffers must contain, at least, enough space to store the largest control packet of that type. Also, all data buffers can hold 64 bytes. Moreover, in order to improve performance, a HyperTransport device may have larger buffers, able to store multiple packets of each type.

The HyperTransport protocol states that a transmitter should not issue a packet that cannot be stored by the receiver. Thus, the transmitter must know how many buffers of each type the receiver has available. To achieve this, a credit-based scheme is used between transmitters and receivers. With such scheme, the transmitter has a counter for each type of buffer implemented at the receiver. When the transmitter sends a packet, it decrements the associated counter. When one of the counters reaches zero, the transmitter stops sending packets of that type. On the other hand, when the receiver frees a buffer, it sends back a NOP control packet (No Operation Packet) to the transmitter in order to update it about space availability.

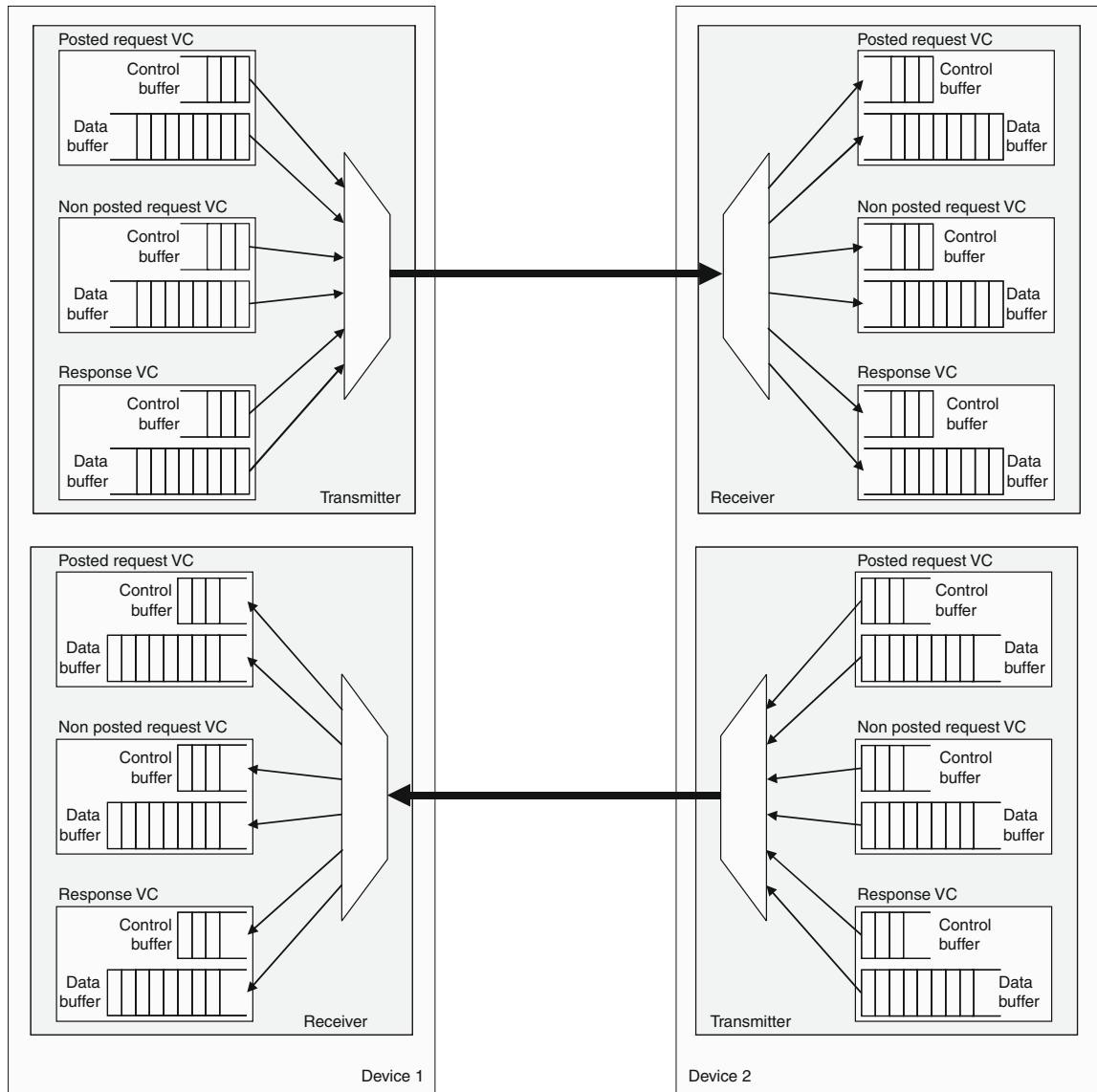
### Extending the HyperTransport Fabric

The system depicted in Fig. 1 consists of several HyperTransport devices interconnected in a daisy chain. However, more complex topologies can also be implemented by using HyperTransport bridges. Bridges in HyperTransport are devices having a primary link connecting toward the processor and one or more secondary links that allow extending the topology in the opposite direction. In this way, HyperTransport trees, like the one shown in Fig. 4, can be implemented.

In addition to the use of bridges, HyperTransport defines two more features able to expand a HyperTransport system. These features are the AC mode and the HTX connector. The AC mode allows devices to be connected over longer distances than allowed by regular links, which make use of the DC mode. On the other hand, the HTX connector allows external expansion cards to be plugged to the HyperTransport link, and be presented to the rest of the system as any other HyperTransport device.

### Improving the Scalability of HyperTransport

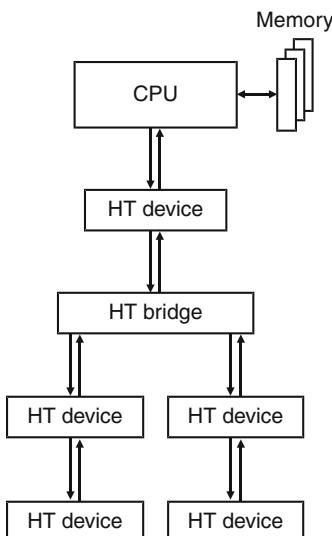
As shown above, HyperTransport offers some degree of scalability that enables the implementation of efficient



HyperTransport. Fig. 3 Buffer configuration for a HyperTransport link

topologies. Nevertheless, as HyperTransport was initially designed to replace traditional system buses, its benefits are mainly confined to interconnects within a single motherboard. Therefore, when HyperTransport topologies are to be scaled to larger sizes, in order to interconnect the processors and I/O subsystems in several motherboards (e.g., an entire cluster), HyperTransport is not able to do so because such large system sizes require routing capabilities that exceed current

HyperTransport ones. More specifically, HyperTransport is not able to provide device addressability beyond 32 devices. Additionally, it does not support efficient routing in scalable network topologies. As a result, high-performance computing vendors have no choice but to complement HyperTransport with other interconnect technologies, like Infiniband in the case for general purpose clusters, or proprietary interconnects, like in the case for Cray's XT4 and XT5 supercomputers [1].



**HyperTransport. Fig. 4** HyperTransport tree topology

In order to overcome the limitations of HyperTransport 3.10, an extension to it named High Node Count HyperTransport Specification was recently released. This extension supports very large system sizes, like the ones found in large data centers, at the same time that it is fully compatible with the HyperTransport specification. Additionally, the new extension adds a few bytes to current HyperTransport packets, but only in the cases where strictly required, thus minimizing the protocol overhead.

Briefly, the new extension provides an improved addressing scheme and a new control packet that allow HyperTransport devices to address any other device in large clusters. Additionally, new HyperTransport connectors and cables have been recently standardized in order to efficiently allow the deployment of the High Node Count HyperTransport Specification.

## Related Entries

- [Busses and Crossbars](#)
- [Data Centers](#)
- [Interconnection Networks](#)
- [PCI-Express](#)
- [PGAS \(Partitioned Global Address Space\) Languages](#)

H

## Bibliographic Notes and Further Reading

The complete description of HyperTransport 3.10 can be found in the HyperTransport I/O link specification 3.10 [5]. Additionally, readers are also encouraged to look up more information on HyperTransport in [3]. This book nicely describes the HyperTransport technology. On the other hand, a complete description of the High Node Count HyperTransport Specification can be found in [2]. Finally, many white papers and additional information are publicly available in the HyperTransport Consortium web site [4].

## Bibliography

1. Cray Inc. (2009) Cray XT5 specifications. Available online at <http://www.cray.com>
2. Duato J, Silla F, Holden B, Miranda P, Underhill J, Cavalli M, Yalamanchili S, Brüning U (2009) Extending HyperTransport protocol for improved Scalability. In: Proceedings of the first international workshop on hypertransport research and applications, Mannheim, Germany, pp 46–53
3. Holden B, Trodden J, Anderson D (2008) HyperTransport 3.1 interconnect technology: a comprehensive guide to the 1st, 2nd, and 3rd generations. MindShare Inc, Colorado Springs, CO
4. HyperTransport Technology Consortium web site. <http://www.hypertransport.org>. Accessed 2010
5. HyperTransport Technology Consortium. HyperTransport I/O link specification revision 3.10. Available online at <http://www.hypertransport.org>. Accessed 2010

# IBM Blue Gene Supercomputer

ALAN GARA, JOSÉ E. MOREIRA

IBM T.J. Watson Research Center, Yorktown Heights,  
NY, USA

## Synonyms

[Blue Gene/L](#); [Blue Gene/P](#); [Blue Gene/Q](#)

## Definition

The IBM Blue Gene Supercomputer is a massively parallel system based on the PowerPC processor. A Blue Gene/L system at Lawrence Livermore National Laboratory held the number 1 position in the TOP500 list of fastest computers in the world from November 2004 to November 2007.

## Discussion

### Introduction

The IBM Blue Gene Supercomputer is a massively parallel system based on the PowerPC processor. It derives its computing power from scalability and energy efficiency. Each computing node of Blue Gene is optimized to achieve high computational rate per unit of power and to operate with other nodes in parallel. This approach results in a system that can scale to very large sizes and deliver substantial aggregate performance.

Most large parallel systems in the 2000–2010 time frame followed a model of using off-the-shelf processors (typically from Intel, AMD, or IBM) and interconnecting them with either an industry standard network (e.g., Ethernet or Infiniband) or a proprietary network (e.g., as used by Cray or IBM). Blue Gene took a different approach by designing a dedicated system-on-a-chip (SoC) that included not only processors optimized for floating-point computing, but also the networking infrastructure to interconnect these building blocks into

a large system. This customized approach led to the scalability and power efficiency characteristics that differentiated Blue Gene from other machines that existed at the time of its commercial introduction in 2004.

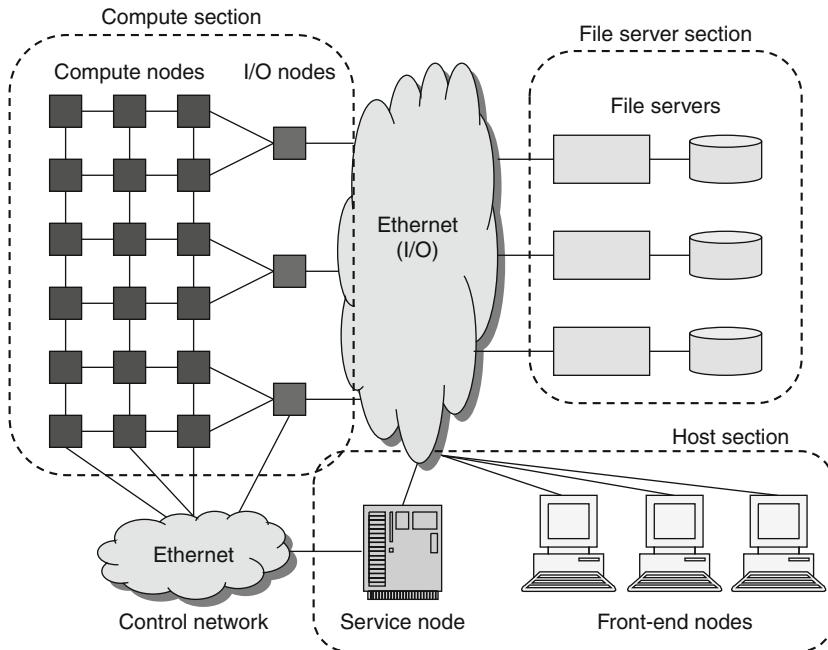
As of 2011, IBM has produced two commercial versions of Blue Gene, Blue Gene/L and Blue Gene/P, which were first delivered to customers in 2004 and 2007, respectively. A third version, Blue Gene/Q, was under development. Both delivered versions follow the same design principles, the same system architecture and the same software architecture. They differ on the specifics of the basic SoC that serves as the building block for Blue Gene. The November 2010 TOP500 list includes four Blue Gene/L system and ten Blue Gene/P systems (and one prototype Blue Gene/Q system). In this article we cover mostly the common aspects of both versions of the Blue Gene supercomputing and discuss details specific to each version as appropriate.

### System Architecture

A Blue Gene system consists of a *compute section*, a *file server section*, and a *host section* (Fig. 1). The compute and I/O nodes in the compute section form the computational core of Blue Gene. User jobs run in the compute nodes, while the I/O nodes connect the compute section to the file servers and front-end nodes through an Ethernet network. The file server section consists of a set of file servers. The host section consists of a *service node* and one or more *front-end nodes*. The service node controls the compute section through an Ethernet control network. The front-end nodes provide job compilation, job submission, and job debugging services.

### Compute Section

The compute section of Blue Gene is what is usually called a Blue Gene machine. It consists of a three-dimensional array of compute nodes interconnected in a toroidal topology along the  $x$ ,  $y$ , and  $z$  axes. I/O nodes are distinct from compute nodes and not in the toroidal



**IBM Blue Gene Supercomputer. Fig. 1** High-level view of a Blue Gene system

interconnect, but also belong to the compute section. A collective network interconnects all I/O and compute nodes of a Blue Gene machine. Each I/O node communicates outside of the machine through an Ethernet link.

Compute and I/O nodes are built out of the same Blue Gene compute ASIC (application-specific integrated circuit) and memory (DRAM) chips. The difference is in the function they perform. Whereas compute nodes connect to each other for passing application data, I/O nodes form the interface between the compute nodes and the outside world by connecting to an Ethernet network. Reflecting the difference in function, the software stacks of the compute and I/O nodes are also different, as will be discussed below.

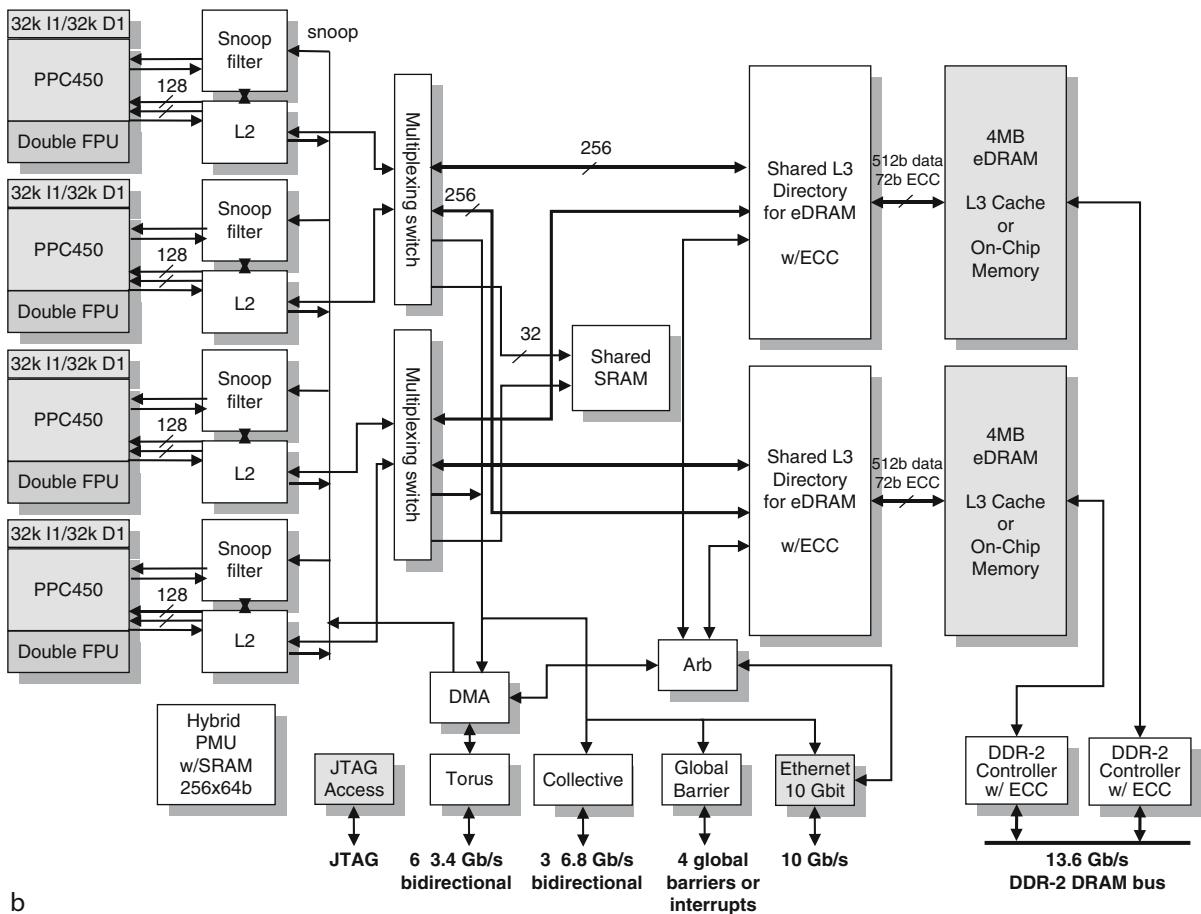
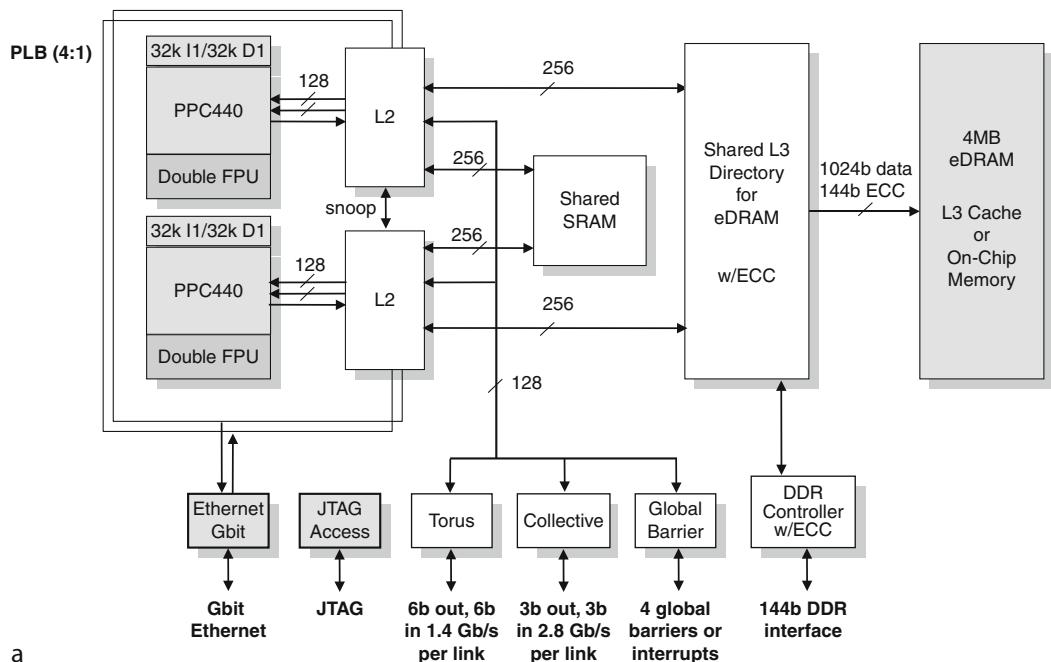
The particular characteristics of the Blue Gene compute ASIC for the two versions (Blue Gene/L and Blue Gene/P) are illustrated in [Fig. 2](#) and summarized in [Table 1](#). The Blue Gene/L compute ASIC contains two non-memory coherent PowerPC 440 cores, each with private L1 data and instruction caches (32 KiB each). Associated with each core is a small (2 KiB) L2 cache that acts as a prefetch engine. Completing the on-chip memory hierarchy is 4 MiB of embedded DRAM (eDRAM) that is configured to operate as a shared L3 cache. Also on the ASIC is a memory controller

(for external DRAM) and interfaces to the five networks used to interconnect Blue Gene/L compute and I/O nodes: torus, collective, global barrier, Ethernet, and control (JTAG) network.

The Blue Gene/P compute ASIC contains four memory coherent PowerPC 450 cores, each with private L1 data and instruction caches (32 KiB each). Associated with each core are both a small (2 KiB) L2 cache that acts as a prefetch engine, as in Blue Gene/L, and a snoop filter to reduce coherence traffic into each core. Two banks of 4 MiB EDRAM are configured to operate as a shared L3 cache. Completing the ASIC are a dual memory controller (for external DRAM), interfaces to the same five networks as Blue Gene/L, and a DMA engine to transfer data directly from the memory of one node to another.

Both the PowerPC 440 and PowerPC 450 cores include a vector floating-point unit that extends the PowerPC instruction set architecture with instructions that assist in matrix and complex-arithmetic operations. The vector floating-point unit can perform two fused multiply-add operations per second for a total of four floating-point operations per cycle per core.

The interconnection networks are primarily used for communication primitives used in parallel high-performance computing applications. The main



IBM Blue Gene Supercomputer. Fig. 2 Blue Gene compute ASIC for Blue Gene/L (**a**) and Blue Gene/P (**b**)

**IBM Blue Gene Supercomputer. Table 1** Summary of differences between Blue Gene/L and Blue Gene/P nodes

| Node characteristics                | Blue Gene/L                      | Blue Gene/P                       |
|-------------------------------------|----------------------------------|-----------------------------------|
| Processors                          | Two PowerPC 440                  | Four PowerPC 450                  |
| Processor frequency                 | 700 MHz                          | 850 MHz                           |
| Peak computing capacity             | 5.6 Gflop/s (4 flops/cycle/core) | 13.6 Gflop/s (4 flops/cycle/core) |
| Main memory capacity                | 512 MiB or 1 GiB                 | 2 GiB or 4 GiB                    |
| Main memory bandwidth               | 5.6 GB/s (16 byte @ 350 MHz)     | 13.6 GB/s (2 × 16 byte @ 425 MHz) |
| Torus link bandwidth (6 links)      | 175 MB/s per direction           | 425 MB/s per direction            |
| Collective link bandwidth (3 links) | 350 MB/s per direction           | 850 MB/s per direction            |

interconnection network is the torus network, which provides point-to-point and multicast communication across compute nodes. A collective network in a tree topology interconnects all compute and I/O nodes and supports efficient collective operations, such as reductions and broadcasts. Arithmetic and logical operations are implemented as part of the communication primitives to facilitate low-latency collective operations. A global barrier network supports fast synchronization and notification across compute and I/O nodes. The Ethernet network is used to connect the I/O nodes outside the machine, as previously discussed. Finally, the control network is used to control the hardware from the service node. [Figure 3](#) illustrates the topology of the various networks.

Compute and I/O nodes are grouped into units of *midplanes*. A midplane contains 16 node cards and each node card contains 32 compute nodes and up to four (Blue Gene/L) or two (Blue Gene/P) I/O nodes. The 512 compute nodes in a midplane are arranged as an  $8 \times 8 \times 8$  three-dimensional mesh. Midplanes are typically configured with 8–32 (Blue Gene/P) or 8–64 (Blue Gene/L) I/O nodes. Each midplane also has 24 link chips used for inter-midplane connection to build larger systems. The link chips also implement the “closing” of the toroidal interconnect, by connecting the two ends of a dimension. Midplanes are arranged two to a rack, and racks are arranged in a two-dimensional layout of rows and columns. Because the midplane is the basic replication unit, the dimensions of the array of compute nodes must be a multiple of 8.

The configuration of the original Blue Gene/L system at Lawrence Livermore National Laboratory (LLNL) is shown in [Fig. 4](#). That machine was later upgraded to 104 racks and is the largest Blue Gene

system delivered to date. The highest performing Blue Gene system delivered to date is a 72-rack Blue Gene/P system at Juelich Research Center. Several single-rack (1,024 compute nodes) Blue Gene systems, as well as systems of intermediate size, were also delivered to various customers.

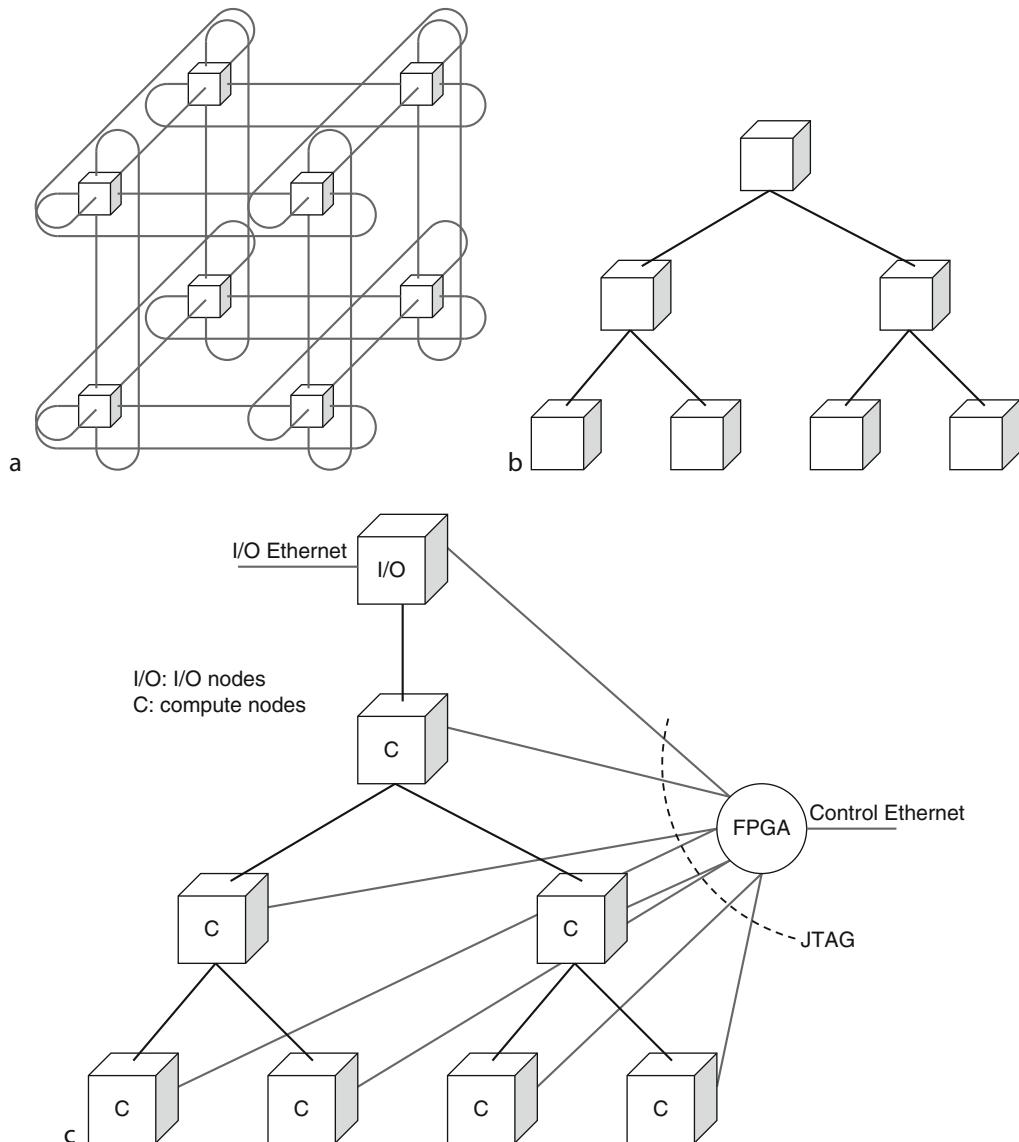
A given Blue Gene machine can be partitioned along midplane boundaries. A partition is formed by a rectangular arrangement of midplanes. Each partition can run only one job at any given time. During each job, all the compute nodes of a partition stay in the same execution mode for the duration of the job. These modes of execution are described in section Overall Operating System Architecture.

## File Server Section

The file server section of a Blue Gene system provides the storage for the file system that runs on the Blue Gene I/O nodes. Several parallel file systems have been ported to Blue Gene, including GPFS, PVFS2, and Lustre. To feed data to a Blue Gene system, multiple servers are required to achieve the required bandwidth. The original Blue Gene/L system at LLNL, for example, uses 224 servers operating in parallel. Data is striped across those servers, and a multi-level switching Ethernet network is used to connect the I/O nodes to the servers. The servers themselves are standard rack-mounted machines, typically Intel, AMD, or POWER processor based.

## Host Section

The host section for a Blue Gene/L system consists of one service node and one or more front-end nodes. These nodes are standard POWER processor machines. For the LLNL machine, the service node is a



**IBM Blue Gene Supercomputer. Fig. 3** Blue Gene networks. Three-dimensional torus (**a**), collective network (**b**), and control network and Ethernet (**c**)

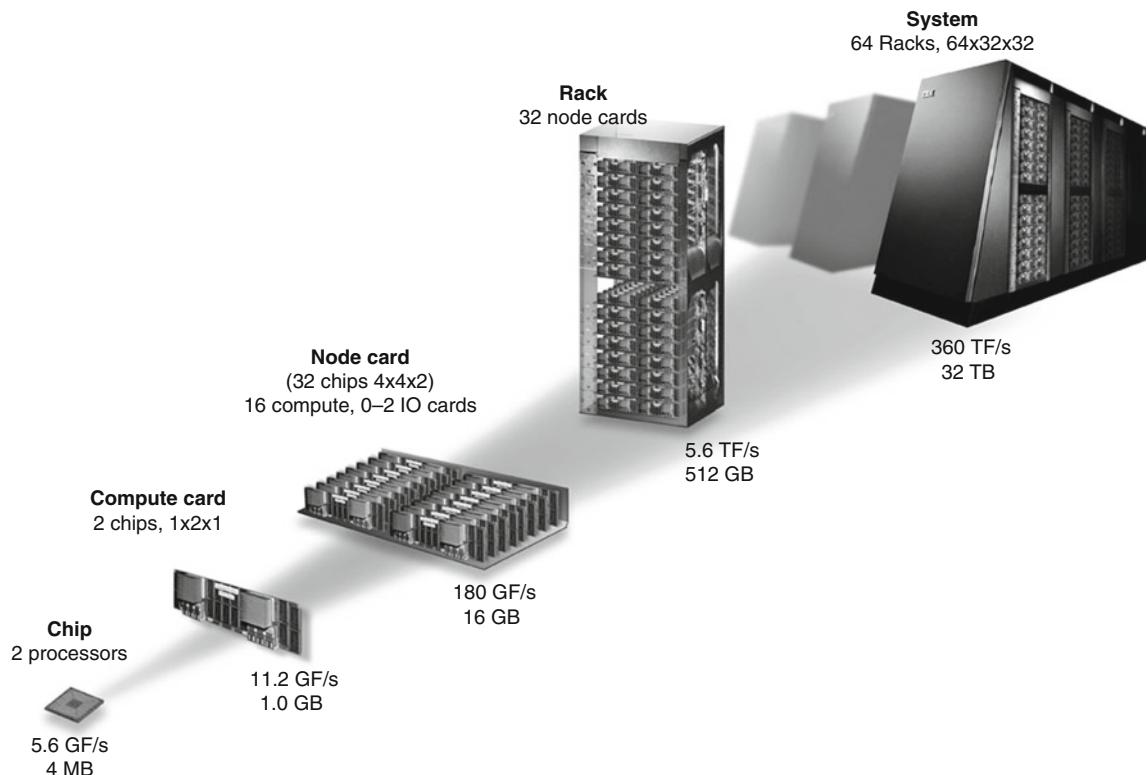
16-processor POWER4 machine, and each of the 14 front-end nodes is a PowerPC 970 blade.

The service node is responsible for controlling and monitoring the operation of the compute section. The services it implements include: machine partitioning, partition boot, application launch, standard I/O routing, application signaling and termination, event monitoring (for events generated by the compute and I/O nodes), and environmental monitoring (for things like power supply voltages, fan speeds, and temperatures).

The front-end nodes are where users work. They provide access to compilers, debuggers, and job submission services. Console I/O from user applications are routed to the submitting front-end node.

### Blue Gene System Software

The primary operating system for Blue Gene compute nodes is a lightweight operating system called the Compute Node Kernel (CNK). This simple kernel implements only a limited set of services, which are



**IBM Blue Gene Supercomputer.** Fig. 4 Packaging hierarchy for the original Blue Gene/L at Lawrence Livermore National Laboratory

complemented by services provided by the I/O nodes. The I/O nodes, in turn, run a version of the Linux operating system. The I/O nodes act as gateways between the outside world and the compute nodes, complementing the services provided by the CNK with file and socket operations, debugging, and signaling.

This split of functions between I/O and compute nodes, with the I/O nodes dedicated to system services and the compute nodes dedicated to application execution, resulted in a simplified design for both components. It also enables Blue Gene scalability and robustness and achieves a deterministic execution environment.

Scientific middleware for Blue Gene includes a user-level library implementation of the MPI standard, optimized to take advantage of Blue Gene networks, and various math libraries, also in the user level. Implementing all the message passing functions in user mode simplifies the supervisor (kernel) code of Blue Gene and results in better performance by reducing the number of kernel system calls that an application performs.

### Overall Operating System Architecture

A key concept in the Blue Gene operating system solution is the organization of compute and I/O nodes into logical entities called processing sets or psets. A pset consists of one I/O node and a collection of compute nodes. Every system partition, in turn, is organized as a collection of psets. All psets in a partition must have the same number of compute nodes, and the psets of a partition must cover all the I/O and compute nodes of the partition. The psets of a partition never overlap. The supported pset sizes are 8 (Blue Gene/L only), 16, 32, 64, and 128 compute nodes, plus the I/O node. The psets are a purely logical concept implemented by the Blue Gene system software stack. They are built to reflect the topological proximity between I/O and compute nodes, thus improving communication performance within a pset.

A Blue Gene job consists of a collection of  $N$  compute processes. Each process has its own private address space and two processes of the same job communicate only through message passing. The

primary communication model for Blue Gene is MPI. The  $N$  compute processes of a Blue Gene job correspond to tasks with ranks 0 to  $N - 1$  in the MPI COMM WORLD communicator. Compute processes run only on compute nodes; conversely, compute nodes run only compute processes.

In Blue Gene/L, the CNK implements two modes of operation for the compute nodes: coprocessor mode and virtual node mode. In the coprocessor mode, the single process in the node has access to the entire node memory. One processor executes user code while the other performs communication functions. In virtual node mode, the node memory is split in half between the two processes running on the two processors. Each process performs both computation and communication functions.

In Blue Gene/P, the CNK provides three modes of operation for the compute nodes: SMP mode, dual mode, and quad mode. The SMP mode supports a single multithreaded (up to four threads) application process per compute node. That process has access to the entire node memory. Dual mode supports two multithreaded (up to two threads each) application processes per compute node. Quad mode supports four single-threaded application processes per compute node. In dual and quad modes, the application processes in a node split the memory of that node. Blue Gene/P supports at most one application thread per processor. It also supports an optional communication thread in each processor.

Each I/O node runs one image of the Linux operating system. It can offer the entire spectrum of services expected in a Linux box, such as multiprocessing, file systems, and a TCP/IP communication stack. These services are used to extend the capabilities of the compute node kernel, providing a richer functionality to the compute processes. Due to the lack of cache coherency between the processors of a Blue Gene/L node, only one of the processors of each I/O node is used by Linux, while the other processor remains idle. Since the processors in a Blue Gene/P node are cache coherent, Blue Gene/P I/O nodes can run a true multiprocessor Linux instance.

## The Compute Node Kernel

CNK is a lean operating system that performs a simple sequence of operations at job start time. This sequence

of operations happens in every compute node of a partition, at the start of each job:

1. It creates the address space(s) for execution of compute process(es) in a compute node.
2. It loads code and initialized data for the executable of that (those) process(es).
3. It transfers processor control to the loaded executable, changing from supervisor to user mode.

The address spaces of the processes are flat and fixed, with no paging. The entire mapping is designed to fit statically in the TLBs of the PowerPC processors. The loading of code and data occurs in push mode. The I/O node of a pset reads the executable from the file system and forwards it to all compute nodes in the pset. The CNK in a compute node receives that executable and stores the appropriate memory values in the address space(s) of the compute process(es).

Once the CNK transfers control to the user application, its primary mission is to “stay out of the way.” In normal execution, the processor control stays with the compute process until it requests an operating system service through a system call. Exceptions to this normal execution are caused by hardware interrupts: either timer alarms requested by the user code, communication interrupts caused by arriving packets, or an abnormal hardware event that requires attention by the compute node kernel.

When a compute process makes a system call, three things may happen:

1. “Simple” system calls that require little operating system functionality, such as getting the time or setting an alarm, are handled locally by the compute node kernel. Control is transferred back to the compute process at completion of the call.
2. “I/O” system calls that require infrastructure for file systems and IP stack are shipped for execution in the I/O node associated with that compute node (i.e., the I/O node in the pset of the compute node). The compute node kernel waits for a reply from the I/O node, and then returns control back to the compute process.
3. “Unsupported” system calls that require infrastructure not present in Blue Gene are returned right away with an error condition.

There are two main benefits from the simple approach for a compute node operating system: robustness and scalability. Robustness comes from the fact that the compute node kernel performs few services, which greatly simplify its design, implementation, and test. Scalability comes from lack of interference with running compute processes.

### System Software for the I/O Node

The I/O node plays a dual role in Blue Gene. On one hand, it acts as an effective master of its corresponding pset. On the other hand, it services requests from compute nodes in that pset. Jobs are launched in a partition by contacting corresponding I/O nodes. Each I/O node is then responsible for loading and starting the execution of the processes in each of the compute nodes of its pset. Once the compute processes start running, the I/O nodes wait for requests from those processes. Those requests are mainly I/O operations to be performed against the file systems mounted in the I/O node.

Blue Gene I/O nodes execute an embedded version of the Linux operating system. It is classified as an embedded version because it does not use any swap space, it has an in-memory root file system, it uses little memory, and it lacks the majority of daemons and services found in a server-grade configuration of Linux. It is, however, a complete port of the Linux kernel and those services can be, and in various cases have been, turned on for specific purposes. The Linux in Blue Gene I/O nodes includes a full TCP/IP stack, supporting communications to the outside world through Ethernet. It also includes file system support. Various network file systems have been ported to the Blue Gene/L I/O node, including GPFS, Lustre, NFS, and PVFS2.

Blue Gene/L I/O nodes never run application processes. That duty is reserved to the compute nodes. The main user-level process running on the Blue Gene/L I/O node is the control and I/O daemon (CIOD). CIOD is the process that links the compute processes of an application running on compute nodes to the outside world. To launch a user job in a partition, the service node contacts the CIOD of each I/O node of the partition and passes the parameters of the job (user ID, group ID, supplementary groups, executable name, starting working directory, command-line arguments, and environment variables). CIOD swaps itself to the user's identity,

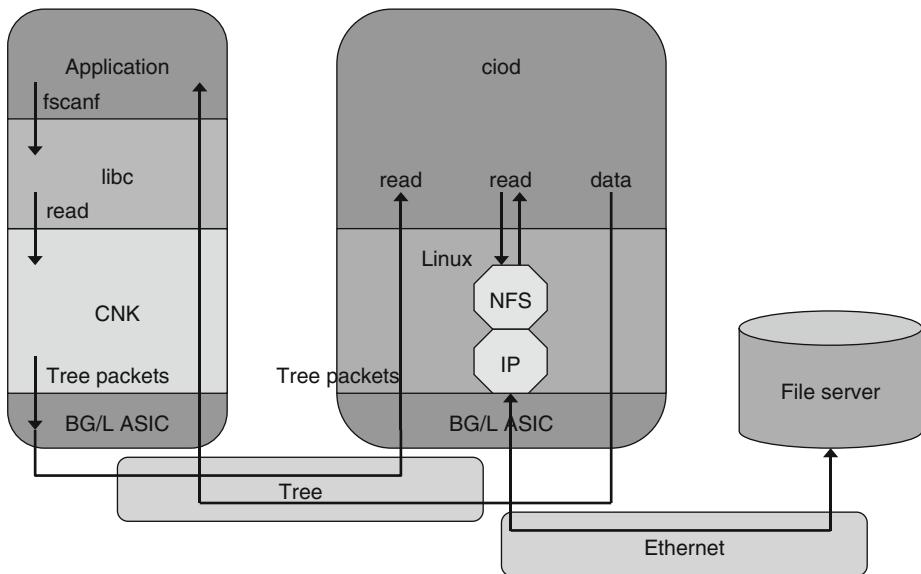
which includes the user ID, group ID, and supplementary groups. It then retrieves the executable from the file system and sends the code and initialized data through the collective network to each of the compute nodes in the pset. It also sends the command-line arguments and environment variables, together with a start signal.

[Figure 5](#) illustrates how I/O system calls are handled in Blue Gene. When a compute process performs a system call requiring I/O (e.g., open, close, read, write), that call is trapped by the compute node kernel, which packages the parameters of the system call and sends that message to the CIOD in its corresponding I/O node. CIOD unpacks the message and then reissues the system call, this time under the Linux operating system of the I/O node. Once the system call completes, CIOD packages the result and sends it back to the originating compute node kernel, which, in turn, returns the result to the compute process.

There is a synergistic effect between simplification and separation of responsibilities. By offloading complex system operations to the I/O node, Blue Gene keeps the compute node operating system simple. Correspondingly, by keeping application processes separate from the I/O node activity, it avoids many security and safety issues regarding execution in the I/O nodes. In particular, there is never a need for the common scrubbing daemons typically used in Linux clusters to clean up after misbehaving jobs. Just as keeping system services in the I/O nodes prevents interference with compute processes, keeping those processes in compute nodes prevents interference with system services in the I/O node. This isolation is particularly helpful during performance debugging work. The overall simplification of the operating system has enabled the scalability, reproducibility (performance results for Blue Gene applications are very close across runs), and high-performance of important Blue Gene/L applications.

### System Software for the Service Node

The Blue Gene service node runs its control software, typically referred to as the Blue Gene control system. The control system is responsible for the operation and monitoring of all compute and I/O nodes. It is also responsible for other hardware components such as link chips, power supplies, and fans. Tight integration between the Blue Gene control system and the I/O and compute nodes operating systems is central to the Blue



**IBM Blue Gene Supercomputer. Fig. 5** Function shipping between CNK and CIOD

Gene software stack. It represents one more step in the specialization of services that characterize that stack.

In Blue Gene, the control system is responsible for setting up system partitions and loading the initial code and state in the nodes of a partition. The Blue Gene compute and I/O nodes are completely stateless: no hard drives and no persistent memory. When a partition is created, the control system programs the hardware to isolate that partition from others in the system. It computes the network routing for the torus, collective, and global interrupt networks, thus simplifying the compute node kernel. It loads the operating system code for all compute and I/O nodes of a partition through the dedicated control network. It also loads an initial state in each node (called the personality of the node). The personality of a node contains information specific to the node.

### Blue Gene and Its Impact on Science

Blue Gene was designed to deliver a level of performance for scientific computing that enables entire new studies and new applications. One of the main areas of applications of the original LLNL system is in materials' science. Scientists at LLNL use a variety of models, including quantum molecular dynamics, classical molecular dynamics, and dislocation dynamics, to study materials at different levels of resolution.

Typically, each model is applicable to a range of system sizes being studied. First principle models can be used for small systems, while more phenomenological models have to be used for large systems. The original Blue Gene/L at LLNL was the first system that allowed crossing of those boundaries. Scientists can use first principle models in systems large enough to validate the phenomenological models, which in turn can be used in even larger systems.

Applications of notable significance at LLNL include ddcMD, a classical molecular dynamics code that has been used to simulate systems with approximately half a billion atoms (and in the process win the 2005 Gordon Bell award), and QBox, a quantum molecular dynamics code that won the 2006 Gordon Bell award. Other success stories for Blue Gene in science include: (1) astrophysical simulations in Argonne National Laboratory (ANL) using the FLASH code; (2) global climate simulations in the National Center for Atmospheric Research (NCAR) using the HOMME code; (3) biomolecular simulations at the T.J. Watson Research Center using the Blue Matter code; and (4) quantum chromo dynamics (QCD) at IBM T.J. Watson Research Center, LLNL, San Diego Supercomputing Center, Juelich Research Center, Massachusetts Institute of Technology, Boston University, University of Edinburgh, and KEK (Japan) using a variety of codes.

One of the most innovative uses of Blue Gene is as the central processor for the large-scale LOFAR radio telescope in the Netherlands.

## Related Entries

- [IBM Power Architecture](#)
- [LINPACK Benchmark](#)
- [MPI \(Message Passing Interface\)](#)
- [TOP500](#)

## Bibliographic Notes and Further Reading

Additional information about the system architecture of Blue Gene can be found in [6, 8, 10, 11, 15]. For details on the Blue Gene system software, the reader is referred to [9, 10]. Finally, examples of the impact of Blue Gene on science can be found in [1–5, 7, 12–14].

## Bibliography

1. Almasi G, Chatterjee S, Gara A, Gunnels J, Gupta M, Henning A, Moreira JE, Walkup B (2004) Unlocking the performance of the BlueGene/L supercomputer. In: Proceeding IEEE/ACM SC04, Pittsburgh, Nov 2004
2. Almasi G, Bhanot G, Gara A, Gupta M, Sexton J, Walkup B, Bulatov VV, Cook AW, de Supinski BR, Glosli JN, Greenough JA, Gygi F, Kubota A, Louis S, Streitz FH, Williams PL, Yates RK, Archer C, Moreira J, Rendleman C (2005) Scaling physics and material science applications on a massively parallel Blue Gene/L system. In: Proceedings of the 19th ACM international conference on supercomputing. Cambridge, MA, 246–252, 20–22 June 2005
3. Bulatov V, Cai W, Fier J, Hiratani M, Hommes G, Pierce T, Tang M, Rhee M, Yates RK, Arsenlis T (2004) Scalable line dynamics in ParaDiS. In: Proceeding IEEE/ACM SC04, Pittsburgh, Nov 2004
4. Fitch BG, Rayshubskiy A, Eleftheriou M, Ward TJ, Giampapa M, Pitman MC, Germain RS (2006) Blue matter: approaching the limits of concurrency for classical molecular dynamics. In: Proceeding IEEE/ACM SC06, Tampa, Nov 2006
5. Fryxell B, Olson K, Ricker P, Timmes FX, Zingale M, Lamb DQ, MacNeice P, Rosner R, Truran JW, Tufo H (2000) FLASH: an adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *Astrophys J Suppl* 131:273
6. Gara A, Blumrich MA, Chen D, Chiu GL-T, Coteus P, Giampapa ME, Haring RA, Heidelberger P, Hoenicke D, Kopcsay GV, Liebsch TA, Ohmacht M, Steinmacher-Burow BD, Takken T, Vranas P (2005) Overview of the Blue Gene/L system architecture. *IBM J Res Dev* 49(2/3):195–212
7. Gygi F, Yates RK, Lorenz J, Draeger EW, Franchetti F, Ueberhuber CW, de Supinski BR, Kral S, Gunnels JA, Sexton JC (2005) Large-scale first-principles molecular dynamics simulations on the BlueGene/L platform using the Qbox code. *IEEE/ACM SC05*, Seattle, Nov 2005
8. IBM Blue Gene Team (2008) Overview of the IBM Blue Gene/P project. *IBM J Res Dev* 52(1/2):199–220
9. Moreira JE, Almasi G, Archer C, Bellofatto R, Bergner P, Brunheroto JR, Brutman M, Castaños JG, Crumley PG, Gupta M, Inglett T, Lieber D, Limpert D, McCarthy P, Megerian M, Mendell M, Mundy M, Reed D, Sahoo RK, Sanomiya A, Shok R, Smith B, Stewart GG (2005) Blue Gene/L programming and operating environment. *IBM J Res Dev* 49(2/3):367–376
10. Moreira JE et al (2007) The Blue Gene/L supercomputer: a hardware and software story. *Int J Parallel Program* 35(3):181–206
11. Salapura V, Bickford R, Blumrich M, Bright AA, Chen D, Coteus P, Gara A, Giampapa M, Gschwind M, Gupta M, Hall S, Haring RA, Heidelberger P, Hoenicke D, Kopcsay GV, Ohmacht M, Rand RA, Takken T, Vranas P (2005) Power and performance optimization at the system level. In: Proceeding ACM Computing Frontiers 2005, Ischia, May 2005
12. van der Schaa K (2005) Blue Gene in the heart of a wide area sensor network. In: Proceeding QCDOC and Blue Gene: next generation of HPC architecture workshop, Edinburgh, Oct 2005
13. Streitz FH, Glosli JN, Patel MV, Chan B, Yates RK, de Supinski BR, Sexton J, Gunnels JA (2005) 100+ TFlop solidification simulations on BlueGene/L. In: Proceeding IEEE/ACM SC05, Seattle, Nov 2005
14. Vranas P, Bhanot G, Blumrich M, Chen D, Gara A, Heidelberger P, Salapura V, Sexton JC (2006) The BlueGene/L supercomputer and quantum chromodynamics. In: Proceeding IEEE/ACM SC06, Tampa, Nov 2006
15. Wait CD (2005) IBM PowerPC 440 FPU with complex-arithmetic extensions. *IBM J Res Dev* 49(2/3):249–254

## IBM Power

- [IBM Power Architecture](#)

## IBM Power Architecture

TEJAS S. KARKHANIS, JOSÉ E. MOREIRA  
IBM T.J. Watson Research Center, Yorktown Heights, NY, USA

## Synonyms

[IBM power](#); [IBM powerPC](#)

## Definition

The IBM Power architecture is an instruction set architecture (ISA) implemented by a variety of processors from IBM and other vendors, including Power7, IBM's

latest server processor. The IBM Power architecture is designed to exploit parallelism at the instruction-, data-, and thread-level.

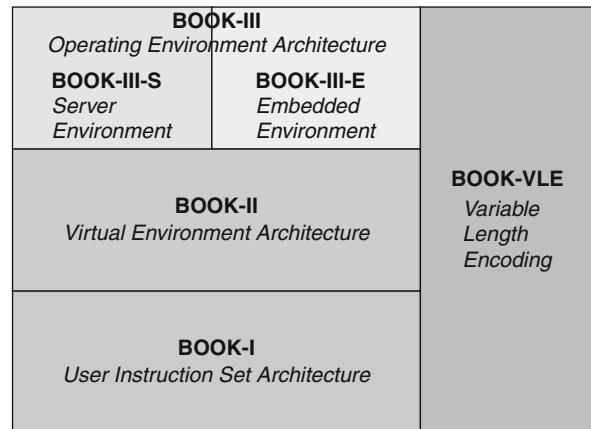
## Discussion

### Introduction

IBM's Power ISA™ is an instruction set architecture designed to expose and exploit parallelism in a wide range of applications, from embedded computing to high-end scientific computing to traditional transaction processing. Processors implementing the Power ISA have been used to create several notable parallel computing systems, including the IBM RS/6000 SP, the Blue Gene family of computers, the Deep Blue chess playing machine, the PERCS system, the Sony Playstation 3 game console, and the Watson system that competed in the popular television show Jeopardy!

Power ISA covers both 32-bit and 64-bit variants and, as of its latest version (2.06 Revision B [8]), is organized in a set of four “books,” as shown in Fig. 1. Books I and II are common to all implementations. Book I, Power ISA User Instruction Set Architecture, covers the base instruction set and related facilities available to the application programmer. Book II, Power ISA Virtual Environment Architecture, defines the storage (memory) model and related instructions and facilities available to the application programmer. In addition to the specifications of Books I and II, implementations of the Power ISA need to follow either Book III-S or Book III-E. Book III-S, Power ISA Operating Environment Architecture – Server Environment, defines the supervisor instructions and related facilities used for general purpose implementations. Book III-E, Power ISA Operating Environment Architecture – Embedded Environment, defines the supervisor instructions and related facilities used for embedded implementations. Finally, Book VLE, Power ISA Operating Environment Architecture – Variable Length Encoding Environment, defines alternative instruction encodings and definitions intended to increase instruction density for very low end implementations.

Figure 2 shows the evolution of the main line of Power architecture server processors from IBM, just one of the many families of products based on Power ISA. The figure shows, for each generation of processors, its introduction date, the silicon technology used,



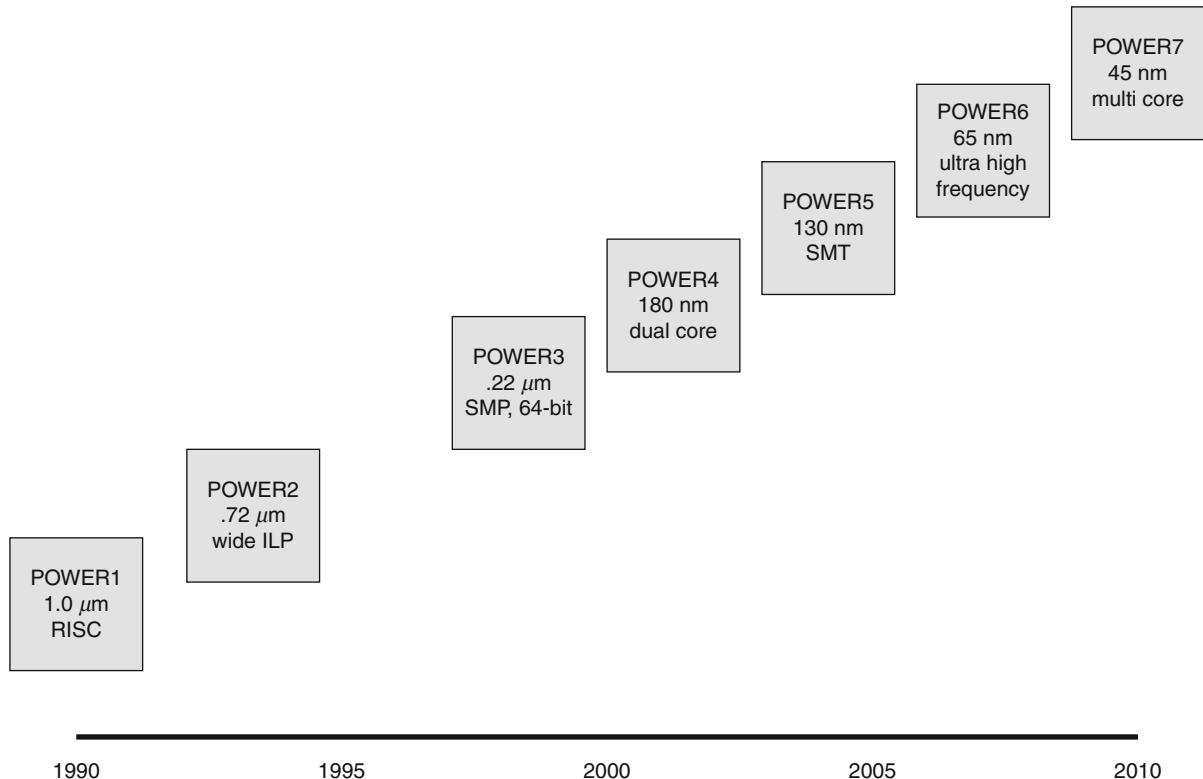
**IBM Power Architecture. Fig. 1** Books of Power ISA version 2.06

and the main architectural innovations delivered in that generation. Power7 [9] is IBM's latest-generations server processor and implements the Power ISA according to Books I, II, and III-S of [8]. Power7 is used in both the PERCS and Watson systems and in a variety of servers offered by IBM. The latest machine in the Blue Gene family, Blue Gene/Q, follows a Book III-E implementation.

Power ISA was designed to support high program execution performance and efficient utilization of hardware resources. To that end, Power ISA provides facilities for expressing instruction-level parallelism, data-level parallelism, and thread-level parallelism. Providing facilities for a variety of parallelism types gives the programmer the flexibility in extracting the particular combination of parallelism that is optimal for his or her program.

### Instruction-Level Parallelism

Instruction-level parallelism (ILP) is the simultaneous processing of several instructions by a processor. ILP is important for performance because it allows instructions to overlap, thus effectively hiding the execution latency of long latency computational and memory access instructions. Achieving ILP has been so important in the processor industry that processor core designs have gone from simple multicycle designs to complex designs that implement superscalar pipelines and out-of-order execution [15]. Key aspects of Power



**IBM Power Architecture. Fig. 2** Evolution of main line Power architecture server processors

ISA that facilitate ILP are independent instruction facilities, reduced set of instructions, fixed length instructions, and large register set.

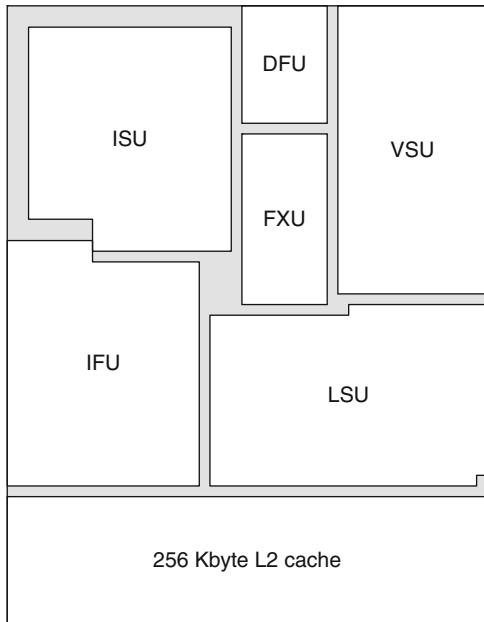
### Independent Instruction Facilities

Conceptually, Power ISA views the underlying processor as composed of several engines or units, as illustrated in the floor plan for the Power7 processor core shown in [Fig. 3](#). Book I of the Power ISA groups the instructions into “facilities,” including (1) the branch facility, with instructions implemented by the instruction fetch unit (IFU); (2) the fixed-point facility, with instructions implemented by the fixed-point unit (FXU) and load-store unit (LSU); (3) the floating-point facility, with instructions implemented by the vector and scalar unit (VSU); (4) the decimal floating-point facility, with instructions implemented by the decimal floating-point unit (DFU); and (5) the vector facility, with instructions implemented by the vector and scalar

unit (VSU). Also shown in [Fig. 3](#) is the instruction-sequencing unit (ISU), which controls the execution of instructions, and a level-2 cache.

Origins of this conceptual decomposition are in the era of building processors with multiple integrated circuits (chips). With a single processor spread across multiple chips, the communication between two integrated circuits took significantly more time relative to communication internal to a chip. Consequently, either the clock frequency would have to be reduced or the number of stages in the processor pipeline would have to be increased. Both approaches, reducing clock frequency and increasing the number of pipeline stages, can degrade performance. The decomposition into multiple units allowed a clear separation of work, and each unit could be implemented on a single chip for maximum performance.

Today, the conceptual decomposition provides two primary benefits. First, because of the conceptual decomposition, the interfaces between the engines



**IBM Power Architecture. Fig. 3** Power7 processor core floor plan, showing the main units

are clearly defined. Clearly defined interfaces lead to hardware design that is simpler to implement and to verify. Second, the conceptual decomposition addresses the inability of scaling frequency of long on-chip wires that can be a performance limiter, just as it addressed wiring issues between two or more integrated circuits when the conceptual decomposition was introduced.

### Reduced Set of Nondestructive Fixed Length Instructions

Power ISA consists of a reduced set of fixed length 32-bit instructions. A large fraction of the set of instructions are nondestructive. That is, the result register is explicitly identified, as opposed to implicitly being one of the source registers. A reduced set of instructions simplifies the design of the processor core and also verification of corner cases in the hardware.

Ignoring the Book-VLE case, which is targeted to very low end systems, Power ISA instructions are all 32-bits in length, thus the beginning and end of every instruction is known before decode. The bits in an instruction word are numbered from 0 (most significant) to 31 (least significant), following the big-endian

convention of the Power architecture. All Power ISA instructions have a major opcode that is located at instruction bits 0–5. Some instructions also have a minor opcode, to differentiate among instructions with the same major opcode. The location and length of the minor opcode depends on the major opcode. Additionally, every instruction is word-aligned. Fixed length, word aligned, and fixed opcode location make the instruction predecode, fetch, branch prediction, and decode logic simpler, when compared to the decode logic of variable length ISA. Srinivasan et al. [16] present a comprehensive study of optimality of pipeline length of Power processors from a power and performance perspective.

Instruction set architectures that employ destructive operations (i.e., one of the source registers is also the target) must temporarily save one of the source registers, if the contents of that register are required later in the program. Temporarily saving and later restoring registers often lead to store to and load from, respectively, a memory location. Memory operations can take longer to complete than a computational operation. Nondestructive operations in Power ISA eliminate the need for extra instructions for saving and restoring one of the source registers, facilitating higher instruction level parallelism.

### Large Register Set

Power ISA originally specified 32 general-purpose (fixed-point, either 32- or 64-bit) and 32 floating-point (64-bit) registers. An additional set of 32 vector (128-bit) registers were added with the first set of vector instructions. The latest specification, Power ISA 2.06, expands the number of vector registers to 64. A large number of registers means that more data, including function and subroutine parameters, can be kept in fast registers. This in turn avoids load/store operations to save and retrieve data to and from memory and supports concurrent execution of more instructions.

### Load/Store Architecture

Power ISA specifies a load-store architecture consisting of two distinct types of instructions: (1) memory access instructions, and (2) compute instructions. Memory

access instructions load data from memory into computational registers and store the data from the computational registers to the memory. Compute instructions perform computations on the data residing in the computational registers. This arrangement decouples the responsibilities of the memory instructions and computational instructions, providing a powerful lever to hide the memory access latency by overlapping the long latency of memory access instructions with compute instructions.

### **ILP in Power7**

Power7 is an out-of-order superscalar processor that can operate at frequencies exceeding 4 GHz. In a given clock cycle, a Power7 processor core can fetch up to eight instructions, decode and dispatch up to six instructions, issue and execute up to eight instructions, and commit up to six instructions. To ensure a high instruction throughput, Power7 can simultaneously maintain about 250 instructions in various stages of processing. To further extract independent instruction for parallelism, Power7 implements register renaming – each of the architected register files are mapped to a much larger set of physical registers. Execution of the instructions is carried out by a total of 12 execution units. Power7 implements the Power ISA in a way that extracts high levels of instruction-level parallelism while operating at a high clock frequency.

### **Data Level Parallelism**

Data-level parallelism (DLP) consists of simultaneously performing the same type of operations on different data values, using multiple functional units, with a single instruction. The most common approach of providing DLP in general purpose processors is the Single Instruction Multiple Data (SIMD) technique. SIMD (also called vector) instructions provide a concise and efficient way to express DLP. With SIMD instructions, fewer instructions are required to perform the same data computation resulting in lower fetch, decode and dispatch bandwidth, and consequently higher power efficiency.

Power ISA 2.06 contains two sets of SIMD instructions. The first one is the original set of instructions implemented by the vector facility since 1998 and also known as AltiVec [6] or Vector Media Extensions

(VMX) instructions. The second is a new set of SIMD instructions called Vector-Scalar Extension (VSX).

### **VMX Instructions**

VMX instructions operate on 128-bit wide data, which can be vectors of byte (8-bit), half-word (16-bit) and word (32-bit) elements. The word elements can be either integer or single-precision floating-point numbers. The VMX instructions follow the load/store model, with a 32-entry register set (each entry is 128-bit wide) that is separate from the original (scalar) fixed- and floating-point registers in Power ISA 2.06.

### **VSX Instructions**

VSX also operates on 128-bit wide data, which can be vectors of word (32-bit) and double word (64-bit) elements. Most operations are on floating-point numbers (single and double precision) but VSX also includes integer conversion and logical operations. VSX instructions also follow the load/store model, with a 64-entry register set (128 bits per entry) that overlaps the VMX and floating-point registers. VSX requires no operating-mode switches. Therefore, it is possible to interleave VSX instructions with floating point and integer instructions.

### **Power7 Vector and Scalar Unit (VSU)**

The vector and scalar unit of Power7 is responsible for execution of the VMX and VSX SIMD instructions. The unit contains one vector pipeline and four double-precision floating-point pipelines. A VSX floating-point instruction uses two floating-point pipelines, and two VSX instructions can be issued every cycle, to keep all floating-point pipelines busy. The four floating-point pipelines can each execute a double-precision fused multiply-add operation, leading to a performance of 8 flops/cycle for a Power7 core.

### **Thread Level Parallelism**

Thread-level parallelism (TLP) is the simultaneous execution of multiple threads of instructions. Unlike ILP and DLP, that rely on extracting parallelism from within the same program thread, TLP relies on explicit parallelism from multiple concurrently running threads. The multiple threads can come from the decomposition of

a single program or from multiple independent programs.

### Thread Level Parallelism Within a Processor Core

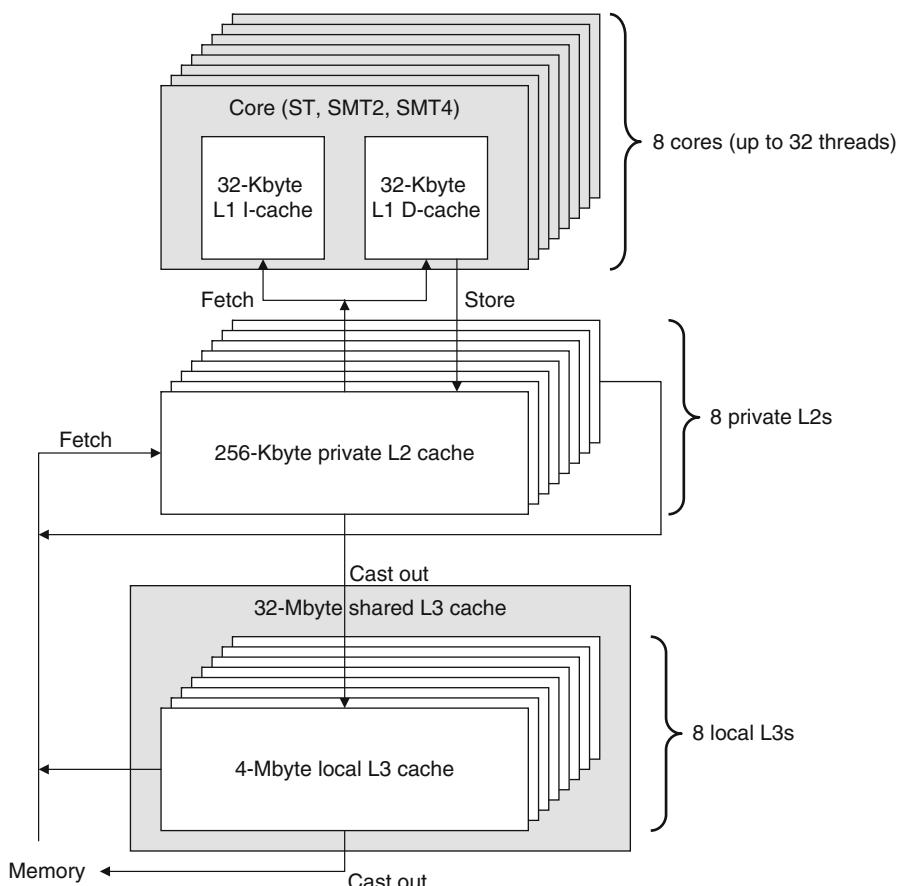
In the first systems that exploited thread-level parallelism, different threads executed on different processor cores and shared a memory system. Today, processor core designs have evolved such that multiple threads can run on single processor core. This increases resource utilization and, consequently, the computational throughput of the core. Effectively, TLP within a core enables hiding of the long latency events of stalled threads with forward progress of active threads. Examples of Power ISA processors that support multithreading within a core include Power5 [14], Power6 [10], and Power7 [9] processors.

### Memory Coherence Models

For programs where the concurrent threads share memory while working on a common task, the memory consistency model of the architectures plays a key role in the performance of TLP as a function of the number of threads. The memory consistency model specifies how memory references from different threads can be interleaved. Power ISA specifies a *release consistency* memory model. A release consistency model relaxes the ordering of memory references as seen by different threads. When a particular ordering of memory references among threads is necessary for the program, explicit synchronization operations must be used.

### TLP Support in Power7 Processor

The structure of a Power7 processor chip is shown in Fig. 4. There are eight processor cores and three levels of cache in a single chip. Each processor core (which



IBM Power Architecture. Fig. 4 Structure of a Power7 Processor Chip

includes 32-Kbyte level 1 data and instruction caches) is paired with a 256-Kbyte level 2 (L2) cache that is private to the core. There is also a 32-Mbyte level 3 (L3) cache that is shared by all cores. The level 3 cache is organized as eight 4-Mbyte caches, each local to a core/L2 pair. Cast outs from an L2 cache can only go to its local L3 cache, but from there data can be cast out across the eight local L3s.

Each core is capable of operating in three different threading modes: single-threaded (ST), dual-threaded (SMT2), or quad-threaded (SMT4). The cores can switch modes while executing, thus adapting to the needs of different applications. The ST mode delivers higher single-thread performance, since the resources of a core are dedicated to the execution of that single thread. The SMT4 mode partitions the core resources among four threads, resulting in higher total throughput at the cost of reduced performance for each thread. The SMT2 mode is an intermediate point.

A single Power7 processor chip supports up to 32 simultaneous threads of execution (8 cores, 4 threads per core). Power7 scales to systems of 32 processor chips or up to 1024 threads of execution sharing a single memory image.

## Summary

Since its inception in the Power1 processor in 1991, the Power architecture has evolved to address the technology and applications issues of the time. The ability of Power architecture to provide instruction-, data-, and thread-level parallelism has enabled a variety of parallel systems, including some notable supercomputers.

Power ISA allows exposing and extraction of ILP primarily because of the RISC principles embodied in the ISA. The reduced set of fixed length instructions enables simple hardware implementation that can be efficiently pipelined, thus increasing concurrency. The larger register set provides several optimization opportunities for the compiler as well as the hardware.

Power ISA provides facilities for data-level parallelism via SIMD instructions. VMX and VSX instructions increase the computational efficiency of the processor by performing the same operation on multiple data values. For some programs DLP can be extracted automatically by the compiler. For others, explicit SIMD programming is more appropriate.

Power ISA supports thread-level parallelism through a release consistency memory model. Because of the release consistency, Power ISA based systems permit aggressive software and hardware optimizations that would otherwise be restricted under a sequential consistency model.

The Power7 processor implements the latest version of Power ISA and exploits all forms of parallelism supported by the instruction set architecture: instruction-level parallelism, data-level parallelism, and thread-level parallelism.

## Related Entries

- ▶ [IBM Blue Gene Supercomputer](#)
- ▶ [Cell Broadband Engine Processor](#)
- ▶ [IBM RS/6000 SP](#)
- ▶ [PERCS System Architecture](#)

## Bibliographic Notes

Official information on the Power instruction set architecture is available in the POWER.ORG website ([www.power.org](http://www.power.org)). In particular, the latest version of the Power ISA (2.06 revision B), implemented by the Power7 processor, can be found in [8].

An early history of the Power architecture is provided by Diefendorff [3]. Details of Power architecture including the instruction specification and programming environment are given in several reference manuals [7, 11–13].

Evolution of IBM's RISC philosophy is explained in [2]. More detailed information on the microarchitecture of specific Power processors can be found for Power4 [17], Power5 [14], Power6 [10], and Power7 [9] processors.

The AltiVec Programming Environment Manual [12] and AltiVec Programming Interface Manual [13] are two thorough references for effectively employing AltiVec. Gwennap [6] and Diefendorff [4] have a good survey of Power AltiVec.

Methods for extracting instruction-level parallelism for the Power architecture are described in [7]. One of the key impediments to data-level parallelism is unaligned memory accesses. To overcome these unaligned accesses Eichenberger, Wu and O'Brien [5] present some data-level parallelism optimization techniques. Finally, Adve and Gharacharloo's tutorial on

shared memory consistency model [1] is a great reference for further reading on thread-level parallelism.

## Bibliography

1. Adve SV, Gharachorloo K (1996) Shared memory consistency models: a tutorial. *IEEE Comput* 29:66–76
2. Cocke J, Markstein V (1990) The evolution of RISC technology at IBM. *IBM J Res Dev* 40:48–55
3. Diefendorff K (1994) History of the PowerPC architecture. *Commun ACM* 37(6):28–33
4. Diefendorff F, Dubey P, Hochsprung R, Scales H (2000) AltiVec extension to PowerPC accelerates media processing. *IEEE Micro* 20(2):85–95
5. Eichenberger AE, Wu P, O'Brien K (2004) Vectorization for SIMD architectures with alignment constraints. *Sigplan Not* 39(6):82–93
6. Gwennap L (1998) AltiVec vectorizes PowerPC. *Microprocessor Report* 12(6):1–5
7. Hoxey S, Karim F, Hay B, Warren H (eds) (1996) The PowerPC compiler writer's guide. Warthman Associates, Palo Alto
8. IBM Power ISA Version 2.06 Revision B. [http://www.power.org/resources/downloads/PowerISA\\_V2.06B\\_V2\\_PUBLIC.pdf](http://www.power.org/resources/downloads/PowerISA_V2.06B_V2_PUBLIC.pdf)
9. Kalla R, Sinharoy B, Starke WJ, Floyd M (2010) Power7: IBM's next-generation server processor. *IEEE Micro* 30(2):7–15
10. Le HQ, Starke WJ, Fields JS, O'Connell FP, Nguyen DQ, Ronchetti BJ, Sauer WM, Schwarz EM, Vaden MT (2007) IBM Power6 microarchitecture. *IBM J Res Dev* 51(6):639–662
11. May C, Silha ED, Simpson R, Warren H (eds) (1994) The PowerPC architecture: a specification for a new family of RISC processors. Morgan Kaufmann, San Francisco
12. (1999) Freescale Semiconductor. In: AltiVec technology programming environments manual.
13. (1999) Freescale Semiconductor. In: AltiVec technology programming interface manual.
14. Sinharoy B, Kalla RN, Tendler JM, Eickemeyer RJ, Joyner JB (2005) Power5 system microarchitecture. *IBM J Res Dev* 49(4/5):505–521
15. Smith JE, Sohi GS (1995) The microarchitecture of superscalar processors. *P IEEE* 83:1609–1624
16. Srinivasan V, Brooks D, Gschwind M, Bose P, Zyuban V, Strenski PN, Emma PG (2002) Optimizing pipelines for power and performance. In: Proceedings of the 35th annual ACM/IEEE international symposium on microarchitecture, MICRO 35. IEEE Computer Society Press, Los Alamitos, pp 333–344
17. Tendler JM, Dodson JS, Fields JS, Le H, Sinharoy B (2002) Power4 system microarchitecture. *IBM J Res Dev* 46(1):5–25

## IBM PowerPC

### ► IBM Power Architecture

## IBM RS/6000 SP

JOSÉ E. MOREIRA

IBM TJ. Watson Research Center, Yorktown Heights, NY, USA

### Synonyms

[IBM SP](#); [IBM SP1](#); [IBM SP2](#); [IBM SP3](#)

### Definition

The IBM RS/6000 SP is a distributed memory, message passing parallel system based on the IBM POWER processors. Several generations of the system were developed by IBM and thousands of systems were delivered to customers. The pinnacle of the IBM RS/6000 SP was the ASCI White machine at Lawrence Livermore National Laboratory, which held the number 1 spot in the TOP500 list from November 2000 to November 2001.

### Discussion

#### Introduction

The IBM RS/6000 SP (SP for short) is a general-purpose parallel system. It was one of the first parallel systems designed to address both technical computing applications (the usual domain of parallel supercomputers) and commercial applications (e.g., database servers, transaction processing, multimedia servers). The SP is a distributed memory, message passing parallel system. It consists of a set of nodes, each running its own operating system image, interconnected by a high-speed network. In the TOP500 classification, it falls into the cluster class of machines.

IBM delivered several generations of SP machines, all based on IBM POWER processors. The initial machines were simply called the IBM SP (or SP1) and were based on the original POWER processors. Later, IBM delivered the SP2 machines based on POWER2 and then finally a generation based on POWER3 processors (which was unofficially called the SP3 by some). IBM continued to deliver parallel systems based on later generations of POWER processors (POWER4 and beyond), but those were no longer considered IBM RS/6000 SP systems. The November 2010 TOP500 list shows 16 POWER Systems 575 machines (one based on

POWER5 processors and the rest based on POWER6 processors), which can be considered direct follow-on to the RS/6000 SP.

Notable IBM RS/6000 SP systems include the Argonne National Laboratory SP1 (installed in 1993) [6], the Cornell Theory Center IBM SP2, the Lawrence Livermore National Laboratory ASCI Blue Pacific, and the Lawrence Livermore National Laboratory ASCI White. This last system consisted of 512 nodes with 16 POWER3 processors each and held the number 1 spot in the TOP500 list from November 2000 to November 2001.

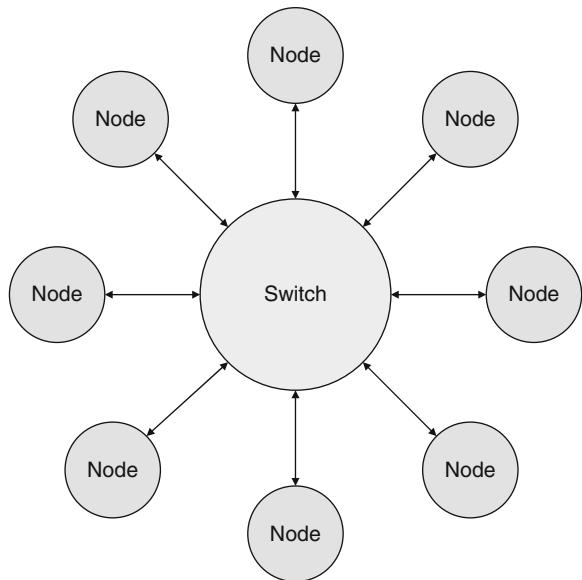
The RS/6000 SP was designed to serve a broad range of applications, from both the technical and commercial computing domains. The designers of the system followed a set of principles [1] that can be summarized as follows: maximize the use of off-the-shelf hardware and software components while developing some special-purpose components that maximize the value of the system. As a result, the RS/6000 SP utilizes the same processors, operating systems, and compilers as the contemporary IBM workstations. It also utilizes a special-purpose high-speed interconnect switch, a parallel operating environment and message passing libraries, and a parallel programming environment, including a High Performance Fortran (HPF) compiler.

To further enable the system for commercial applications, IBM and other vendors developed parallel versions of important commercial middleware such as DB2 and CICS/6000. With these parallel middleware, customers were able to quickly port applications from the more conventional single system image commercial servers to the IBM SP.

## Hardware Architecture

The IBM RS/6000 SP consists of a cluster of nodes interconnected by a switch (Fig. 1). The nodes (Fig. 2) are independent computers based on hardware (processors, memory, disks, I/O adapters) developed for IBM workstations and servers. Each node has its own private memory and runs its own image of the AIX operating system.

Through the evolution of the RS/6000 SP, different nodes were used. The initial nodes for SP1 and SP2 models were single processor nodes. Later, with the introduction of PowerPC and POWER3

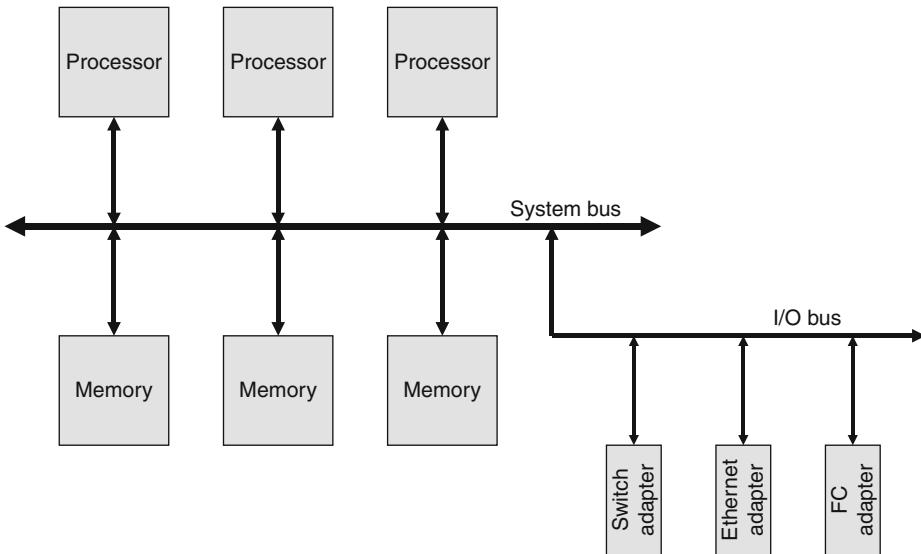


**IBM RS/6000 SP. Fig. 1** High-level hardware architecture of an IBM RS/6000 SP

processors, symmetric multiprocessing (SMP) nodes became available. Nodes could be configured to better serve specific purposes. For example, compute nodes could be configured with more processors and memory, whereas I/O nodes could be configured with more I/O adapters. The RS/6000 SP architecture supports different kinds of nodes in the same system, and it was usual to have both compute-optimized and I/O-optimized nodes in the same system.

The node architecture (illustrated in Fig. 2) is essentially the same as contemporary standalone workstations and servers based on POWER processors. Processor and memory modules are interconnected by a system bus that supports memory coherence within the node. An I/O bus also hangs off this system bus. This I/O bus supports off-the-shelf adapters found on standalone machines, such as Ethernet and Fibre Channel. It also supports the switch adapters that connect the node to the network.

Whereas the SP nodes are built primarily out of off-the-shelf hardware (except for the switch adapter), the SP switch is a special-purpose design. At the time the SP was conceived, standard interconnection networks (Ethernet, FDDI, ATM) delivered neither the bandwidth nor the latency that a large-scale general-purpose

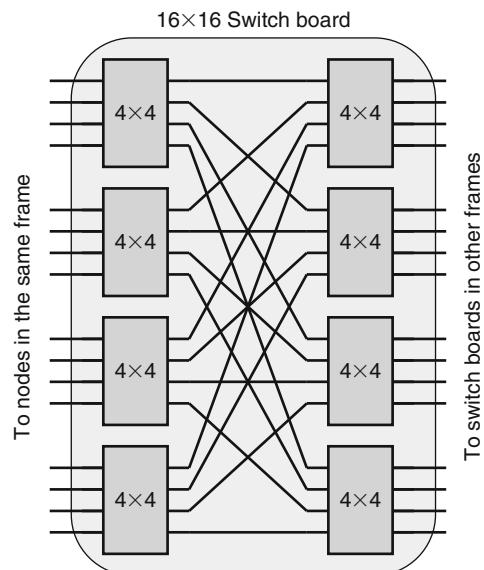


**IBM RS/6000 SP. Fig. 2** Hardware architecture of an IBM RS/6000 SP node. Different kinds of nodes can be and have been used in SP systems

parallel system like the SP required. Therefore, the designers decided that a special-purpose interconnect was necessary [1, 9].

The IBM RS/6000 SP switch [9] is an any-to-any packet-switched multistage network. The bisection bandwidth of the switch scales linearly with the size (number of nodes) of the system. The available bandwidth between any pair of communicating nodes remains constant irrespective of where in the topology the two nodes lie. These features supported both system scalability and ease of use. The system could be viewed as a flat collection of nodes, which could be freely selected for parallel jobs irrespective to their location. Selection could focus on other features, such as processor speed and memory size. As a consequence, the IBM RS/6000 SP did not suffer from the fragmentation problem that was observed in other parallel systems of the time [4].

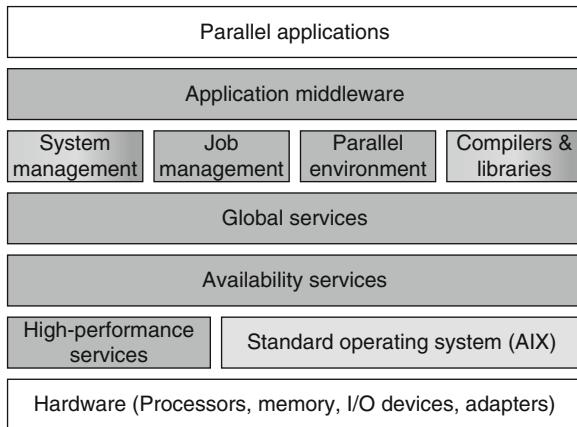
The SP switch is built from a basic  $4 \times 4$  bidirectional crossbar switching elements, which are grouped eight to a board to form a  $16 \times 16$  switch board, as shown in Fig. 3. A switch board connects to nodes on one side and to other switch boards on the other side. Systems with up to 80 nodes can be assembled with just one layer of switch boards, whereas larger systems require additional layers.



**IBM RS/6000 SP. Fig. 3** A  $16 \times 16$  switch board is built by interconnecting eight  $4 \times 4$  bidirectional crossbar switching elements. The switch board connects to nodes on one side and other boards on the other side

## Software Architecture

Figure 4 illustrates the software stack of the RS/6000 SP. That software stack is built upon off-the-shelf UNIX components and specialized services for parallel



IBM RS/6000 SP. Fig. 4 Software stack for RS/6000 SP

processing. Each node runs a full AIX operating system instance. That operating system is complemented at the bottom layer of the software stack by high-performance services that provide connectivity to the SP switch.

The availability and global services layers implement aspects of a single-system image. They are intended to be the basis for parallel applications and application middleware. The availability services of the IBM SP support heartbeat, membership, notification, and recovery coordination. Heartbeat services implement the monitoring of components to detect failures. Membership services allow processors and processes to be identified as belonging to a group. Notification services allow members of a group to be notified when new members are added or old members are removed from that group. Finally, recovery coordination services provide a mechanism for performing recovery procedures within the group in response to changes in the membership.

The global services of the IBM SP provide global access to resources such as disks, files, and networks. Global access to files is provided by networked file solutions, either with a client-server model (e.g., NFS) or with a parallel file system model (e.g., GPFS). With this approach, processes in every node have access to the same file space. Global network access is implemented through TCP/IP and UDP/IP protocols over the IBM SP switch. Gateway nodes, connected to both the SP switch and an external Ethernet network, allow all nodes to access the Ethernet network. Global access to disks is implemented by virtual shared disk (VSD) functionality. VSD allows a process running on any SP

node to access any disk in the system as if it were locally attached to that node. Another global service is the system data repository (SDR). The SDR contains system-wide information about the nodes, switches, and jobs currently in the system.

The job management system of the IBM SP supports both interactive and batch jobs. Batch jobs are submitted, scheduled, and controlled by LoadLeveler [3]. For interactive jobs, a user can login directly to any node in the SP, since the nodes all run a full version of the AIX operating system.

System management for the IBM SP is built upon components used for management of RS/6000 AIX workstations. It also includes extensions developed specifically for the SP to facilitate performing standard management functions across the many nodes of an SP. The functions supported include system installation, system operation, user management, configuration management, file management, security management, job accounting, problem management, change management, hardware monitoring and control, and print and mail services. The system management functions can be performed via a control workstation that acts as the system console.

The compilers and run-time libraries for the IBM RS/6000 SP are based on the standard software stack for IBM AIX augmented with certain features specific to the SP. Fortran, C and C++ compilers, and run-time libraries for POWER-based workstations and servers can be directly used in the SP, since the nodes of the latter are based on hardware developed for the former. The software stack for the SP also includes message-passing libraries that implement both IBM proprietary models, such as MPL, and standard models such as PVM and MPI [5, 8]. Also available for the IBM RS/6000 SP is an implementation of the High Performance Fortran (HPF) programming language [7].

In addition to middleware like MPI libraries that cater to scientific applications, the RS/6000 SP software stack also includes middleware targeted at enabling parallel commercial applications. The main example is DB2 Parallel Edition (PE) [2], an implementation of the DB2 relational database product that runs in parallel across the nodes of the SP. DB2 PE is a shared-nothing parallel database system, in which the data is partitioned across the nodes. DB2 PE splits SQL queries into multiple operations that are then shipped to the nodes for

execution against their local data. A final stage combines the results from each node into a single result. DB2 PE enables database applications to use the parallelism of the RS/6000 SP without changes to the application itself, since the exploitation of parallelism happens in the database middleware layer.

## Example Applications

Thousands of IBM RS/6000 SP systems were delivered over the product lifetime, ranging in size from as few as two nodes all the way up to 512 nodes and larger. The availability of a full workstation- and server-compatible software stack on the SP nodes allowed it to run off-the-shelf AIX applications with zero porting effort. The availability of standard message passing libraries (such as MPI), High Performance Fortran, and parallel commercial middleware such as DB2 PE also meant that existing parallel applications could be moved to the IBM RS/6000 SP with relative ease. Furthermore, several applications were specifically developed or optimized for the SP.

At a relatively early point in the product lifetime (1995), the SP was already being used in many different areas, including computational chemistry, crash analysis, electronic design analysis, seismic analysis, reservoir modeling, decision support, data analysis, on-line transaction processing, local area network consolidation, and as workgroup servers. In terms of economic sectors, SP systems were being used in manufacturing, distribution, transportation, petroleum, communications, utilities, education, government, finance, insurance, and travel [1].

The IBM RS/6000 SP played an important role in the Accelerated Strategic Computing Initiative by the US Department of Energy. That program was responsible for several of the fastest computers in the world, including two SPs: the ASCI Blue-Pacific and ASCI White machines. ASCI Blue-Pacific was the largest SP in number of nodes. It consisted of 1,464 nodes, each with four PowerPC 604e processors. Each processor had a clock speed of 332 MHz and a peak floating-point performance of 664 Mflops. As indicated in the TOP500 list, the machine had a peak performance (Rpeak) of 3856.5 Gflops and a Linpack performance (Rmax) of 2144 Gflops. It was ranked #2 in the November 1999 and June 2000 lists. ASCI White was the largest SP in number of processors (or cores). It consisted of 512 nodes,

each with 16 POWER3 processors. Each processor had a clock speed of 375 MHz and a peak floating-point performance of 1.5 Gflops. As indicated in the TOP500 list, the machine had a peak performance (Rpeak) of 12,288 Gflops and a Linpack performance (Rmax) of 7,304 Gflops. It was ranked #1 in the November 2000, June 2001 and November 2001 lists.

## Related Entries

- [IBM Power Architecture](#)
- [LINPACK Benchmark](#)
- [MPI \(Message Passing Interface\)](#)
- [TOP500](#)

## Bibliographic Notes and Further Reading

For a thorough discussion of the system architecture of the RS/6000 SP, the reader is referred to [1]. Details of the RS/6000 SP interconnection network can be found in [9]. An overview of the system software for the RS/6000 SP is given in [8] while details for the MPI environment and the job scheduling facilities are described in [5] and [3], respectively. The HPF compiler for the RS/6000 SP is described in [7]. Commercial middleware is covered in [2] and user experience in a scientific computing environment is described in [6]. Finally, additional information on the machine fragmentation problem is available in [4].

## Bibliography

1. Agerwala T, Martin JL, Mirza JH, Sadler DC, Dias DM, Snir M (1995) SP2 system architecture. *IBM Syst J* 34(2):152–184
2. Baru CK, Fecteau G, Goyal A, Hsiao H, Jhingran A, Padmanabhan S, Copeland GP, Wilson WG (1995) DB2 Parallel Edition. *IBM Syst J* 34(2):292–322
3. Dewey S, Banas J (1994) LoadLeveler: a solution for job management in the UNIX environment. *AIXtra*, May/June 1994
4. Feitelson DG, Jette MA (1997) Improved utilization and responsiveness with gang scheduling. In: Feitelson DG, Rudolph L (eds) *Proceedings of job scheduling strategies for parallel processing (JSSPP '97)*. Lecture notes in computer science, vol 1291. Springer, Berlin, pp 238–261
5. Franke H, Wu CE, Riviere M, Patnaik P, Snir M (1995) MPI programming environment for IBM SP1/SP2. In: *Proceedings of the 15th international conference on distributed computing systems (ICDCS '95)*, Vancouver, May 30–June 2, 1995, pp 127–135
6. Gropp WD, Lusk E (1995) Experiences with the IBM SP1. *IBM Syst J* 34(2):249–262

7. Gupta M, Midkiff S, Schonberg E, Seshadri V, Shields D, Wang K-Y, Ching W-M, Ngo T (1995) An HPF compiler for the IBM SP2. In: Proceedings of the 1995 ACM/IEEE conference on supercomputing, San Diego, 1995
8. Snir M, Hochschild P, Frye DD, Gildea KJ (1995) The communication software and parallel environment of the IBM SP2. IBM Syst J 34(2):205–221
9. Stunkel CB, Shea DG, Abali B, Atkins MG, Bender CA, Grice DG, Hochschild P, Joseph DJ, Nathanson BJ, Swetz RA, Stucke RF, Tsao M, Varker PR (1995) The SP2 high-performance switch. IBM Syst J 34(2):185–204

## IBM SP

- [IBM RS/6000 SP](#)

## IBM SP1

- [IBM RS/6000 SP](#)

## IBM SP2

- [IBM RS/6000 SP](#)

## IBM SP3

- [IBM RS/6000 SP](#)

## IBM System/360 Model 91

MICHAEL FLYNN  
Stanford University, Stanford, CA, USA

### Definition

The Model 91 was the highest performing computer system introduced by IBM as part of its System/360 series in the 1960s. It was distinguished by a number of innovations such as out-of-order instruction execution, Tomasulo's algorithm for data flow execution, a limited

form of branch speculation, pipelined execution units, and division by binomial series approximation.

## Discussion

### Introduction

When IBM introduced System/360 in 1964, it included an announcement of a high-end computer referred to as "Model 90." This computer was realized as the Model 91 with first installation in 1967. While the Model 91 had a core memory with 750 ns cycle time, there was a fast memory version which used thin-film magnetic storage with a cycle time of 250 ns. This version was labeled the Model 95; two Model 95s were produced. Orders for the Model 91 were closed in 1968, but a revised version with improved technology and cache was introduced around 1970 as the Model 195. The Model 195 used the same logic as the Model 91. In the mid 1960s, there were some references to a Model 92 (with 500 ns memory cycle). This version was never realized.

The Model 90 series used a hybrid technology called ASLT (advanced solid logic technology), and had a processor cycle of 60 ns for Models 91 and 95; the Model 195 had a processor cycle of 54 ns.

The total production of all computers in the Model 90 series was about two dozen.

### Overview of the System and CPU

#### Instruction Processing

The instruction set was exactly that of System/360; however, instruction execution of the decimal instruction set was not supported in hardware in the Model 91. If such an instruction was invoked, it would trap and be interpreted by the processor [1, 2].

The design was predicated on the requirements of a long (20 stage) pipeline. Instructions were prefetched into an 8 entry I buffer each entry having 8 bytes. The I buffer enabled limited speculation on branch outcomes. If the branch target was backward and in the buffer, the branch was predicted to be successful otherwise the branch was predicted to be untaken and proceed in line.

A significant feature of the execution units was a mechanism called the common data bus, developed by R Tomasulo [3] and better known as Tomasulo's algorithm. This is a data flow control algorithm, which renames registers into reservation stations.

Each register in the central register set is extended to include a tag that identifies the functional unit that produces a result to be placed in a particular register. Similarly, each of the multiple functional units has one or more reservation stations. The reservation station, however, can contain either a tag identifying another functional unit or register, *or it can contain the variable needed*. Each reservation station effectively defines its own functional unit; thus, two reservations for a floating point multiplier are two functional unit tags: multiplier 1 and multiplier 2. If operands can go directly into the multiplier, then there is another tag: multiplier 3. Once a pair of operands has a designated functional unit tag, that tag remains with that operand pair until completion of the operation. Any unit (or register) that depends on that result has a copy of the functional unit tag and in gates the result that is broadcast on the common data bus. In this dataflow approach, the results to a targeted register may never actually go to that register; in fact, the computation based on the load of a particular register may be continually forwarded to various functional units, so that before the value is stored, a new value based upon a new computational sequence (a new load instruction) is able to use the targeted register.

The renaming of registers enabled out of order instruction execution, but in doing so it did not allow precise interrupts for all cases of exceptions. Since one instruction may have completed execution before a slow executing earlier issued one (such as divide) takes an exception and causes an interrupt it is impossible to reconstruct the machine state precisely.

### Execution Functional Units

The floating point units also provided significant innovation. The floating point adder executed an instruction in two cycles but it accepted a new instruction each cycle. This functional unit pipelining was novel at this time but now widely used [4].

The floating point multiplier executed an instruction in three cycles. It used a Booth 2 encoding of the multiplier, requiring the addition of 28 signed multiples of the multiplicand (the mantissa had 56 bits). These were added in Sum + Carry form, six at a time using a tree of carry save adders (CSA). The partial product was fed back into the tree for assimilation with the next six multiples. Using a newly developed Earle latch, this assimilation iteration eliminated latching overhead and

required only four logic stages (two CSAs) to implement. This allowed six signed multiples to be assimilated every 20 ns. Thus, the assimilation process took 120 ns to form the produce; another 60 ns was required for startup.

The divide process introduced the Goldschmidt algorithm based on a binomial expansion of the reciprocal. The value of the reciprocal was then multiplied by the dividend to form the quotient. In binomial expansion the divisor,  $d = 1 - x$ ; now the expansion of  $1/(1-x)$  can be represented as  $(1-x)(1+x^2)(1+x^4)(1+x^8)\dots$  Since  $d$  is binary normalized,  $x$  is less than or equal  $\frac{1}{2}$ . So each term doubles the number of zeros following the 1 and the resulting product quadratically converges to the reciprocal. A initial table lookup reduces the number of multiplies, the Model 91 took 11 cycles for floating point divide.

### Memory

The Model 91 memory system used conventional (for the day) magnetic core technology. It was interleaved 32 ways and had a 0.75 microsecond cycle time with a 10 cycle access time. A similar memory technology supported the Model 195's cache based memory. The memory system buffered 32 outstanding requests [5].

### The Technology

The machine itself was implemented with ECL (emitter coupled logic) as the basic circuit technology using multi transistor chips mounted on a  $1 \times 1$  cm aluminum ceramic substrate. Passive devices were implemented as thin-film printed components on the substrate. Two substrates were stacked one on top of each other forming a module which formed a cube about 1 cm on a side. The module provided a circuit density of about two to three circuits. Approximately 20 modules could be mounted on a daughterboard and twenty daughter boards could be plugged in to a motherboard ( $20 \times 20$  cm). Twenty motherboards formed a frame about  $2 \times 2$  m  $\times$  20 cm and four frames formed the basic CPU for the system [6–8].

The ECL circuit delay was about 1.8 ns but the transit time between logic gates added another 1.5 ns. All interconnections were made by terminated transmission line except for a small number of stubbed transmission lines. A great deal of care was paid in developing the signal transmission system. For example: a dual

impedance system of 50 and 90 ohms was used and the width of the basic 50 ohm line was reduced to create a 90 ohm line in the vicinity of loads so that the effective impedance of the loaded 90 ohm line would appear as a 50 ohm line.

The processor had in total about 120,000 gates. Since each gate had an associated interconnection delay the transit time plus loading effects made the total delay per gate approximately 3.5 ns. With 12 stages of logic as the definition for cycle time, this defined 60 ns as the basic CPU cycle. The multiply-divide unit had a subcycle of 20 ns for a partial product iteration.

The processor used water-cooled heat exchangers between motherboards for cooling. A motor generator set powered the system, isolating the system from short power disruptions. The total power consumption was a significant fraction of a megawatt.

## Bibliographic Notes and Further Reading

The basic source material for the Model 91 is the IBM J of Research and Development cited below [1–8]. The term “Tomasulo’s algorithm” and some similar designations are introduced in [9]. Thirty years after the Model 91 was initially dedicated there was a retrospective paper on its accomplishments and problems [10].

## Bibliography

The following eight papers are all from the special issue of the IBM Journal of Research and Development devoted to The IBM System/360 Model 9; vol II, issue 1, January 1967

1. Flynn MJ, Low PR Some remarks on system development, pp 2–7
2. Anderson DW, Sparacio FJ, Tomasulo RM Machine philosophy and instruction handling, pp 8–24
3. Tomasulo RM An efficient algorithm for exploiting multiple arithmetic units, pp 25–33
4. Anderson SF, Goldschmidt RE, Earle JG, Powers DM, Floating-point execution unit, pp 34–53
5. Boland LJ, Messina BU, Granito GD, Smith JW, Marcotte AU Storage system, pp 54–68
6. Langdon JL, Van Derveer EJ Design of a high-speed transistor for the ASLT current switch, pp 69–73
7. Sechler RF, Strube AR, Turnbull JR ASLT circuit design, pp 74–85
8. Lloyd RF ASLT: an extension of hybrid miniaturization techniques, pp 86–92

Other papers mentioning the Model 91 or related computers

9. Flynn MJ (1966) Very high speed computers. Proc IEEE 54: 1901–1909
10. Flynn MJ (1998) Computer engineering 30 years after the IBM model 91. IEEE Comput 31(4):27–31

## IEEE 802.3

- [Ethernet](#)

## Illegal Memory Access

- [Intel® Parallel Inspector](#)

## Illiac IV

YOICHI MURAOKA  
Waseda University, Tokyo, Japan

### History

The Illiac IV computer was the first practical large-scale array computer, which can be classified as an SIMD (single-instruction-stream-multiple-data-streams)-type computer. As the name suggests, the project was managed at the University of Illinois Digital Computer Laboratory under the contract from the Defense Advanced Research Project Agency (then called ARPA). The project started in 1965, and the machine was delivered to the NASA Ames Research Center in 1971. It took 5 years to run its first successful application and was made available via the ARPANET, the predecessor of the Internet. The principal investigator was Professor Daniel Slotnick, who conceived the idea in the mid-1960s as the Solomon computer. The machine was built to answer the large computational requirements such as ballistic missile defense analysis, climate modeling, and so on. It was said then that two Illiac-IV computers would suffice to cover all computational requirements in the planet.

The design of the computer and the majority of early software suits development were done by Illinois researchers, including many ambitious graduate students, while the computer was built by the Burroughs Corporation.

To be precise, the project was transferred from the University of Illinois to the Ames Research Center in 1970 before its completion due to the heavy anti-Vietnam war protest activity on the campus.

The hardware is now being displayed in the Computer History Museum at Silicon Valley.

## Hardware

An instruction is decoded by Control Unit (CU), and the decoded signals are sent to 64 processors, called PE. PE is basically a combination of arithmetic-logic-unit, registers, and 2,048-words memory of 64-bit length (So in total, 1 MB). It operates at a 13 MHz clock. Each PE loads data to a register from its own memory. Thus, the 64 PEs perform an identical operation over independent data simultaneously. For example, two 64 elements vectors can be added by one operation.

The operation of each PE can be controlled by a mode bit. If a control bit is set, then a PE participates the operation, otherwise it does not. Thus, PE has some degree of freedom. There is an instruction to set/reset the mode bit.

A memory address to access the memory in each PE is also provided from CU. Included in each PE is an index register, so one can modify and access to a different memory address in different PEs.

To exchange data between PEs, PEs are set in the shape of a list with wrapped around connection. All PEs move data, for example, to their left neighbor PEs simultaneously. The mechanism is called the routing.

The megabyte memory is far less from ideal. So it is backed up by a 16 million word head-per-track desk with an I/O rate of 500 MB/s. The average access time is 20 ms.

To manage the operation of the computer and to provide the I/O capability, a Burroughs B6700 machine is used. The majority of operating system functions are run on this machine.

To further speed up the instruction execution, the instruction decoding in CU is overlapped with the execution in PEs, i.e., the instruction decoding and the instruction execution is pipelined.

## Software

Besides an ordinary assembler, called ASK, originally, two compilers were planned, the TRANQUIL and the GLYPNIR.

The TRANQUIL language is an extension of the ALGOL. Its main feature is the capability to specify the parallel execution of a FOR loop. For example,

FOR SIM I = (1...64) A(I) = B(I) + C(I)

adds two 64 elements vectors in one step. Furthermore, it allows a user to specify how to store data in the memory. For example, suppose we have an array of 64 columns by 64 rows. If we store each entire column in separate PE memory, then while operation on rows (e.g., addition of corresponding elements of two rows) can be done in parallel, operation on columns must be done sequentially. To avoid this situation, we may store data as

| PE1     | PE2     | PE3    | PE64    |
|---------|---------|--------|---------|
| a(1,1)  | a(1,2)  | a(1,3) | a(1,64) |
| a(2,64) | a(2,1)  | a(2,2) | a(2,63) |
| a(3,63) | a(3,64) | a(3,1) | a(3,62) |

While the simple data mapping scheme is called straight storage, this scheme is called skewed storage. TRANQUIL provides further varieties of storage scheme for an array to allow efficient parallel computation.

While the TRANQUIL language aims at a high-level programming gear, the GLYPNIR language, so to speak, provides a low-level view of the computer. In GLYPNIR, one writes a program for a PE in ALGOL like statements. The parallel execution is implicit, i.e., basically the identical program is executed in all PEs. The special features are added to take advantage of the mode-bit control and the separate memory indexing in each PE.

For example, to add two vectors, we declare variables as

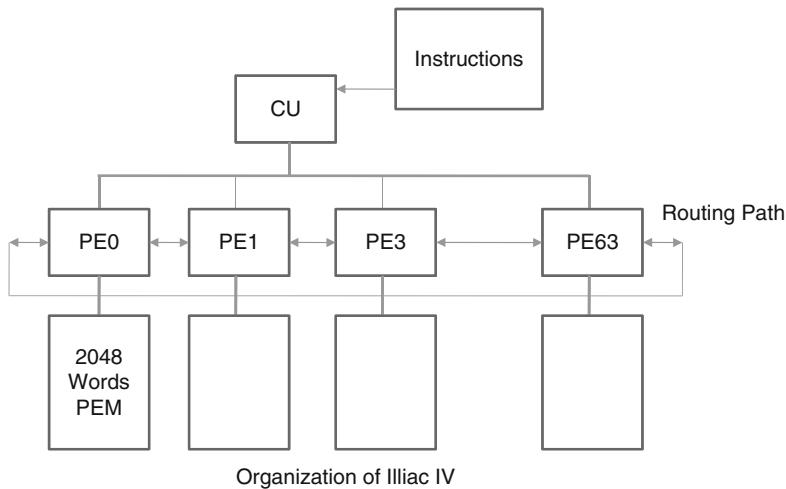
PE REAL X,Y

by which two 64 elements vectors are created, and an element X(i) is stored in PEM of PEi. Thus, the statement  $X \leftarrow X + Y$  will add two vectors in parallel.

To control the execution in each PE, there is a control statement such as

IF < BE > THEN S1

where BE is a 64 elements Boolean values, each of which corresponds the mode bit of PEs. Thus, the statement



**Illiad IV. Fig.1** Organization of Illiac IV

S1 is executed in PEs whose corresponding element of BE is true.

## Applications

The Illiac IV was the first practical parallel computer to allow writing real parallel codes.

Many original and great parallel application ideas have been developed by the project which contributed to solve real world problems. Among many influential results, we just list a few of them below. Application areas attacked by the project are:

### 1. Computational fluid dynamics

NASA people developed an aerodynamic flow simulation program on Illiac IV which helped them to replace their wind tunnel with a computer simulation system. Other programs developed include the Navier-Stokes solver for two dimensional unsteady transonic flow, a viscous-flow airfoil code, turbulence modeling for three-dimensional incompressible flow, and many more.

### 2. Image processing

Image processing may be one of the application areas that is most suited for parallel computation. Many innovative parallel algorithms were developed and implemented on the Illiac IV which include mage line detection, image skeletonizing,

shape detection, and many others. Also, the Illiac IV was successfully used to analyze LANDSAT satellite data, especially in clustering and classification.

### 3. Astronomy

Although very little research was done in this area three-dimensional galaxy simulations have been run successfully on the Illiac IV. This is a typical n-body problem.

### 4. Seismic

This is one of applications which were studied intensively to develop parallel algorithms. The application is characterized as a three-dimensional finite difference code with many irregular data structure.

### 5. Mathematical programs

Many basic numerical computations have been coded in parallel. They include dense and sparse matrix calculations, the single-value decomposition, and so on.

### 6. Weather/climate simulation

A dynamic atmospheric model incorporating chemistry and heat exchange was build.

## Assessment

The project took a decade of development and was also massively over budget. Costs escalated from the

\$8 million estimated in 1966 to \$31 million by 1972. Only a quarter of the fully planned machine, i.e., 64 PEs instead of 256 PEs, was ever built. Nevertheless, the project developed a very unique and for the time, a powerful computer. Also, many ideas in software are still vital and appreciated.

The project pushed research forward, leading the way for machines such as the Thinking Machines CM-1 and CM-2. It is also true that the software team of the project developed not only a whole new set of parallel programming ideas, but also a set of experts in parallel programming.

## Bibliography

1. Barnes GH et al (1963) The Illiac IV computer. IEEE Trans Comput C-17(8):746–757. The paper summarizes hardware and software of the Illiac IV
2. Abel N et al. TRABQUIL, a language for an array processing computer. In: Proceedings AFIPS 1969 SJCC, vol 34. AFIPS Press, Montvale NJ, pp 57–73. The TRANQUIL language is introduced
3. Kuck D, Sameh A (1972) Parallel computation of eigenvalues of real matrices, In: Information Processing 71, vol II. North-Holland, Amsterdam, pp 1266–1272. Describes parallel eigenvalue algorithm
4. Hord RM (1982) The Illiac IV. Computer Science Press, Rockville. A complete description of the project. Out of print now

## ILUPACK

MATTHIAS BOLLHÖFER<sup>1</sup>, JOSÉ I. ALIAGA<sup>2</sup>, ALBERTO F. MARTÍN<sup>1</sup>, ENRIQUE S. QUINTANA-ORTÍ<sup>1</sup>

<sup>1</sup>Universitat Jaume I, Castellón, Spain

<sup>2</sup>TU Braunschweig Institute of Computational Mathematics, Braunschweig, Germany

## Definition

ILUPACK is the abbreviation for Incomplete LU factorization PACKage. It is a software library for the iterative solution of large sparse linear systems. It is written in FORTRAN 77 and C and available at <http://ilupack.tu-bs.de>. The package implements a multilevel incomplete factorization approach (multilevel ILU) based on a special permutation strategy called “inverse-based pivoting” combined with Krylov subspace iteration methods. Its main use consists of application problems such

as linear systems arising from partial differential equations (PDEs). ILUPACK supports single and double precision arithmetic for real and complex numbers. Among the structured matrix classes that are supported by individual drivers are symmetric and/or Hermitian matrices that may or may not be positive definite and general square matrices. An interface to MATLAB (via MEX) is available. The main drivers can be called from C, C++, and FORTRAN.

## Discussion

### Introduction

Large sparse linear systems arise in many application areas such as partial differential equations, quantum physics, or problems from circuit and device simulation. They all share the same central task that consists of efficiently solving large sparse systems of equations. For a large class of application problems, sparse direct solvers have proven to be extremely efficient. However, the enormous size of the underlying applications arising in 3-D PDEs or the large number of devices in integrated circuits currently requires fast and efficient iterative solution techniques, and this need will be exacerbated as the dimension of these systems increases. This in turn demands for alternative approaches and, often, approximate factorization techniques, combined with iterative methods based on Krylov subspaces, reflecting as an attractive alternative for these kinds of application problems. A comprehensive overview over iterative methods can be found in [15].

The ILUPACK software is mainly built on incomplete factorization methods (ILUs) applied to the system matrix in conjunction with Krylov subspace methods. The ILUPACK hallmark is the so-called *inverse-based approach*. It was initially developed to connect the ILUs and their approximate inverse factors. These relations are important since, in order to solve linear systems, the inverse triangular factors resulting from the factorization are applied rather than the original incomplete factors themselves. Thus, information extracted from the inverse factors will in turn help to improve the robustness for the incomplete factorization process. While this idea has been successfully used to improve robustness, its downside was initially that the norm of the inverse

factors could become large such that small entries could hardly be dropped during Gaussian elimination. To overcome this shortcoming, a multilevel strategy was developed to limit the growth of the inverse factors. This has led to the inverse-based approach and hence the incomplete factorization process that has eventually been implemented in ILUPACK benefits from the information of bounded inverse factors while being efficient at the same time [6].

A parallel version of ILUPACK on shared-memory multiprocessors, including current multicore architectures, is under development and expected to be released in the near future. The ongoing development of the parallel code is inspired by a nested dissection hierarchy of the initial system that allows to map tasks concurrently to independent threads within each level.

### The Multilevel Method

To solve a linear system  $Ax = b$ , the multilevel approach of ILUPACK performs the following steps:

1. The given system  $A$  is scaled by diagonal matrices  $D_l$  and  $D_r$  and reordered by permutation matrices  $\Pi_l, \Pi_r$  as

$$A \rightarrow D_l A D_r \rightarrow \Pi_l^T D_l A D_r \Pi_r = \hat{A}.$$

These operations can be considered as a preprocessing prior to the numerical factorization. They typically include scaling strategies to equilibrate the system, scaling and permuting based on maximum weight matchings [7], and, finally, fill-reducing orderings such as nested dissection [14], (approximate) minimum degree [4, 11], and some more.

2. An incomplete factorization  $A \approx LDU$  is next computed for the system  $\hat{A}$ , where  $L, U^T$  are unit lower triangular factors and  $D$  is diagonal. Since the main objective of ILUPACK (*inverse-based approach*) is to limit the norm of the inverse triangular factors,  $L^{-1}$  and  $U^{-1}$ , the approximate factorization process is interlaced with a pivoting strategy that cheaply estimates the norm of these inverse factors. The pivoting process decides in each step to reject a factorization step if an imposed bound  $\kappa$  is exceeded, or to accept a pivot and continue the approximate factorization otherwise. The set of rejected rows and columns is permuted to the lower and right end of the matrix. This process is illustrated in Fig. 1.

The pivoting strategy computes a partial factorization

$$\begin{aligned} P^T \hat{A} P &= \begin{pmatrix} B & E \\ F & C \end{pmatrix} \\ &= \begin{pmatrix} L_B & 0 \\ L_F & I \end{pmatrix} \begin{pmatrix} D_B U_B & D_B U_E \\ 0 & S_C \end{pmatrix} + R, \end{aligned}$$

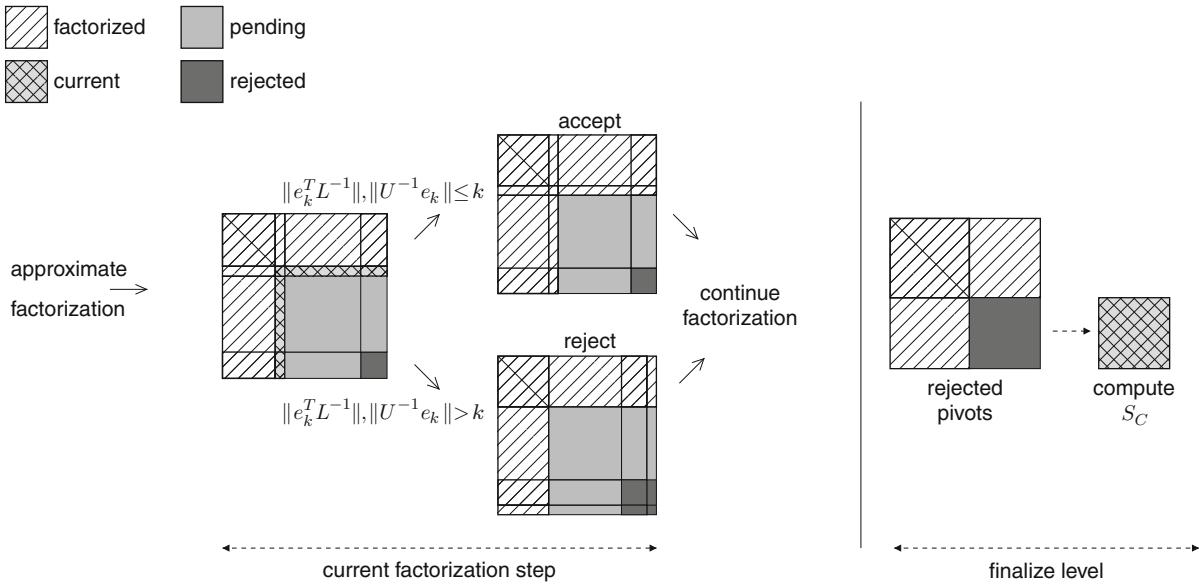
where  $R$  is the error matrix, which collects those entries of  $\hat{A}$  that were dropped during the factorization and “ $S_C$ ” is the Schur complement consisting of all rows and columns associated with the rejected pivots. By construction the inverse triangular factors satisfy

$$\left\| \begin{pmatrix} L_B & 0 \\ L_F & I \end{pmatrix}^{-1} \right\|, \left\| \begin{pmatrix} U_B & U_E \\ 0 & I \end{pmatrix}^{-1} \right\| \lesssim \kappa.$$

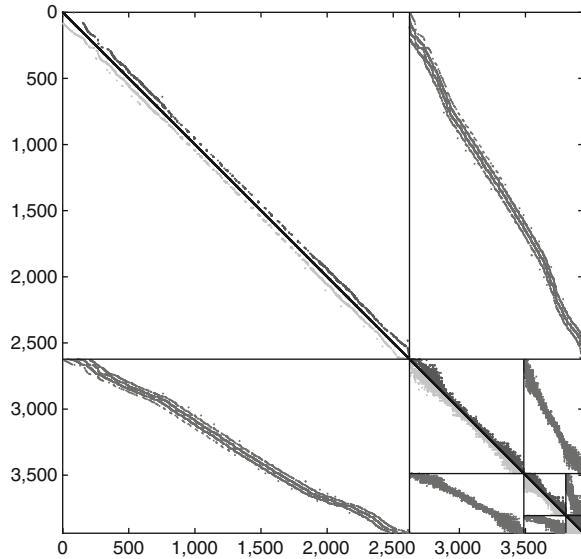
3. Steps 1 and 2 are successively applied to  $\hat{A} = S_C$  until  $S_C$  is void or “sufficiently dense” to be efficiently factorized by a level 3 BLAS-based direct factorization kernel.

When the multilevel method is applied over multiple levels, a cascade of factors  $L_B, D_B$ , and  $U_B$ , as well as matrices  $E, F$  are usually obtained (cf. Fig. 2). Solving linear systems via a multilevel ILU requires a hierarchy of forward and backward substitutions, interlaced with reordering and scaling stages. ILUPACK employs Krylov subspace methods to incorporate the approximate factorization into an iterative method.

The computed multilevel factorization is adapted to the structure of the underlying system. For real symmetric positive definite (SPD) matrices, an incomplete Cholesky decomposition is used in conjunction with the conjugate gradient method. For the symmetric and indefinite case, symmetric maximum weight matchings [8] allow to build  $1 \times 1$  and  $2 \times 2$  pivots. In this case ILUPACK constructs a blocked symmetric multilevel ILU and relies on the simplified QMR [10] as iterative solver. The general (unsymmetric) case typically uses GMRES [16] as default driver. All drivers in ILUPACK support real/complex and single/double precision arithmetic.



ILUPACK. Fig. 1 ILUPACK pivoting strategy



ILUPACK. Fig. 2 ILUPACK multilevel factorization

### Mathematical Background

The motivation for inverse-based approach of ILUPACK can be explained in two ways. First, when the approximate factorization

$$A = LDU + R$$

is computed for some error matrix  $R$ , the inverse triangular factors  $L^{-1}$  and  $U^{-1}$  have to be applied to solve a

linear system  $Ax = b$ . From this point of view,

$$L^{-1}AU^{-1} = D + L^{-1}RU^{-1}$$

ensures that the entries in the error matrix  $R$  are not amplified by some large inverse factors  $L^{-1}$  and  $U^{-1}$ . The second and more important aspect links ILUPACK's multilevel ILU to algebraic multilevel methods. The inverse  $\hat{A}^{-1}$  can be approximately written as

$$\begin{aligned} \hat{A}^{-1} \approx P &\left( \begin{pmatrix} (L_B D_B U_B)^{-1} & 0 \\ 0 & 0 \end{pmatrix} \right. \\ &\left. + \begin{pmatrix} -U_B^{-1} U_E \\ I \end{pmatrix} S_C^{-1} \begin{pmatrix} -L_F L_B^{-1} & I \end{pmatrix} \right) P^T. \end{aligned}$$

In this expression, all terms on the right hand side are constructed to be bounded except  $S_C^{-1}$ . Since in general  $\hat{A}^{-1}$  has a relatively large norm, so does  $S_C^{-1}$ . In principle this justifies that eigenvalues of small modulus are revealed by the approximate Schur complement  $S_C$  (a precise explanation is given in [5]).  $S_C$  serves as some kind of coarse grid system in the sense of discretized partial differential equations. This observation goes hand in hand with the observation that the inverse-based pivoting approach selects pivots similar to coarsening strategies in algebraic multilevel methods,

which is demonstrated for the following simple model problem:

$$\begin{aligned} -10^{-2}u_{xx}(x,y) - u_{yy}(x,y) &= f(x,y) \text{ for all} \\ (x,y) \in [0,1]^2, u(x,y) &= 0 \text{ on } \partial[0,1]^2. \end{aligned}$$

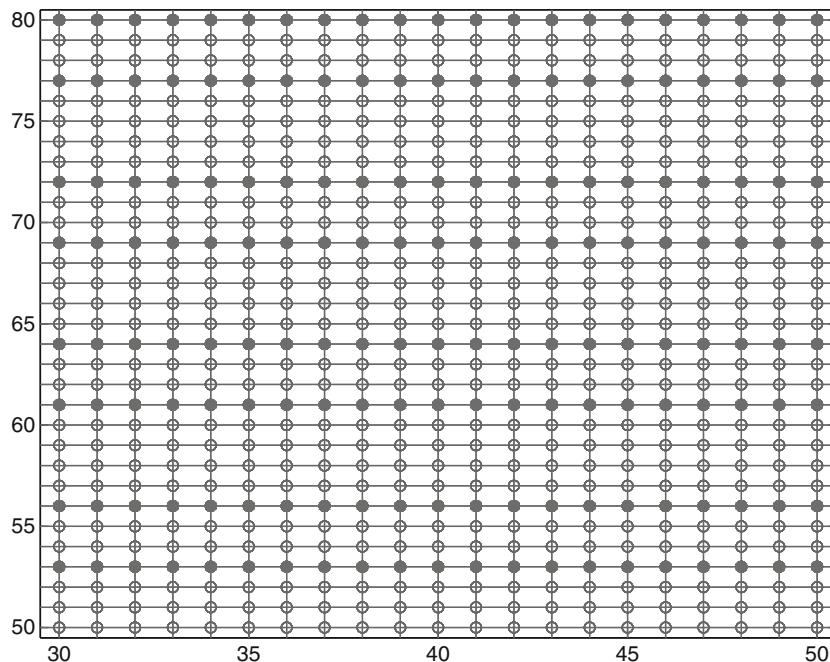
This partial differential equation can be easily discretized on a square grid  $\Omega_h = \{(kh, lh) : k, l = 0, \dots, N+1\}$ , where  $h = \frac{1}{N+1}$  is the mesh size, using standard finite difference techniques. Algebraic approaches to multilevel methods roughly treat the system as if the term  $-10^{-2}u_{xx}(x,y)$  is hardly present, i.e., the coarsening process treats it as if it were a sequence of one-dimensional differential equations in the  $y$ -direction. Thus, semi-coarsening in the  $y$ -direction is the usual approach to build a coarse grid. ILUPACK inverse-based pivoting algebraically picks and rejects the pivots precisely in the same way as in semi-coarsening. In Fig. 3, this is illustrated for a portion of the grid  $\Omega_h$  using  $\kappa = 3$ . In the  $y$ -direction, pivots are rejected after 3–5 steps while in the  $x$ -direction all pivots are kept or rejected (in blue and red, resp.).

The number of rejected pivots in ILUPACK strongly depends on the underlying application problem.

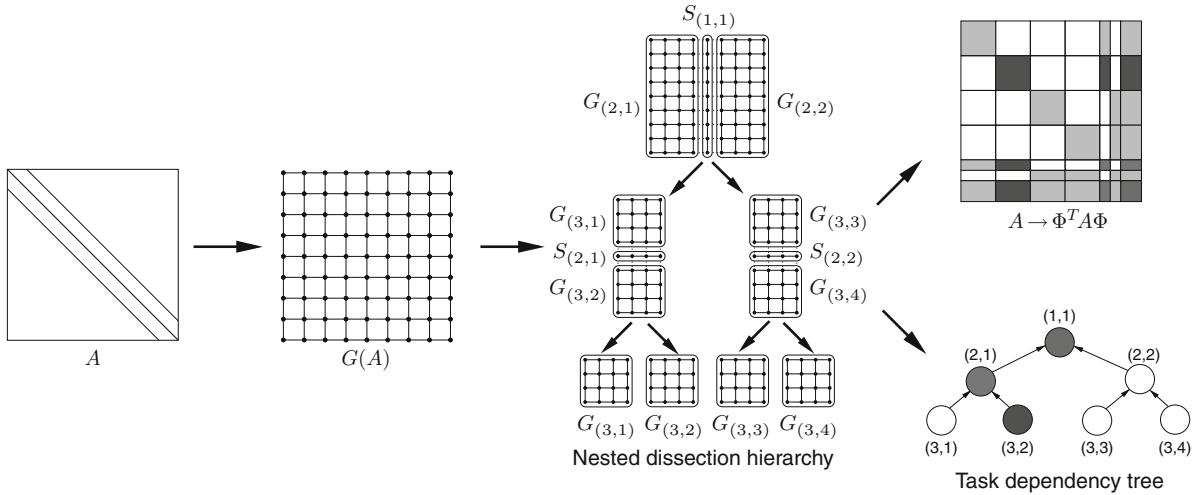
As a rule of thumb, the more rows (and columns) of the coefficient matrix satisfy  $|a_{ii}| \gg \sum_{j \neq i} |a_{ij}|$ , the less pivots tend to be rejected.

### The Parallelization Approach

Parallelism in the computation of approximate factorizations can be exposed by means of graph-based symmetric reordering algorithms, such as graph coloring or graph partitioning techniques. Among these classes of algorithms, nested dissection orderings enhance parallelism in the approximate factorization of  $A$  by partitioning its associated adjacency graph  $G(A)$  into a hierarchy of vertex separators and independent subgraphs. For example, in Fig. 4,  $G(A)$  is partitioned after two levels of recursion into four independent subgraphs,  $G_{(3,1)}$ ,  $G_{(3,2)}$ ,  $G_{(3,3)}$ , and  $G_{(3,4)}$ , first by separator  $S_{(1,1)}$ , and then by separators  $S_{(2,1)}$  and  $S_{(2,2)}$ . This hierarchy is constructed so that the size of vertex separators is minimized while simultaneously balancing the size of the independent subgraphs. Therefore, relabeling the nodes of  $G(A)$  according to the levels in the hierarchy leads to a reordered matrix,  $A \rightarrow \Phi^T A \Phi$ , with a structure amenable to efficient parallelization. In particular, the leading diagonal blocks of



**ILUPACK. Fig. 3** ILUPACK pivoting for partial differential equations. Blue and red dots denote, respectively, accepted and rejected pivots



**ILUPACK. Fig. 4** Nested dissection reordering

$\Phi^T A \Phi$  associated with the independent subgraphs can be first eliminated independently; after that,  $S_{(2,1)}$  and  $S_{(2,2)}$  can be eliminated in parallel, and finally separator  $S_{(1,1)}$  is processed. This type of parallelism can be expressed by a binary task dependency tree, where nodes represent concurrent tasks and arcs dependencies among them. State-of-the-art reordering software packages e.g., METIS (<http://glaros.dtc.umn.edu/gkhome/views/metis>) or SCOTCH (<http://www.labri.fr/perso/pelegrin/scotch>), provide fast and efficient multilevel variants [14] of nested dissection orderings.

The dependencies in the task tree are resolved while the computational data and results are generated and passed from the leaves toward the root. The leaves are responsible for approximating the leading diagonal blocks of  $\Phi^T A \Phi$ , while those blocks which will be later factorized by their ancestors are updated. For example, in Fig. 4, colors are used to illustrate the correspondence between the blocks of  $\Phi^T A \Phi$  to be factorized by tasks (3,2), (2,1), and (1,1). Besides, (3,2) only updates those blocks that will be later factorized by tasks (2,1) and (1,1). Taking this into consideration,  $\Phi^T A \Phi$  is decomposed into the sum of several submatrices, one local block per each leaf of the tree, as shown in Fig. 5. Each local submatrix is composed of the blocks to be factorized by the corresponding task, together with its local contributions to the blocks that are later factorized by its ancestors, hereafter referred as contribution blocks. This strategy significantly increases the degree of parallelism,

because the updates from descendant nodes to an ancestor node can also be performed locally/independently by its descendants.

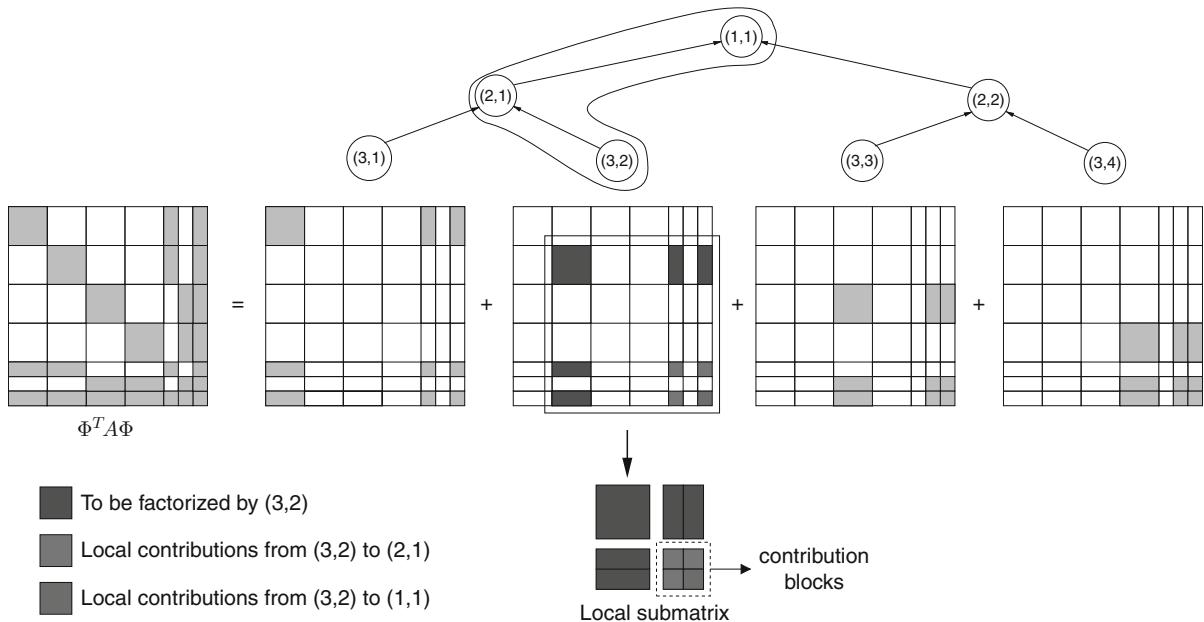
The parallel multilevel method considers the following partition of the local submatrices into  $2 \times 2$  block matrices

$$A_{\text{par}} = \begin{pmatrix} A_X & | & A_V \\ \hline A_W & | & A_Z \end{pmatrix},$$

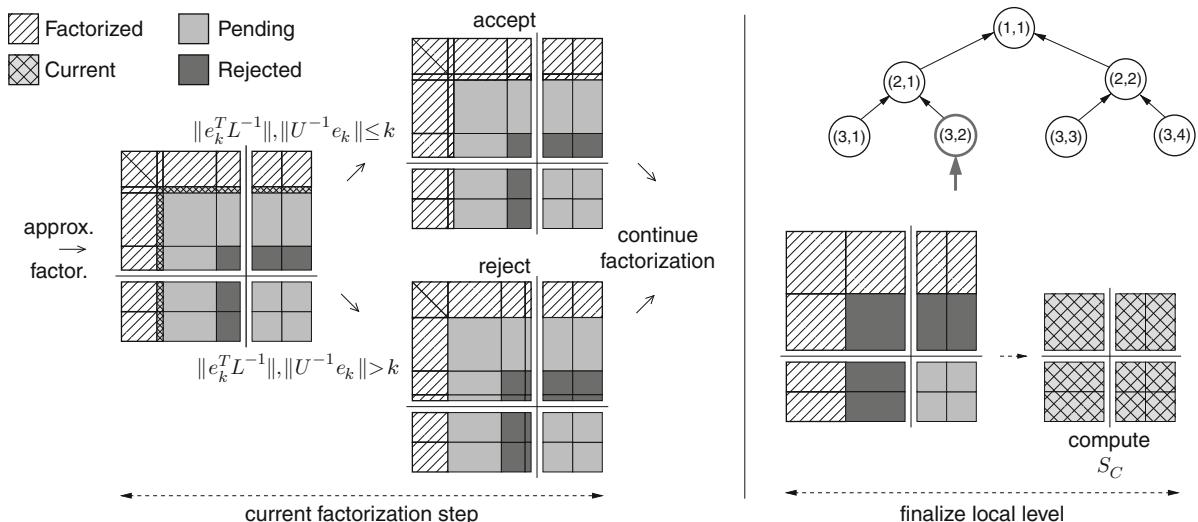
where the partitioning lines separate the blocks to be factorized by the task, i.e.,  $A_X$ ,  $A_W$ , and  $A_V$ , and its contribution blocks, i.e.,  $A_Z$ . It then performs the following steps:

1. Scaling and permutation matrices are only applied to the blocks to be factorized,

$$\begin{aligned} A_{\text{par}} &\rightarrow \left( \begin{array}{c|c} \Pi_l^T & 0 \\ \hline 0 & I \end{array} \right) \left( \begin{array}{c|c} D_l & 0 \\ \hline 0 & I \end{array} \right) \\ &\quad \times \left( \begin{array}{c|c} A_X & A_V \\ \hline A_W & A_Z \end{array} \right) \left( \begin{array}{c|c} D_r & 0 \\ \hline 0 & I \end{array} \right) \\ &\quad \times \left( \begin{array}{c|c} \Pi_r & 0 \\ \hline 0 & I \end{array} \right) = \left( \begin{array}{c|c} \hat{A}_X & \hat{A}_V \\ \hline \hat{A}_W & \hat{A}_Z \end{array} \right) = \hat{A}_{\text{par}}. \end{aligned}$$



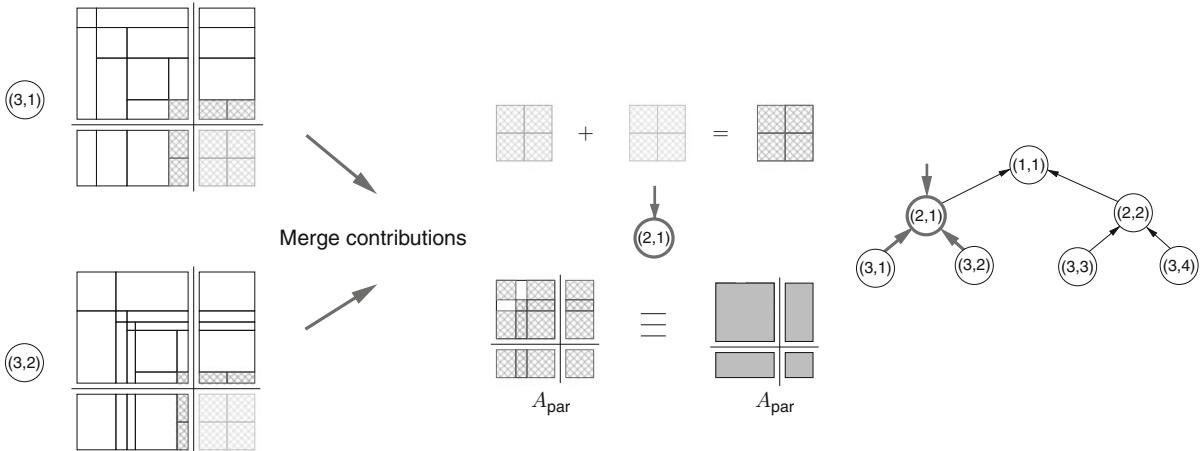
**ILUPACK. Fig. 5** Matrix decomposition and local submatrix associated to a single node of the task tree



**ILUPACK. Fig. 6** Local incomplete factorization computed by a single node of the task tree

2. These blocks are next approximately factorized using inverse-based pivoting, while  $\hat{A}_Z$  is only updated. For this purpose, during the incomplete

factorization of  $\hat{A}_{\text{par}}$ , rejected rows and columns are permuted to the lower and right end of the leading block  $\hat{A}_X$ . This is illustrated in [Fig. 6](#).



ILUPACK. Fig. 7 Parent nodes construct their local submatrix from the data generated by their children

This step computes a partial factorization

$$\begin{aligned} \left( \begin{array}{c|c} P^T & 0 \\ \hline 0 & I \end{array} \right) \hat{A}_{\text{par}} \left( \begin{array}{c|c} P & 0 \\ \hline 0 & I \end{array} \right) &= \left( \begin{array}{ccc|c} B_X & E_X & & E_V \\ F_X & C_X & & C_V \\ \hline F_W & C_W & & C_Z \end{array} \right) \\ &= \left( \begin{array}{cc|c} L_{B_X} & 0 & 0 \\ L_{F_X} & I & 0 \\ \hline L_{F_W} & 0 & I \end{array} \right) \left( \begin{array}{ccc|c} D_{B_X} U_{B_X} & D_{B_X} U_{E_X} & D_{B_X} U_{E_V} & \\ 0 & S_{C_X} & S_{C_V} & \\ \hline 0 & S_{C_W} & S_{C_Z} & \end{array} \right) + R, \end{aligned}$$

where the inverse triangular factors approximately satisfy

$$\left\| \left( \begin{array}{cc|c} L_{B_X} & 0 & 0 \\ L_{F_X} & I & 0 \\ \hline L_{F_W} & 0 & I \end{array} \right)^{-1} \right\|, \quad \left\| \left( \begin{array}{cc|c} U_{B_X} & U_{E_X} & U_{E_V} \\ 0 & I & 0 \\ \hline 0 & 0 & I \end{array} \right)^{-1} \right\| \lesssim \kappa.$$

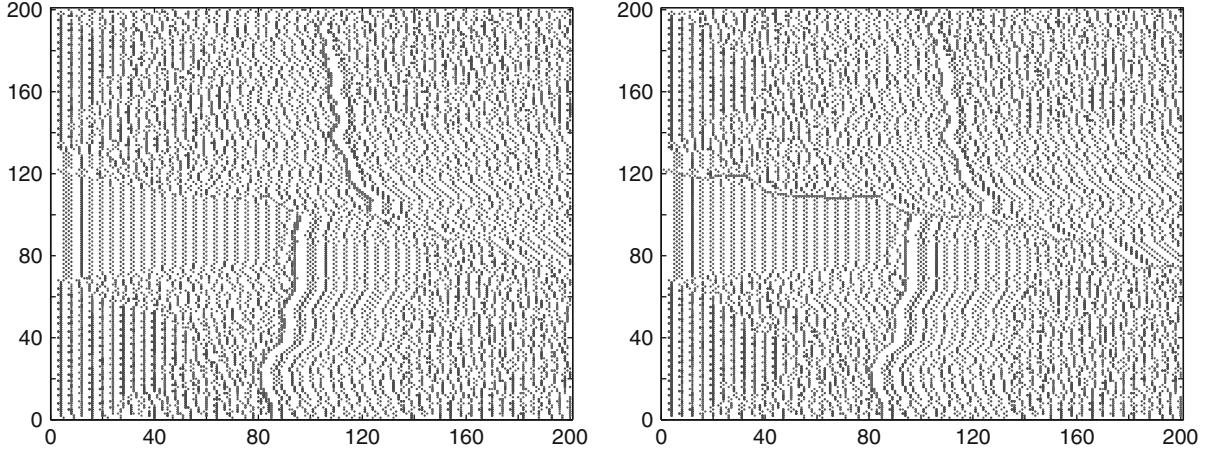
3. Steps 1 and 2 are successively applied to the Schur complement until  $S_{C_X}$  is void or “sufficiently small”, i.e.,  $A_{\text{par}}$  and its  $2 \times 2$  block partition are redefined as

$$A_{\text{par}} = \left( \begin{array}{c|c} A_X & A_V \\ \hline A_W & A_Z \end{array} \right) := \left( \begin{array}{c|c} S_{C_X} & S_{C_V} \\ \hline S_{C_W} & S_{C_Z} \end{array} \right).$$

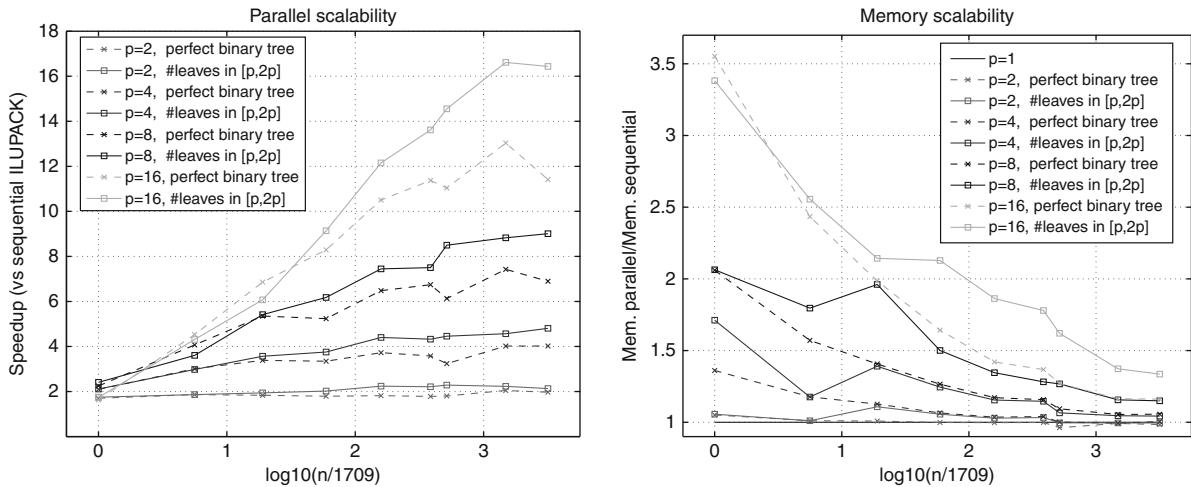
4. The task completes its local computations and the result  $A_{\text{par}}$  is sent to the parent node in the task dependency tree. When a parent node receives the

data from its children, it must first construct its local submatrix  $A_{\text{par}}$ . To achieve this, it incorporates the pivots rejected from its children and accumulates their contribution blocks, as shown in Fig. 7. If the parent node is the root of the task dependency tree, it applies the sequential multilevel algorithm to the new constructed submatrix  $A_{\text{par}}$ . Otherwise, the parallel multilevel method is restarted on this matrix at step 1.

To compare the cascade of approximations computed by the sequential multilevel method and its parallel variant, it is helpful to consider the latter as an algorithmic variant, which enforces a certain order of elimination. Thus, the parallel variant interlaces the algebraic levels generated by the pivoting strategy of ILUPACK with the nested dissection hierarchy levels in order to expose a high degree of parallelism. The leading diagonal blocks associated with the last nested dissection hierarchy level are first factorized using inverse-based pivoting, while those corresponding to previous hierarchy levels are only updated, i.e., the nodes belonging to the separators are rejected by construction. The multilevel algorithm is restarted on the rejected nodes, and only when it has eliminated the bulk of the nodes of the last hierarchy level, it starts approximating the blocks belonging to previous hierarchy levels. Figure 8 compares the distribution of



**ILUPACK. Fig. 8** ILUPACK pivoting strategy (*left*) and its parallel variant (*right*). Blue and red dots denote, respectively, accepted and rejected nodes



**ILUPACK. Fig. 9** Parallel speedup as a function of problem size (*left*) and ratio among the memory consumed by the parallel multilevel method and that of the sequential method (*right*) for a discretized 3-D elliptic PDE

accepted and rejected nodes (in blue and red, resp.) by the sequential and parallel inverse-based incomplete factorizations when they are applied to the Laplace PDE with discontinuous coefficients discretized with a standard  $200 \times 200$  finite-difference grid. In both cases, the grid was reordered using nested dissection. These diagrams confirm the strong similarity between the sequential inverse-based incomplete factorization and its parallel variant for this particular example. The experimentation with this approach in the SPD case reveals that this compromise has a negligible impact on

the numerical properties of the inverse-based preconditioning approach, in the sense that the convergence rate of the preconditioned iterative method is largely independent on the number of processors involved in the parallel computation.

### Numerical Example

The example illustrates the parallel and memory scalability of the parallel multilevel method on a SGI Altix 350 CC-NUMA shared-memory multiprocessor with 16 Intel Itanium2@1.5 GHz processors

sharing 32 GBytes of RAM connected via a SGI NUMA-link network. The nine linear systems considered in this experiment are derived from the linear finite element discretization of the irregular 3-D elliptic PDE  $[-\operatorname{div}(A \operatorname{grad} u) = f]$  in a 3-D domain, where  $A(x, y, z)$  is chosen with positive random coefficients. The size of the systems ranges from  $n = 1,709\text{--}5,413,520$  equations/unknowns.

The parallel execution of the task tree on shared-memory multiprocessors is orchestrated by a runtime which dynamically maps tasks to threads (processors) in order to improve load balance requirements during the computation of the multilevel preconditioner. Figure 9 displays two lines for each number of processors: dashed lines are obtained for a perfect binary tree with the same number of leaves as processors, while solid lines correspond to a binary tree with (potentially) more leaves than processors (up to  $2p$  leaves). The higher speedups revealed in the solid lines demonstrate the benefit of dynamic scheduling on shared-memory multiprocessors.

As shown in Fig. 9 (left), the speedup always increases with the number of processors for a fixed problem size, and the parallel efficiency rapidly grows with problem size for a given number of processors. Besides, as shown in Fig. 9 (right), the memory overhead associated with parallelization relatively decreases with the problem size for a fixed number of processors, and it is below 1.5 for the two largest linear systems; this is very moderate taking into consideration that the amount of available physical memory typically increases linearly with the number of processors. These observations confirm the excellent scalability of the parallelization approach up to 16 cores.

### Future Research Directions

ILUPACK is currently parallelized for SPD matrices only. Further classes of matrices such as symmetric and indefinite matrices or unsymmetric, but symmetrically structured matrices are subject to ongoing research. The parallelization of these cases shares many similarities with the SPD case. However, maximum weight matching, or more generally, methods to rescale and reorder the system to improve the size of the diagonal entries (resp. diagonal blocks) are hard to parallelize [9]. Although the current implementation of ILUPACK is designed for shared-memory

multiprocessors using OpenMP, the basic principle is currently being transferred to distributed-memory parallel architectures via MPI. Future research will be devoted to block-structured algorithms to improve cache performance. Block structures have been proven to be extremely efficient for direct methods, but their integration into incomplete factorizations remains a challenge; on the other hand, recent research indicates the high potential of block-structured algorithms also for ILUs [12, 13].

### Related Entries

- ▶ [Graph Partitioning](#)
- ▶ [Linear Algebra Software](#)
- ▶ [Load Balancing, Distributed Memory](#)
- ▶ [Nonuniform Memory Access \(NUMA\) Machines](#)
- ▶ [Shared-Memory Multiprocessors](#)

### Bibliographic Notes and Further Reading

Around 1985, a first package also called ILUPACK was developed by H.D. Simon [18], which used reorderings, incomplete factorizations, and iterative methods in one package. Nowadays, as the development of preconditioning methods has advanced significantly by novel approaches like improved reorderings, multilevel methods, maximum weight matchings, and inverse-based pivoting, incomplete factorization methods have changed completely and gained wide acceptance among the scientific community. Besides ILUPACK, there exist several software packages based on incomplete factorizations and on multilevel factorizations. For example, Y. Saad (<http://www-users.cs.umn.edu/~saad/software/>) et al. developed the software packages SPARSKIT, ITSOL, and pARMS, which also inspired the development of ILUPACK. In particular, pARMS is a parallel code based on multilevel ILU interlaced with reorderings. J. Mayer (<http://iamlasun8.mathematik.uni-karlsruhe.de/~ae04/iluplusplus.html>) developed ILU++, also for multilevel ILU. MRILU, by F.W. Wubs (<http://www.math.rug.nl/~wubs/mrilu/>) et al., uses multilevel incomplete factorizations and has successfully been applied to problems arising from partial differential equations. There are many other scientific publications on multilevel ILU methods, most of them especially tailored for the use in partial differential equations.

ILUPACK has been successfully applied to several large scale application problems, in particular, to the Anderson model of localization [17] or the Helmholtz equation [5]. Its symmetric version is integrated into the software package JADAMILU (<http://homepages.ulb.ac.be/~jadamilu/>), which is a Jacobi-Davidson-based eigenvalue solver for symmetric eigenvalue problems.

More details on the design aspects of the parallel multilevel preconditioner (including dynamic scheduling) and experimental data with a benchmark of irregular matrices from the UF sparse matrix collection (<http://www.cise.ufl.edu/research/sparse/matrices/>) can be found in [1, 2]. The computed parallel multilevel factorization is incorporated to a Krylov subspace solver, and the underlying parallel structure of the former, expressed by the task dependency tree, is exploited for the application of the preconditioner as well as other major operations in iterative solvers [2]. The experience with these parallel techniques has revealed that the sequential computation of the nested dissection hierarchy (included in, e.g., METIS or SCOTCH) dominates the execution time of the whole solution process when the number of processors is large. Therefore, additional types of parallelism have to be exploited during this stage in order to develop scalable parallel solutions; in [3], two parallel partitioning packages, ParMETIS (<http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview>) and PTSCOTCH (<http://www.labri.fr/perso/pelegrin/scotch>), are evaluated with this purpose.

## Bibliography

- Aliaga JI, Bollhöfer M, Martín AF, Quintana-Ortí ES (2008) Design, tuning and evaluation of parallel multilevel ILU preconditioners. In: Palma J, Amestoy P, Dayde M, Mattoso M, Lopes JC (eds) High performance computing for computational science – VECPAR 2008, Toulouse, France. Number 5336 in Lecture Notes in Computer Science, pp 314–327. Springer, Berlin/Heidelberg
- Aliaga JI, Bollhöfer M, Martín AF, Quintana-Ortí ES (2008) Exploiting thread-level parallelism in the iterative solution of sparse linear systems. Technical report, Dpto. de Ingeniería y Ciencia de Computadores, Universitat Jaume I, Castellón (submitted for publication)
- Aliaga JI, Bollhöfer M, Martín AF, Quintana-Ortí ES (2009) Evaluation of parallel sparse matrix partitioning software for parallel multilevel ILU preconditioning on shared-memory multiprocessors. In: Chapman B et al (eds) Parallel computing: from multicore and GPUs to petascale. Advances in parallel computing, vol 19. IOS Press, Amsterdam, pp 125–132
- Amestoy P, Davis TA, Duff IS (1996) An approximate minimum degree ordering algorithm. *SIAM J Matrix Anal Appl* 17(4): 886–905
- Bollhöfer M, Grote MJ, Schenk O (2009) Algebraic multilevel preconditioner for the Helmholtz equation in heterogeneous media. *SIAM J Sci Comput* 31(5):3781–3805
- Bollhöfer M, Saad Y (2006) Multilevel preconditioners constructed from inverse-based ILUs. *SIAM J Sci Comput* 27(5):1627–1650
- Duff IS, Koster J (1999) The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J Matrix Anal Appl* 20(4):889–901
- Duff IS, Pralet S (2005) Strategies for scaling and pivoting for sparse symmetric indefinite problems. *SIAM J Matrix Anal Appl* 27(2):313–340
- Duff IS, Uçar B (Aug 2009) Combinatorial problems in solving linear systems. Technical Report TR/PA/09/60, CERFACS
- Freund R, Nachtigal N (1995) Software for simplified Lanczos and QMR algorithms. *Appl Numer Math* 19(3):319–341
- George A, Liu JW (1989) The evolution of the minimum degree ordering algorithm. *SIAM Rev* 31(1):1–19
- Gupta A, George T (2010) Adaptive techniques for improving the performance of incomplete factorization preconditioning. *SIAM J Sci Comput* 32(1):84–110
- Hénon P, Ramet P, Roman J (2008) On finding approximate supernodes for an efficient block-ILU(k) factorization. *Parallel Comput* 34(6–8):345–362
- Karypis G, Kumar V (1998) A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J Sci Comput* 20(1):359–392
- Saad Y (2003) Iterative methods for sparse linear systems, 2nd edn. SIAM Publications, Philadelphia, PA
- Saad Y, Schultz M (1986) GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J Sci Stat Comput* 7:856–869
- Schenk O, Bollhöfer M, Römer RA (2008) Awarded SIGEST paper: on large scale diagonalization techniques for the Anderson model of localization. *SIAM Rev* 50:91–112
- Simon HD (Jan 1985) User guide for ILUPACK: incomplete LU factorization and iterative methods. Technical Report ETA-LR-38, Boeing Computer Services

## Impass

### ► Deadlocks

## Implementations of Shared Memory in Software

### ► Software Distributed Shared Memory

## Index

► [All-to-All](#)

## InfiniBand

DHABALESWAR K. PANDA, SAYANTAN SUR  
The Ohio State University, Columbus, OH, USA

### Synonyms

[Interconnection network](#); [Network architecture](#)

### Definition

The InfiniBand Architecture (IBA) describes a switched interconnect technology for inter-processor communication and I/O in a multiprocessor system. The architecture is independent of the host operating system and the processor platform. InfiniBand is based on a widely adopted open standard.

### Discussion

#### Introduction

As the commodity clusters started getting popular around mid-nineties, the common interconnect used for such clusters was Fast Ethernet (100 Mbps). These clusters were identified as Beowulf clusters [1]. Even though it was cost effective to design such clusters with Ethernet/Fast Ethernet, the communication performance was not very good because of the high overhead associated with the standard TCP/IP communication protocol stack. The high overhead was because of the TCP/IP protocol stack being completely executed on the host processor. The network interface cards ((NICs) or commonly known as network adapters) on those systems were also not intelligent. Thus, these adapters did not allow any overlap of computation with communication or I/O. This led to high latency, low bandwidth, and high CPU requirement for communication and I/O operations on those clusters.

The above limitations led to a goal for designing a new and converged interconnect with open standard. Seven industry leaders (Compaq, Dell, IBM, Intel, Microsoft, HP, and Sun) formed a new InfiniBand Trade association [2]. The charter for this association was “to

design a scalable and high performance communication and I/O architecture by taking an integrated view of computing, networking and storage technologies.” Many other companies participated in this consortium later. The members of this consortium deliberated and defined the InfiniBand architecture specification. The first specification (Volume 1, Version 1.0) was released to the public on October 24, 2000. Since then this standard is becoming enhanced periodically. The latest version is 1.2.1 was released in January 2008.

The word “InfiniBand” is coined from two words “Infinite Bandwidth”. The architecture is defined in such a manner that as the speed of computing and networking technologies improve over time, the InfiniBand architecture should be able to deliver higher and higher bandwidth. During the inception in 2001, InfiniBand was delivering a link speed of 2.5 Gbps (payload data rate of 2 Gbps due to the underlying 8/10 encoding and decoding). Now in 2011, it is able to deliver a link speed of 40 Gbps (payload data rate of 32 Gbps).

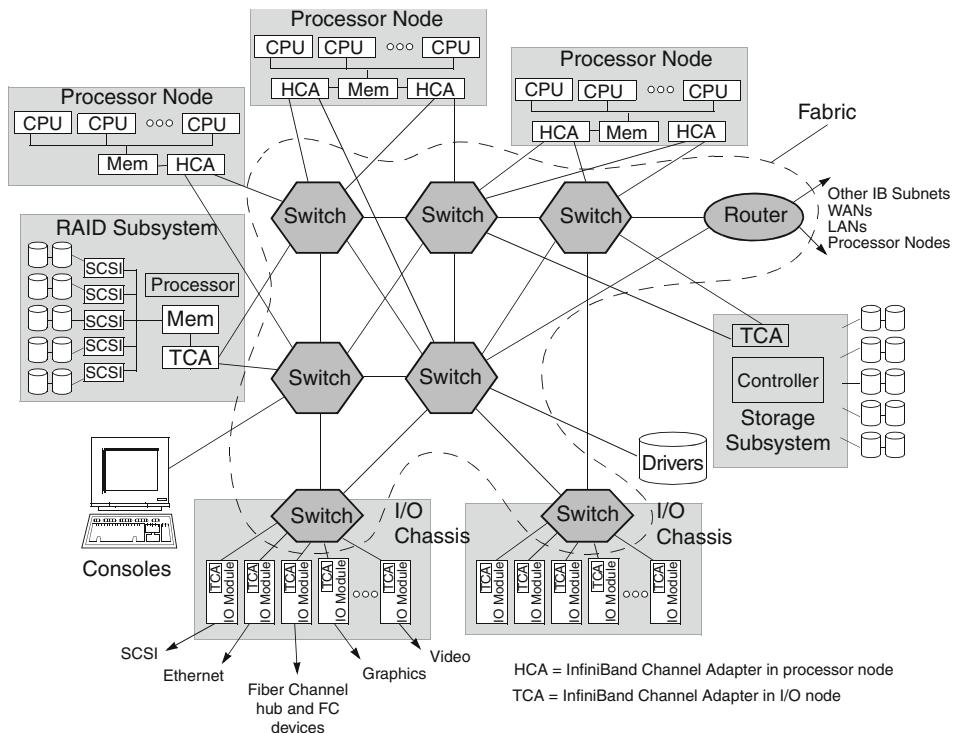
#### Communication Model

The communication model comprises of an interaction of multiple components in the interconnect system. The major components and their interaction are described below.

#### Topology and Network Components

At a high level, IBA serves as an interconnection of nodes, where each node can be a processor node, an I/O unit, a switch or a router to another network, as illustrated in Fig. 1. Processor nodes or I/O units are typically referred to as “end nodes,” while switches and routers are referred to as “intermediate nodes” (or sometimes as “routing elements”). An IB network is subdivided into subnets interconnected by routers. Overall, the IB fabric is comprised of four different components: (a) channel adapters, (b) links and repeaters, (c) switches, and (d) routers.

*Channel adapters:* A processor or I/O node connects to the fabric using channel adapters connecting them to the IB fabric. These channel adapters consume and generate IB packets. Most current channel adapters are equipped with programmable direct memory access (DMA) engines with protection features implemented in hardware. Each adapter can have one or more physical ports connecting it to either a



**InfiniBand. Fig. 1** Typical InfiniBand Cluster (Courtesy InfiniBand Standard)

switch/router or another adapter. Each physical port itself internally maintains two or more virtual channels (or virtual lanes) with independent buffering for each of them. The channel adapters also provide a memory translation and protection (MTP) mechanism that translates virtual addresses to physical addresses and validates access rights. Each channel adapter has a globally unique identifier (GUID) assigned by the channel adapter vendor. Additionally, each port on the channel adapter has a unique port GUID assigned by the vendor as well.

*Links and repeaters:* Links interconnect channel adapters, switches, routers, and repeaters to form an IB network fabric. Different forms of IB links are currently available, including copper links, optical links, and printed circuit wiring on a backplane. Repeaters are transparent devices that extend the range of a link. Links and repeaters themselves are not accessible in the IB architecture. However, the status of a link (e.g., whether it is up or down) can be determined through the devices which the link connects.

**Switches:** IBA switches are the fundamental routing components for intra-subnet routing. Switches do not generate or consume data packets (they generate/consume management packets). Every destination within the subnet is configured with one or more unique local identifiers (LIDs). Packets contain a destination address that specifies the LID of the destination (DLID). The switch is typically configured out-of-band with a forwarding table that allows it to route the packet to the appropriate output port. This out-of-band configuration of the switch is handled by a separate component within IBA called as the subnet manager, as we will discuss later in this chapter. It is to be noted that such LID-based forwarding allows the subnet manager to configure multiple routes between the same two destinations. This, in turn, allows the network to maximize availability by rerouting packets around failed links through reconfiguration of the forwarding tables.

**Routers:** IBA routers are the fundamental routing component for inter-subnet routing. Like switches, routers do not generate or consume data packets; they

simply pass them along. Routers forward packets based on the packet's global route header and replace the packet's local route header as the packet passes from one subnet to another. The primary difference between a router and a switch is that routers are not completely transparent to the end nodes since the source must specify the LID of the router and also provide the global identifier (GID) of the destination. IB routers use the IPv6 protocol to derive their forwarding tables.

## Messaging

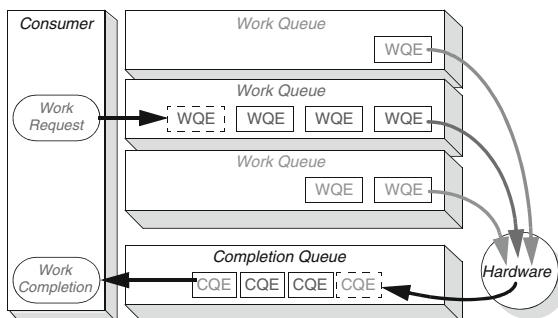
Communication operations in InfiniBand are initiated by the consumer by setting up a list of instructions that the hardware executes. In IB terminology, this facility is referred to as a work queue. In general, work queues are created in pairs, referred to as queue pairs (QPs), one for send operations and one for receive operations. The consumer process submits a work queue element (WQE) to be placed in the appropriate work queue. The channel adapter executes WQEs in the order in which they were placed. After executing a WQE, the channel adapter places a completion queue entry (CQE) in a completion queue (CQ); each CQE contains or points to all the information corresponding to the completed request. This is illustrated in Fig. 2.

*Send operations:* There are three classes of send queue operations: (a) Send, (b) Remote Direct Memory Access (RDMA), and (c) Memory Binding. For a send operation, the WQE specifies a block of data in the consumer's memory. The IB hardware uses this information to send the associated data to the destination. The send operation requires the consumer process

to prepost a receive WQE, which the consumer network adapter uses to place the incoming data in the appropriate location.

For an RDMA operation, together with the information about the local buffer and the destination end point, the WQE also specifies the address of the remote consumer's memory. Thus, RDMA operations do not need the consumer to specify the target memory location using a receive WQE. There are four types of RDMA operations: RDMA write, RDMA write with immediate data, RDMA read, and Atomic. In an RDMA write operation, the sender specifies the target location to which the data has to be written. In this case, the consumer process does not need to post any receive WQE at all. In an RDMA write with immediate data operation, the sender again specifies the target location to write data, but the receiver still needs to post a receive WQE. The receive WQE is marked complete when all the data is written. An RDMA read operation is similar to an RDMA write operation, except that instead of writing data to the target location, data is read from the target location. Finally, there are two types of atomic operations specified of RDMA operations: RDMA write, RDMA write with immediate data, RDMA read, and Atomic. In an RDMA write operation, the sender specifies the target location to which the data has to be written. In this case, the consumer process does not need to post any receive WQE at all. In an RDMA write with immediate operation requests for the same location simultaneously, the second operation is not started till the first one completes. However, such atomicity is not guaranteed when another device is operating on the same data. For example, if the CPU or another InfiniBand adapter modifies data while one InfiniBand adapter is performing an atomic operation on this data, the target location might get corrupted.

*Receive operations:* Unlike the send operations, there is only one type of receive operation. For a send operation, the corresponding receive WQE specifies where to place data that is received, and if requested places the receive WQE in the completion queue. For an RDMA write with immediate data operation, the consumer process does not have to specify where to place the data (if specified, the hardware will ignore it). When the message arrives, the receive WQE is updated with



InfiniBand. Fig. 2 Consumer Queuing Model (Courtesy InfiniBand Standard)

the memory location specified by the sender, and the WQE placed in the completion queue, if requested.

## Overview of Features

IBA describes a multilayer network protocol stack. Each layer provides several features that are usable by applications : (1) link layer features, (2) network layer features, and (3) transport layer features.

### Link Layer Features

*CRC-based data integrity:* IBA provides two forms of CRC-based data integrity to achieve both early error detection as well as end-to-end reliability: invariant CRC and variant CRC. Invariant CRC (ICRC) covers fields that do not change on each network hop. Variant CRC (VCRC), on the other hand, covers the entire packet including the variant as well as the invariant fields.

*Buffering and flow control:* IBA provides an absolute credit-based flow control where the receiver guarantees that it has enough space allotted to receive  $N$  blocks of data. The sender sends only  $N$  blocks of data before waiting for an acknowledgment from the receiver. The receiver occasionally updates the sender with an acknowledgment as and when the receive buffers get freed up. Note that this link-level flow control has no relation to the number of messages sent, but only to the total amount of data that has been sent.

*Virtual lanes, service levels, and QoS:* Virtual lanes (VLs) are a mechanism that allow the emulation of multiple virtual links within a single physical link as illustrated in Fig. 3. Each port provides at least 2 and up to 16 virtual lanes (VL0 to VL15). VL15 is reserved exclusively for subnet management traffic.

*Congestion control:* Together with flow-control, IBA also defines a multistep congestion control mechanism for the network fabric. Specifically, congestion control when used in conjunction with the flow-control mechanism is intended to alleviate head-of-line blocking for non-congested flows in a congested environment. To achieve effective congestion control, IB switch ports need to first identify whether they are the root or the victim of a congestion. A switch port is a root of a congestion if it is sending data to a destination faster than it can receive, thus using up all the flow-control credits available on the switch link. On the other hand, a port is a victim of a congestion if it is unable to send

data on a link because another node is using up all of the available flow-control credits on the link. In order to identify whether a port is the root of the victim of a congestion, IBA specifies a simple approach. When a switch port notices congestion, if it has no flow-control credits left, then it assumes that it is a victim of congestion. On the other hand, when a switch port notices congestion, if it has flow-control credits left, then it assumes that it is the root of a congestion. This approach is not perfect: it is possible that even a root of a congestion may not have flow-control credits remaining at some point of the communication (for example, if the receiver process is too slow in receiving data); in this case, even the root of the congestion would assume that it is a victim of the congestion. Thus, though not required by the IBA, in practice, IB switches are configured to react with the congestion control protocol irrespective of whether they are the root of the victim of the congestion or not.

*Static rate control:* IBA defines a number of different link bit rates, such as 1x SDR (2.5 Gbps), 4x SDR (10 Gbps), 12x SDR (30 Gbps), 4x DDR (20 Gbps), and 4x QDR (40Gbps), and so on. It is to be noted that these rates are signaling rates; with an 8b/10b data encoding, the actual maximum data rates would be 2 Gbps (1x SDR), 8 Gbps (4x SDR), 24 Gbps (12x SDR), 16 Gbps (4x DDR), and 32 Gbps (4x QDR), respectively. To simultaneously support multiple link speeds within a fabric, IBA defines a static rate control mechanism that prevents faster links from overrunning the capacity of ports with slower links. Using a static rate control mechanism, each destination has a timeout value based on the ratio of the destination and source link speeds.

### Network Layer Features

*IB routing:* IB network-layer routing is similar to the link-layer switching, except for two primary differences. First, network-layer routing relies on a global routing header (GRH) that allows packets to be routed between subnets. Second, VL15 (the virtual lane reserved for management traffic) is not respected by IB routers since management traffic stays within the subnet and is never routed across subnets.

*Flow labels:* Flow labels identify a single flow of packets. For example, these can identify packets belonging to a single connection that need to be delivered in order, thus allowing routers to optimize traffic routing when

| Attribute                                                                                           | Reliable Connection                                                                                                                       | Reliable Datagram                                                                                                                                                                                                                                                                                                                                       | Unreliable Datagram                                                          | Unreliable Connection                                                                                                                          | Raw Datagram (both IPv6 & ethertype)                                                                                                               |
|-----------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Scalability</b> (M processes on N processor nodes communicating with all processes on all nodes) | $M^2 \times N$ QPs required on each processor node, per CA.                                                                               | M QPs required on each processor node, per CA.                                                                                                                                                                                                                                                                                                          | M QPs required on each processor node, per CA.                               | $M^2 \times N$ QPs required on each processor node, per CA.                                                                                    | 1 QP required on each end node, per CA.                                                                                                            |
| <b>Reliability</b>                                                                                  | Corrupt data detected                                                                                                                     | Yes                                                                                                                                                                                                                                                                                                                                                     |                                                                              |                                                                                                                                                |                                                                                                                                                    |
|                                                                                                     | Data delivery guarantee                                                                                                                   | Data delivered exactly once                                                                                                                                                                                                                                                                                                                             |                                                                              | No guarantees                                                                                                                                  |                                                                                                                                                    |
|                                                                                                     | Data order guaranteed                                                                                                                     | Yes, per connection                                                                                                                                                                                                                                                                                                                                     | Yes, packets from any one source QP are ordered to multiple destination QPs. | No                                                                                                                                             | Unordered and duplicate packets are detected.                                                                                                      |
|                                                                                                     | Data loss detected                                                                                                                        | Yes                                                                                                                                                                                                                                                                                                                                                     |                                                                              | No                                                                                                                                             | Yes                                                                                                                                                |
|                                                                                                     | Error recovery                                                                                                                            | <b>Reliable.</b> Errors are detected at both the requestor and the responder. The requestor can transparently recover from errors (retransmission, alternate path, etc.) without any involvement of the client application. QP processing is halted only if the destination is inoperable or all fabric paths between the channel adapters have failed. |                                                                              | <b>Unreliable.</b> Packets with some types of errors may not be delivered. Neither source nor destination QPs are informed of dropped packets. | <b>Unreliable.</b> Packets with errors, including sequence errors, are detected and may be logged by the responder. The requestor is not informed. |
| RDMA and ATOMIC operations                                                                          | Yes                                                                                                                                       | Yes                                                                                                                                                                                                                                                                                                                                                     | No                                                                           | Yes : RDMA WRITEs<br>No: RDMA READs & ATOMICs                                                                                                  | No                                                                                                                                                 |
| Bind memory window                                                                                  | Yes                                                                                                                                       | Yes                                                                                                                                                                                                                                                                                                                                                     | No                                                                           | Yes                                                                                                                                            | No                                                                                                                                                 |
| IBA unreliable multicast support                                                                    | No                                                                                                                                        | No                                                                                                                                                                                                                                                                                                                                                      | Yes                                                                          | No                                                                                                                                             | No                                                                                                                                                 |
| Raw multicast                                                                                       | No                                                                                                                                        | No                                                                                                                                                                                                                                                                                                                                                      | No                                                                           | No                                                                                                                                             | Yes                                                                                                                                                |
| Message size                                                                                        | Message size 0 to $2^{31}$ bytes. Smaller max size may be negotiated by Connection Management. A message may consist of multiple packets. |                                                                                                                                                                                                                                                                                                                                                         | <b>Single PMTU packet</b> datagrams – 0 to 4,096 bytes.                      | Message size 0 to $2^{31}$ bytes. Smaller max size may be negotiated by Connection Management. A message may consist of multiple packets.      | <b>Single PMTU packet</b> datagrams – 0 to 4,096 bytes.                                                                                            |
| Connection oriented?                                                                                | <b>Connected.</b> The client connects the local QP to one and only one remote QP. No other traffic flows over these QPs.                  | <b>Connectionless.</b> Appears connectionless to the client – uses one or more End-to-End contexts per CA to provide reliability service.                                                                                                                                                                                                               | <b>Connectionless.</b> No prior connection is needed for communication.      | <b>Connected.</b> The client connects the local QP to one and only one remote QP. No other traffic flows over these QPs.                       | <b>Connectionless.</b> No prior connection is needed for communication.                                                                            |

InfiniBand. Fig. 3 IB Transport Services (Courtesy InfiniBand Standard)

using multiple paths for communication. IB routers are allowed to change flow labels as needed, for example, to distinguish two different flows which have been given the same label. However, during such relabeling, routers ensure that all packets corresponding to the same flow will have the same label.

### Transport Layer Features

*IB transport services:* IBA defines four types of transport services (reliable connection, reliable datagram, unreliable connection, and unreliable datagram) and two types of raw communication services (raw IPv6 datagram and raw Ethertype datagram) to allow for

encapsulation of non-IBA protocols. The transport service describes the degree of reliability and how the data is communicated.

As illustrated in Fig. 3, each transport service provides a trade-off in the amount of resources used and the capabilities provided. For example, while the reliable connection provides the most features, it has to establish a QP for each peer process it communicates with, thus requiring a quadratically increasing number of QPs with the number of processes. Reliable and unreliable datagram services on the other hand utilize fewer resources since a single QP can be used to communicate with all peer processes.

*Automatic path migration:* The Automatic Path Migration (APM) feature of InfiniBand allows a connection end point to specify a fallback path, together with a primary path for a data connection. All data is initially sent over the primary path. However, if the primary path starts throwing excessive errors or is heavily loaded, the hardware can automatically migrate the connection to the new path, after which all data is only sent over the new path.

*Message-level flow control:* Together with the link-level flow control, for reliable connection, IBA also defines an end-to-end message-level flow-control mechanism. This message-level flow control does not deal with the number of bytes of data being communicated, but rather with the number of messages being communicated. Specifically, a sender is only allowed to send as many messages that use up receive WQEs as there are receive WQEs posted by the consumer. That is, if the consumer has posted 10 receive WQEs, after the sender sends out 10 send or RDMA write with immediate data messages, the next message is not communicated till the receiver posts another receive WQE.

*Shared Receive Queue (SRQ):* Introduced in the InfiniBand 1.2 specification, shared receive queues (SRQs) were added to help address scalability issues with InfiniBand memory usage. When using the RC transport of InfiniBand, one QP is required per communicating peer. To prepost receives on each QP, however, can have very high memory requirements for communication buffers. To give an example, consider a fully connected MPI job of 1K processes. Each process in the job will require 1K - 1 QPs, each with n buffers of size s posted to it. Given a conservative setting of  $n = 5$  and  $s = 8 \text{ KB}$ , over 40 MB of memory per process

would be required simply for communication buffers that may not be used. Given that current InfiniBand clusters now reach 60K processes, maximum memory usage would potentially be over 2 GB per process in that configuration.

Recognizing that such buffers could be pooled, SRQ support was added, so instead of connecting a QP to a dedicated RQ, buffers could be shared across QPs. In this method, a smaller pool can be allocated and then refilled as needed instead of pre-posting on each connection.

*Extended Reliable Connection (XRC):* eXtended reliable connection (XRC) provides the services of the RC transport, but defines a very different connection model and method for determining data placement on the receiver in channel semantics. This mode was mostly designed to address multi-core clusters and lower memory consumption of QPs. For one process to communicate with another over RC, each side of communication must have a dedicated QP for the other. There is no distinction as to the node in terms of allowing communication. In XRC, a process no longer needs to have a QP to every process on a remote node. Instead, once one QP to a node has been set up, messages can be routed to the other processes by giving the address/number of a shared receive queue (SRQ). In this model, the number of QPs required is based on the number of nodes in the job rather than the number of processes.

## Management and Services

Together with the regular communication capabilities, IBA also defines an elaborate management semantics and several services. The basic management messaging is handled through special packets called as management datagrams (MADs). The IBA management model defines several management classes as described below.

*Subnet Management:* The subnet management class deals with discovering, initializing, and maintaining an IB subnet. It also deals with interfacing with diagnostic frameworks for handling subnet and protocol errors. For subnet management, each IB subnet has at least one subnet manager (SM). An SM can be a collection of multiple processing elements, though they appear as a single logical management entity. These processes can internally make management decisions, for example, to manage large networks in a distributed fashion. Each

channel adapter, switch, and router, also maintains a subnet management agent (SMA) that interacts with the SM and handles management of the specific device on which it resides as directed by the SM. An SMA can be viewed as a worker process, though in practice it could be implemented as hardware logic.

*Subnet Administration:* Subnet administration (SA) deals with providing consumers with access to information related to the subnet through the subnet management interface (SMI). Most of this information is collected by the SM, and hence the SA works very closely with the SM. In fact, in most implementations, a single logical SM processing unit performs the tasks of both the SM and the SA. The SA typically provides information that cannot be locally computed to consumer processes (such as data paths, SL-to-VL mappings, and partitioning information). Further, it also deals with inter-SM management such as handling standby SMs.

*Communication Management:* Communication management deals with the protocols and mechanisms used to establish, maintain, and release channels for RC, UC, and RD transport services. At creation, QPs are not ready for communication. The CM infrastructure is responsible for preparing the QPs for communication by exchanging appropriate information.

*Performance Management:* The performance management class allows performance management entities to retrieve performance and error statistics from IB components. There are two classes of statistics: (a) mandatory statistics that have to be supported by all IB devices and (b) optional statistics that are specified to allow for future standardization. There are several mandatory statistics supported by current IB devices including the amount of bytes/packets sent and received, transmit queue depth at various intervals, number of ticks during which the port had data to send but had no flow-control credits, etc.

*Device Management:* Device management is an optional class that mainly focuses on devices that do not directly connect to the IB fabric, such as I/O devices and I/O controllers. An I/O unit (IOU) containing one or more IOCs is attached to the fabric using a channel adapter. The channel adapter is responsible for receiving packets from the fabric and delivering them to the relevant devices and vice versa. The device management class does not deal with directly managing the end

I/O devices, but rather focuses on the communication with the channel adapter. Any other communication between the adapter and the I/O devices is unspecified by the IB standard and depends on the device vendor.

## InfiniBand Today

To achieve high performance, adapters are continually updated with the latest in speeds (SDR, DDR and QDR) and I/O interface technology. Earlier InfiniBand adapters supported only PCI-X I/O interface technology, but have now moved to PCI-Express (x8 and x16) and Hyper-Transport. More recently, QDR cards have moved to the PCI-Express 2.0 standard to further increase I/O bandwidth. Although InfiniBand adapters have traditionally been add-on components in expansion slots, they have recently started moving onto the motherboard. Many boards including some designed by Tyan and SuperMicro include adapters directly on the motherboard.

In 2008, Mellanox Technologies released the fourth generation of their adapter called ConnectX. This was the first InfiniBand adapter to support QDR speed. This adapter in addition to increasing speeds and lowering latency included support for 10 Gigabit Ethernet. Depending on the cable connected to the adapter port, it either performs as an Ethernet adapter or an InfiniBand adapter. The current version of ConnectX adapter, ConnectX-2 supports advanced offloading of arbitrary lists of send, receive and wait tasks to the network interface. This enables network offloaded collective operations, which are critical to scaling messaging libraries to very large systems.

The IBA specification does not specify the exact software interface through which hardware is accessed. OpenFabrics is an industry consortium that has specified an interface through which IB hardware can be accessed. It also includes support for other RDMA capable interconnects, such as iWARP.

A few current usage environments for IB are mentioned below:

*Message passing interface:* MVAPICH [3] and MPICH2 [4] are implementations of MPI over InfiniBand. They are based on MPICH1 [5] and MPICH2 [6], respectively. As of January 2011, these stacks are used by more than 1,330 organizations in 60 countries around

the world. There have been more than 53,000 downloads from the Ohio State University site directly. These stacks are empowering many powerful computers, including “Ranger,” which is the sixth ranked cluster according to the November 2008 TOP500 rankings. It has 62,976 cores and is located at the Texas Advanced Computing Center (TACC). MVAPICH/MVAPICH2 work with many existing IB software layers, including the OpenFabrics Gen2 interface. These software stacks are also available in an integrated manner with the software stacks of many other InfiniBand vendors, server vendors, and Linux distributors (such as RedHat and SuSE).

The OpenMPI project [7] is yet another implementation of MPI. Open MPI also works on InfiniBand using the OpenFabrics interface and is widely used. In addition to the open-source implementations of MPI, there are several commercial implementations of MPI, such as Intel MPI and Platform MPI.

*File systems:* Modern parallel applications use terabytes and petabytes of data. Thus, parallel file systems are significant components in designing large-scale clusters. Multiple features (low latency, high bandwidth and low CPU utilization) of IB are quite attractive for designing such parallel file systems. There are implementations of widely used parallel file systems such as Lustre [8] and Parallel Virtual File Systems [9] over InfiniBand using OpenFabrics interface.

*Data-centers:* Together with scientific computing domains such as the MPI and parallel file-systems, IB is also making significant inroads into enterprise computing environments such as web data centers and financial markets. There are two ways in which sockets-based applications can execute on top of IB. In the first approach, IB provides a simple driver (known as IPoIB) that emulates the functionality of Ethernet on top of IB. This allows the kernel-based TCP/IP stack to communicate over IB. However, this approach does not utilize any of the advanced features of IB and thus is restricted in the performance it can achieve. The second approach utilizes the Sockets Direct Protocol (SDP) [10]. SDP is a byte-stream transport protocol that closely mimics TCP socket’s stream semantics. It is an industry-standard specification for IB that utilizes advanced capabilities provided by the network stack (such as the hardware offloaded protocol stack and RDMA) to achieve high

performance without requiring modifications to existing sockets-based applications.

Recently, companies such as RNA Networks are providing memory virtualization using IB. These products are based on the core concept of RDMA and os-bypass protocols.

## Future Directions

Strong features and mechanisms of IB, together with the continuous advancements in IB hardware and software products, are leading many high-end clusters and parallel file systems. In less than 9 years of its introduction, the open standard IB is already claiming more than 40% share in TOP500 systems. This market share is expected to grow significantly during the next few years. Multiple parallel file systems, storage systems, database systems, and visualization systems are also using InfiniBand. As indicated in this entry, InfiniBand is also gradually moving into campus backbones and WANs. This is opening up multiple new applications and environments to utilize InfiniBand.

IB 4x QDR products are currently available in the market in a commodity manner. These are getting to be used in modern clusters with multi-core nodes (8–16 cores per node). IB Roadmap also talks about the availability of FDR (Fourteen Data Rate) products during the 2012 time frame. The FDR technology promises to use 64/66 bit encoding and deliver 56 Gbps data rate. The next generation IB products with EDR (Enhanced Data Rate) technology, supporting 100 Gbps data rate, are planned for 2013. The future IB technologies promise to build high performance, scalable and balanced systems with next generation multi-core nodes (16–100 cores per node) and accelerate the field of high-end computing.

## Bibliographic Notes and Further Reading

Open-standard high-performance networking technologies, such as InfiniBand have spurred a lot of interest among researchers. This is an active area of research. There are several sub-domains: scalability and performance, routing, fault-tolerance, design of file-systems etc.

The reader is encouraged to peruse [11–16] for more details on scalability and performance. Routing and

management are covered in [17–20]. Fault-tolerance issues are covered in [21, 22].

## Bibliography

1. Becker DJ et al (1996) Beowulf: a parallel workstation for scientific computation. In: International parallel processing symposium, Honolulu, 1996
2. InfiniBand Trade Association. InfiniBand Architecture Specification, vol 1, Release 1.0. <http://www.infinibandta.com>
3. Network-Based Computing Laboratory. MVAPICH: MPI for InfiniBand, 10GigE/iWARP and RoCEE. <http://mvapich.cse.ohio-state.edu/>
4. Liu J, Jiang W, Wyckoff P, Panda DK, Ashton D, Buntinas D, Gropp W, Toonen B (2004) Design and implementation of MPICH2 over infiniBand with RDMA support. In: Proceedings of international parallel and distributed processing symposium (IPDPS '04), Santa Fe, April 2004
5. Gropp W, Lusk E, Doss N, Skjellum A (1996) A high-performance, portable implementation of the MPI, message passing interface standard. Technical report, Argonne National Laboratory and Mississippi State University
6. Liu J, Jiang W, Wyckoff P, Panda DK, Ashton D, Buntinas D, Gropp B, Tooney B (2004) High performance implementation of MPICH2 over InfiniBand with RDMA support. In: IPDPS, Santa Fe, 2004
7. Garbriel E, Fagg GE, Bosilica G, Angskun T, Dongarra JJ, Squyres JM, Sahay V, Kambadur P, Barrett B, Lumsdaine A, Castain RH, Daniel DJ, Graham RL, Woodall TS (2004) Open MPI: goals, concept, and design of a next generation MPI implementation. In: Proceedings of the 11th European PVM/MPI users' group meeting, Budapest, 2004
8. Cluster File System Inc. Lustre: scalable clustered object storage. <http://www.lustre.org/>
9. PVFS2 Developer Team. Parallel Virtual File System, Version 2. <http://www.pvfs.org/pvfs2/>, November 2003
10. SDP Specification. <http://www.rdmaconsortium.org/home>
11. Sur S, Chai L, Jin H-W, Panda DK (2006) Shared receive queue based scalable MPI design for infiniband clusters. In: International parallel and distributed processing symposium (IPDPS), Rhodes Island
12. Sur S, Chai L, Jin H-W, Panda DK (2006) RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits. In: International symposium on principles and practice of parallel programming (PPoPP), New York
13. Koop M, Kumar R, Panda DK (2008) Can software reliability outperform hardware reliability on high performance interconnects? A case study with MPI over InfiniBand. In: 22nd ACM international conference on supercomputing (ICS08), June 2008
14. Koop M, Sridhar J, Panda DK (2008) Scalable MPI design over InfiniBand using exTended reliable connection. In: IEEE international conference on cluster computing (Cluster 2008), Tsukuba, September 2008
15. Koop M, Jones T, Panda DK (2007) Reducing connection memory requirements of MPI for InfiniBand clusters: a message coalescing approach. In: 7th IEEE international symposium on cluster computing and the grid (CCGrid07), Rio de Janeiro, May 2007
16. Liu J, Mamidala A, Panda DK (2004) Fast and scalable collective operations using remote memory operations on VIA-based clusters. In: International parallel and distributed processing symposium (IPDPS '03), Nice, April 2004
17. Skeie T, Lysne O, Theiss I (2002) Layered shortest path (lash) routing in irregular system area networks. In: Proceedings of the 16th international parallel and distributed processing symposium, IPDPS '02, Washington, DC, 2002. IEEE Computer Society, pp 194
18. Koop VAM, Moody A, Mamidala A, Narravula S, Panda DK (2007) Hot-spot avoidance with multi-pathing over InfiniBand: an MPI perspective. In: 7th IEEE international symposium on cluster computing and the grid (CCGrid07), Rio de Janeiro, May 2007
19. Vishnu A, Mamidala AR, Panda DK (2005) Performance modeling of subnet management on fat tree InfiniBand networks using OpenSM. In: Workshop on system management tools on large scale parallel systems, held in conjunction with IPDPS, Denver
20. Hoer T, Schneider T, Lumsdaine A (2009) Optimized routing for large-scale InfiniBand networks. In: 17th Annual IEEE symposium on high performance interconnects (HOTI 2009), New York, August 2009
21. Gao Q, Yu W, Huang W, Panda DK (2006) Application-transparent checkpoint/restart for MPI programs over InfiniBand. In: Proceedings of the international conference on parallel processing (ICPP), Columbus, August 2006
22. Ouyang X, Gopalakrishnan K, Panda DK (2009) Accelerating checkpoint operation by node-level write aggregation on multicore systems. In: Submitted to international symposium on cluster computing and the grid (CCGrid), May 2009

## Instant Replay

### ► Debugging

## Instruction-Level Parallelism

Instruction-level parallelism is the possibility of scheduling multiple machine instructions from a single program so that their executions overlap in time. Instruction-level parallelism was exploited by some early machines such as the CDC 6600 and IBM 360/91.

Today, superscalar, EPIC, and VLIW machines take advantage of this form of parallelism.

## Related Entries

- [Branch Predictors](#)
- [Control Data 6600](#)
- [EPIC Processors](#)
- [HPS Microarchitecture](#)
- [IBM Power Architecture](#)
- [IBM System/360 Model 91](#)
- [Intel Core Microarchitecture, x86 Processor Family](#)
- [Modulo Scheduling and Loop Pipelining](#)
- [Multiflow Computer](#)
- [Superscalar Processors](#)
- [VLIW Processors](#)

## Instruction Systolic Arrays

- [Systolic Arrays](#)

## Intel Celeron

- [Intel Core Microarchitecture, x86 Processor Family](#)

## Intel Core Microarchitecture, x86 Processor Family

JOSEF WEIDENDORFER  
Technische Universität München, München, Germany

### Synonyms

[Core2-Duo / Core2-Quad Processors](#); [Pentium](#); [Intel Celeron](#)

### Definition

The *Intel Core Microarchitecture* is the name of the internal architecture of multicore processor implementations from *Intel Corporation* introduced in 2006. The processing cores of the Core Microarchitecture use

sophisticated ways to get the highest level of *Instruction Level Parallelism* (ILP) from sequential instruction streams, using out-of-order, superscalar pipelining. The on-chip memory hierarchy employs multiple levels of inclusive caches, uses different kinds of hardware prefetching techniques, and is partly shared among the cores on the chip.

These processors belong to the family of *x86-compatible* processors, originating in the 8086 processor from Intel in 1978. Their Instruction Set Architecture (ISA) – i.e., the programming interface – is classified as CISC (Complex Instruction Set Computing), which means that the assembly code is compact but quite complex and irregular. The 32-bit and 64-bit extensions of the original x86 ISA are called *IA-32* and *Intel 64* by Intel, respectively. Starting their success in Personal Computers (PC) from IBM, processors with these ISAs are dominating today in desktop and laptop machines, but are also used heavily in the server market as well as in HPC systems.

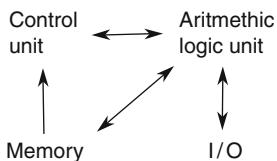
### Discussion

This entry describes Intels Core Microarchitecture as example of a machine with Instruction Level Parallelism (ILP), that is, it focuses on how parallelism is exploited in a processor core of this micro-architecture, executing a single sequential instruction stream. For detailed description of techniques used as well as more about these processors when embedded into shared-memory multiprocessors, see related entries as mentioned in the according section at the end of this entry. Before the description of the micro-architecture, a short overview of microprocessor components, ILP techniques, and the x86 ISA is given as basis. One has to note that the term “processor architecture” itself is used in different ways, one being the pure programming interface (i.e., the ISA), and the other describing the inner workings of the processor, i.e., the micro-architecture.

### Introduction

#### Microprocessor components

Processor cores of Intels x86 family, as almost any processor core nowadays, work according to the von-Neumann architecture: the main components are a control unit, an arithmetic logic unit (ALU), memory, and



**Intel Core Microarchitecture, x86 Processor Family, Fig. 1**  
Components of a von-Neumann architecture

an I/O subsystem as well as connections between all components. This can be seen in [Fig. 1](#). The first two make up a *Central Processing Unit* (CPU). The memory consists of a one-dimensional array of equal-sized memory cells, which both hold data used in computation done by the ALU as well as program code to be executed by the control unit. This makes it possible to treat program code as data, thus vastly simplifying mechanisms needed, for example, to load/store data and code from/to I/O devices. It also allows programs to generate new code, or modify their code. However, as memory regularly needs to be accessed both from the control unit as well as the ALU, this becomes a potential bottleneck. In fact, memory performance (access latency and bandwidth) is a limiting factor for the performance of modern processors (see entry [►Memory Wall](#), also called the von-Neumann bottleneck). The reason is that memory on the one hand, and control unit/ALU on the other hand, are manufactured as separate chips, such that memory accesses have to cross chip boundaries. This is done for more flexibility, for example, enabling the use of different manufacturing technologies. In the von-Neumann architecture, a program is executed by using an instruction pointer. An instruction at the memory pointed to by the instruction pointer is fetched from memory, and afterward interpreted by the control unit. For every instruction type, the control unit has to know the exact sequence of actions to do, be it the execution of a calculation via the ALU, probably involving loading/storing data from memory, or directly influencing the instruction pointer, which otherwise is just incremented to point at the next instruction.

As can be seen from the above description, fast program execution requires fast memory access. This can be improved by not forcing every memory access to reach out to the off-chip memory module, which

is relatively far away. Instead, buffers are introduced which are able to hold copies of memory cells. The fastest buffers are called the register file, and are accessed explicitly by mentioning register numbers in the instructions, and adding instructions to move data between memory and registers. Further, larger on-chip buffers are added, so-called *caches*, which are able to transparently hold copies of memory cells. As access time to these buffers are reciprocal to buffer sizes, it is beneficial to have a hierarchy of caches. Together with register files on the one end and main memory on the other end, this is called the memory hierarchy. As memory accesses for fetching instructions are different from other accesses, it is beneficial to provide separate buffers for these uses, that is, have separate instruction and data caches on the first level. On the second, larger level, a unified cache is used.

One way to make a processor run faster is increasing the clock frequency, another one is to do multiple things simultaneously. Regarding the latter, from the point of view of a programmer, the worst way of doing things simultaneously is the requirement to explicitly have to specify what can be done in parallel and what not. Therefore, processor architects first try to exploit parallelism which is implicitly available in a single sequential instruction stream. An easy way is to increase bit-level parallelism: as long as it is useful for programs without wasting too much resources, it is the best to work with data paths as wide as possible. This is reflected by the x86 ISA starting as 16-bit architecture with the 8086, going to 32-bit with the 80386, and recently to 64-bit. The respective bit-width is used as register width and width of signal paths for data and memory addresses. Similar is data-level parallelism (DLP): if multiple data can be worked on in the same way, these can be packed in vector registers with multiple calculations done synchronously in one clock cycle in enhanced ALUs, following the SIMD (Single Instruction Multiple Data) concept. The according x86 ISA enhancements are called MMX and SSE (see [►Vector Extensions, Instruction-Set Architecture \(ISA\)](#)). Another type of parallelism possible with one instruction stream is called *Instruction Level Parallelism* (ILP). It exploits the fact that sub-actions of one or multiple instructions (if there are no data dependencies) can

be done at the same time. The techniques are described in the next section in more detail, as this is needed to understand the micro-architecture of modern processors. The last type of parallelism is called *Thread Level Parallelism* (TLP), where multiple instruction streams work in parallel to increase performance of a program. This needs special support at different levels: hardware must be added to allow for multiple processors/processor cores to be put together, for example, special care is needed to keep the multiple copies found in different caches consistent. Instructions for synchronization and communication among the processors have to be added. Also, the software, starting with the Operating System, needs to be able to control multiple streams of execution. Finally, correctness of a parallel program is far more difficult to check. Therefore, on smaller systems with only one processor chip, for a long time, only execution of a single instruction stream at a time was supported. However, after exploiting DLP/ILP to the maximum, only increasing the clock frequency was available for improving performance.

From the beginning of microprocessor technology, it could be observed that there was a constant development of miniaturization. This observation manifested in the so-called Moore's Law: every 18 month, the number of transistor budget available for use on one chip doubles. Astonishingly, this "law" holds true up to now and into the foreseeable future. Processor manufacturers were able to extract ever increasing computing power from the exponentially growing transistor budget, by enlarging caches, increasing bit- and data-level parallelism as well as ILP to the maximum (the latter making the control units extremely complex). At the same time, miniaturization allowed to increase the clock frequency. However, this came to a halt in recent years. The high clock rates reached, together with the small structure widths, result in a lot of energy lost because of leakage current in the chip, and thus requires expensive cooling. To be able to further sell processors, the only way for processor manufactures was to add TLP inside one chip while keeping clock rate constant. This was the beginning of the multicore era, with the need to feed current processor chips with parallel programs in the hope to further get exponential performance increase for programs from Moore's law still proving true. Thus, modern processes consist of multiple control/execution units, working in parallel. Yet,

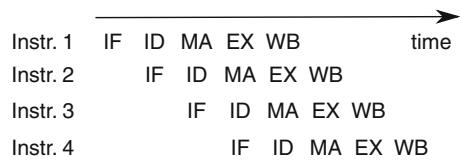
the cores can share resources, the most obvious being caches.

### Instruction Level Parallelism

As written before, Instruction Level Parallelism (ILP) exploits the fact that sub-actions of one or multiple consecutive instruction in one sequential instruction stream can be executed simultaneously. The most important principle is that instructions can be subdivided into sub-actions, also called stages. Each instruction stage is executed in one clock cycle by different hardware. While one instruction "flows" through the different hardware stages (the so-called pipeline), multiple instructions can be in execution at the same time, as shown in Fig. 2. Typical stages are IF (instruction fetch), ID (instruction decode), MA (memory access), EX (execute calculation), and WB (write back).

For a given chip manufacturing technology (typically identified by a structure width metric such as 90 nm, 60 nm, or 45 nm), the time needed for a signal to reliably bypass a gate level is fixed. Smaller stages need less successive gate levels inside, therefore the clock rate can be higher. Yet, the pipeline design makes sure that at every clock cycle one instruction will finish execution. Thus, by increasing the number of stages (with every stage responsible for simpler sub-actions), the performance is increased. However, the number of stages is limited by a few factors: simplifications cannot be done indefinitely, but more important, different conflict cases can appear:

- Resource conflicts happen when hardware resources shared by multiple stages need to be accessed at the same time, such as access paths to registers/caches. Then, the later instructions have to wait. This can be overcome by duplicating resources, for example, by adding multiple access ports to register files.



**Intel Core Microarchitecture, x86 Processor Family. Fig. 2**  
Pipelined execution of instructions

- Data conflicts happen when there is a data dependence between instructions, allowing the execution of the later instruction not to proceed in the pipeline as long as a given result is not available. To avoid this, a compiler should schedule instructions with data dependencies as far away from each other as possible. Anti-dependencies can be avoided by using temporary (shadow) registers. Another possibility is to allow instructions to be reordered, that is, instructions to overtake each other in the pipeline (so-called out-of-order). Aside from the need for complex dependence checking hardware, a final stage needs to be added to get instruction retirement in correct order again.
- Control conflicts arise when the next instruction to be executed is not known. This happens with conditional branches, where the condition is yet to be calculated. Thus, the pipeline cannot be kept filled. To overcome this, the compiler should schedule calculation of branch conditions way before the branch. Another solution is to predict the condition, and to speculatively execute instructions following the branch. If the prediction is wrong, results of already, but wrongly executed stages have to be thrown away. To minimize this problem, sophisticated branch prediction algorithms are used, which store the history of previous branch decisions in a so-called branch target buffer (BTB).

Complementing the pipelining principle described up to now, another technique to exploit ILP is superscalarity. For every stage of a pipeline, there are multiple hardware units available. This allows to feed multiple instructions into the *superscalar* pipeline in one clock cycle. Typically, implementation involves multiple queues filling up with instructions to be executed by different execution units. Extra care has to be taken to meet data dependencies. The number of instructions that can be executed by duplicated execution units for the same pipeline stages is called the *issue width* of the pipeline at this stage. For the execution stage involving the ALU, usually, there is not a complete duplication done. Instead, floating point units, especially for multiplication or division, are available only once, as these take up much space. Therefore, the issue width of the execution stage usually is higher than the rest of the pipeline to minimize the probability that

there are not enough ALUs of needed type available for the instructions in flow.

With this knowledge, the components of the Core Pipeline in the later section are easy to explain. For more details on Pipelining, see the corresponding entry.

## CISC Versus RISC

The programming interface of a processor is called its *Instruction Set Architecture* (ISA). The ISA of x86-compatible processors is classified as CISC (Complex Instruction Set Computing), which means that the assembly code is compact but quite complex and irregular. When the first x86 processor was developed by Intel, this was the standard way: CISC is found in the Motorola 68000, the Zilog Z80, and in IBM mainframe CPUs at that time. A compact ISA was a benefit, as memory was expensive. As programmers often used assembly language, instructions for specific use cases and complex addressing modes made sense. Later in the 1980s, compilers did not make use of such complex assembly instructions, and thus, processors with simpler ISA were designed, reducing resource consumption of the decoding step. Such designs belong to the RISC (Reduced Instruction Set Computing) idea. They use regular instructions of same size, have load/store instructions as only means to access memory, and work with large register files. Examples are the PowerPC, MIPS, SPARC, and ARM processor designs. Traditionally, each CISC instruction was executed by running a microcode program. Starting with the Intel Pentium (see below), instructions are decoded into a sequence of microoperations, similar to RISC instructions. Therefore, aside from the added cost of a decoding step, there is not much difference between CISC and RISC implementations. However, the compactness of CISC code reduces memory pressure, and allows for smaller L1 instruction caches.

## History of x86 Processors

In 1978, Intel introduced its 8086 processor, a 16-bit architecture meant to compete with the Z80 processor famous at that time. Clocking at 10 MHz, it consisted of around 29,000 transistors, and was able to address 1 MB of external memory by adding 16-bit offsets to special segment registers providing the upper bits. The 8086 had seven general-purpose registers plus the stack

pointer, a flag register, and the segment registers. Using an extended version called 8088, IBM sold its first PC in 1981 with an Intel processor. While being able to only do integer calculation, it was possible to add the floating point coprocessor 8087. A significant enhancement to the x86 family happened in 1985 with the introduction of the i386 processor, which was clocked at 33 MHz, added a 32-bit execution mode, enlarged registers and most importantly, introduced virtual memory with 32-bit flat address spaces. The successor i486 integrated the floating point unit.

Intel's x86 processor success strongly relates to the early success of the component-based design of the IBM PC, resulting in a mass market for x86-compatible processors, and thus also competition. To make it more difficult for competitors to place their products, Intel registered the trademark "Pentium" as name of the next processor update in 1993. As the first in the x86 family, this processor ("P5") added 8 kB instruction and 8 kB data caches on-chip, and allowed for an off-chip L2 cache. Further, a superscalar pipeline design was introduced, with two instruction decoders and two ALUs. To reduce pipeline stalls, it introduced a branch prediction unit. A later version, the "Pentium MMX," extended the x86 ISA by data-level parallelism with vector registers and operations working on 8 bytes at once. This parallelism has to be found by the compiler, and correct data layout needs to be used. Yet, this technique proved successful for a set of applications, and is still being extended (Intel recently announced x86 processors supporting vector operations with 64-byte width, marketed as LNI: Larrabee New Instructions). Similar instructions can, for example, be found in the PowerPC (called AltiVec) or SPARC (VIS).

The micro-architecture following P5 in 1995, called P6, increased the superscalarity in the decoding stage to three, and execution stage to five. To effectively allow to exploit these units, support for out-of-order execution in the pipeline was added. While increasing the complexity significantly with hardware for data dependence checking, more physical registers, and new stages, for example, for register renaming and instruction reordering, it significantly improves ILP even with badly scheduled code by some compiler. The P6 needs an increased number of pipeline stages, for example, in total 12 stages for execution of integer instructions. As further improvement, the L1 caches were doubled in size, and chips were packaged together

with modules for 256 kB up to 1 MB of L2 caches. In the following years, the P6 architecture got successive improvements - partly triggered by competition from the company AMD: the "Pentium-III" implementation had improved, 16-byte wide vector support (SSE), on-chip L2 caches, and almost reached 1 GHz clock rate in 1999.

At this time, the clock rate was a selling point for x86 compatible processors. Competitor AMD had a small advantage in this regard. Therefore, the next micro-architecture from Intel, Netburst, was designed to reach extremely high clock rates. While forcing L1 cache sizes back to 8 kB again, more importantly, pipeline stages had to be simplified, resulting in stage numbers of 20 up to 31 in later implementations. However, performance often does not increase accordingly because of much higher probability of pipeline conflicts, and thus pipeline stalls. Netburst has 3 integer and 2 floating point ALUs. However, together with the extremely deep pipeline, one instruction stream rarely can exploit these execution units. As important enhancement, Intel integrated, for the first time, a form of thread-level parallelism (TLP) into its processors. *Simultaneous Multithreading* (SMT), by Intel called Hyperthreading, keeps two instruction stream contexts with own register sets for one superscalar pipeline. The pipeline is filled with instructions from both streams, effectively reducing pipeline conflicts and enhancing utilization of the ALUs. Intel promised a performance increase up to 40%. However, with a big drawback: programs need to be parallelized. An interesting enhancement of Netburst is the *Trace Cache*, an instruction cache containing up to 16 k decoded microoperations. Further, the L2 size was improved up to 2 MB. An important addition to the ISA in the later implementations of Netburst was actually only in reaction to competitor AMD: Intel had to introduce a 64-bit extension, compatible to the 64-bit extension of AMD processors. The problem was that Intel originally intended to reserve 64-bit for its new Itanium architecture for the high-end server and HPC market. With x86 processors supporting 64-bit, customers could choose the cheaper platform for their needs. To emphasize intentions, the Itanium ISA was called IA-64 by Intel, and the 32-bit x86-ISA was renamed IA-32 at that time. After first naming the 64-bit x86 extension EM64T, then IA-32e, it is now simply called "Intel 64" to avoid confusion.

In some way, Hyperthreading allowed Intel to estimate the acceptance of TLP on a chip, and lead to increased interest in multithreaded programming and language extensions such as OpenMP. On the other hand, the race for highest clock rates showed the limits reached: together with the small structure lengths of technology, leakage effects raised the power consumption and heat disposal, requiring expensive cooling techniques. The solution is to cut the clock rate and add more processor cores on a chip instead. The last implementation of Netburst, "Pentium D", was built as dual-core, combining two single-core dies in one package. The multicore era for x86-compatible processors had started.

## The Core Microarchitecture

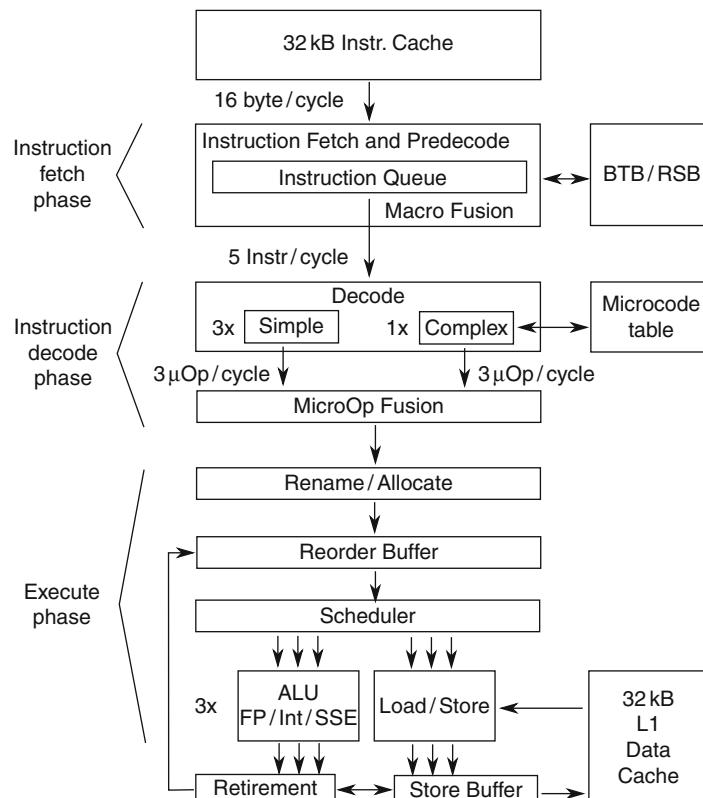
After the architectural problems of Netburst, the following micro-architecture from Intel, just called "Core," is explicitly designed as basis for multicore processors. For a performance increase in contrast to the previous single-core processors, the clock rate can be reduced

with multiple cores. This allows for more complex pipeline stages than found in Netburst, and thus a reduced pipeline depth. The Core architecture goes back to 14 stages, resulting in reduced potential for pipeline stalls, and increased performance when compared to Netburst at same clock rate. Intel used its successful P6 architecture as base for Core. The first implementations, called "Core-2" were introduced in 2006, first as dual-core, later as quad-core designs. The quad-core versions using the Core Microarchitecture always can be seen as coupling two dual-core designs on one chip. The later processor implementations for servers (Xeon) contain a shared L3 for all four cores.

### Pipeline Structure

A simplified version of the pipeline of the Core Microarchitecture with its 14 stages is shown in Fig. 3.

The instruction fetch stage is able to fetch 16 bytes per cycle, filled from a 32 kB instruction cache, and assisted by branch prediction. The prediction uses a branch target buffer (BTB) which holds the history of last branch decisions indexed by instruction address.



Intel Core Microarchitecture, x86 Processor Family. Fig. 3 Block diagram of the Core Microarchitecture

Further, to accurately predict function returns, it has a 16-entry *Return Stack Buffer* (RSB). The prefetched bytes are put into the predecoding stage, where length of instructions is determined, and decoding of any prefix bytes is done (the need for a two-phase decoding shows the reached level of complexity of the x86 ISA). Pre-decoded instructions are put into a instruction queue. From there, four decoders can decode up to five original x86 instructions per cycle. This is possible as some often-used two-instruction sequences are recognized at once (called *Macro-Fusion*). One of the decoders is able to handle complex instructions by looking up corresponding sequences of operations in a microcode table. The generated sequence of microoperations ( $\mu$ Ops) is checked for possibility of so-called  $\mu$ Op-fusion. Some primitive sequences can be joined to form more complex microoperations. This saves bandwidth in the pipeline. An example would be a comparison operation directly followed by a branch.

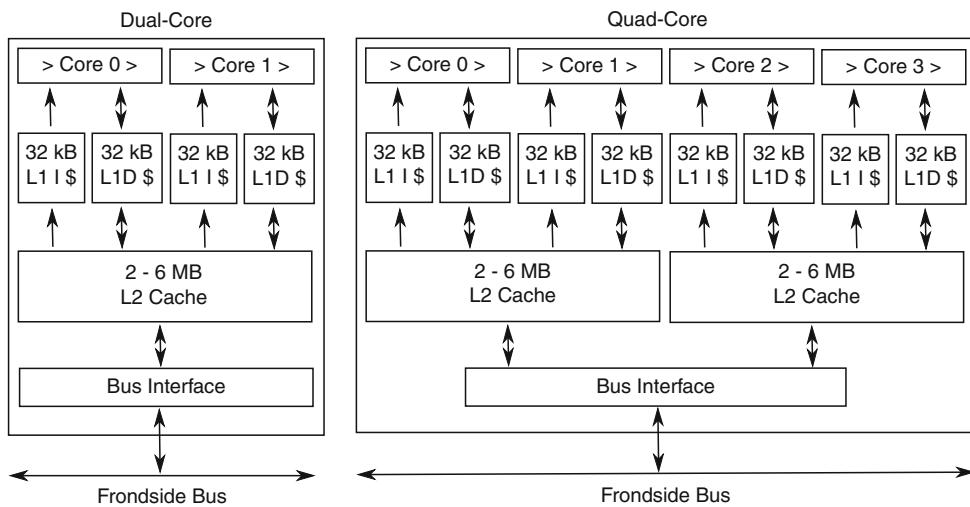
After decoding into microoperations, mapping of ISA registers to physical registers is done (register renaming/allocation). This enables reduction of data dependencies (such as write-after-write hazards), and allows for more ILP to be exploited by out-of-order execution. The following stage is the reorder buffer, where dependencies are checked. Here, operations are allowed to overtake each other. The next stage, the scheduler, is able to dispatch up to six microoperations per cycle. It feeds three ALUs and another three execution units responsible for memory access. All ALUs can handle 128-bit vector operations at a throughput of one result per clock cycle. At most, this allows eight floating point operations to be executed in one cycle (via SSE instructions). The results of the ALUs lead to notifications back to the reorder buffer, allowing waiting operations then to be dispatched by the scheduler according to previously calculated data dependencies. All loads and stores to memory first pass a store buffer before the memory access operations are retired. The store buffer allows loads to directly use the result of store instructions, and allows multiple stores to nearside memory cells to be combined, thus increasing the effective bandwidth to the L1 data cache. Finally, the retirement stage is able to complete the execution of up to four microoperations per cycle, by reordering their retirement in program order of original instructions.

## Cache Subsystem

Figure 4 shows memory hierarchies used in implementations of the Core Microarchitecture, including the cache subsystem. The Core Microarchitecture has separate L1 instruction and data caches, each with a size of 32 kB. Separate L1 caches for different uses is important to reduce access conflicts, but also allows to tune the caches for the different access behavior. The unified L2 cache is 2 up to 6 MB large, serving L1 caches of two cores. With a quad-core design, this means that there are two L2 caches, as can be seen in the figure. For best exploitation, each L1 as well as L2 cache has its own prefetch units, with all load and store operations between cores and L1s done with 128-bit width. Both the L1 data as well as the L2 cache use a write-back scheme, that is, writes done are kept in the fastest caches as long as possible, reducing store transactions.

L1 data caches employ two kinds of prefetchers each: one to detect streaming accesses, the other to detect stride accesses by single instructions, that is, access behavior is detected based on instruction address. Similarly, the L2 caches have stream prefetchers. In addition, the x86 ISA (part of instructions added with SSE) allow for software prefetching, that is, loading instructions without effect aside from loading data into caches. Together, hardware and software prefetching can prove quite effective to reduce cache misses. The memory system of multiple processors with the Core architecture uses a bus to access memory (the so-called, Intel-proprietary *Frontside Bus*). On the bus, a chipset is attached with memory controllers talking to the memory modules. Thus, all cores in the system share the bus. In server systems, more expensive chipsets allow to have point-to-point connections to each processor chip. This way, multiple processors can access memory at the same time.

To assure cache coherence, that is, to make sure that processor cores do not use outdated data because of modified data in other caches, a cache coherency protocol has to be used when there are multiple caches at the same level. This is done on the one side by observing bus transactions done by other processors (“snooping” the bypassing traffic), but because of write-back, also by asking other cores at load time whether they have modified copies. On the level of a multiprocessor system with one Frontside bus, the so-called MESI-protocol is used



**Intel Core Microarchitecture, x86 Processor Family. Fig. 4** Memory hierarchies used in implementations of the Core Microarchitecture

(the four characters represent the four possible states for a cache line), which belongs to the class of snooping protocols. However, in servers with point-to-point connections to the chipset, all the bus transactions usually need to be broadcast to allow for snooping, thus reducing the benefit. These systems employ so-called *Snoop Filters*, which allows to reduce this coherency traffic in given situations.

## New Developments

The next micro-architecture from Intel is the Nehalem architecture. Its main changes are a large L3 cache shared among all four or eight cores on a processor chip. Instead, the unified L2 cache is now private for each core, and reduced in size to 256 kB. Further, all cores (with their L1/L2) can now be driven at different clock rates, and put to sleep independently. The so-called Turbo mode allows to even raise the clock rates of single cores when temperature allows. In effect, this leads to faster program execution when only a few cores are actually used on a chip. Another significant change is the move of memory controllers into the chip, allowing to directly attach main memory. Chips are coupled via a high-speed, point-to-point interconnect, called QPI (Quick-Path Interface). A consequence for multi-chip/socket systems is that distance to memory is different, depending on which core accesses the memory. Up to

the Core Microarchitecture, Intel multiprocessor systems used a Frontside bus, resulting in uniform memory access (UMA). With the introduction of Nehalem, multiprocessor systems now use nonuniform memory access (NUMA), which AMD systems had employed for quite some time. While local memory accesses are faster, remote accesses can become an obstacle to program performance, thus needing new optimization techniques in programs and the operation system. Also, the cache coherence protocol has to be adapted. Intel enhanced MESI by a fifth state (F), which allows to directly forward modified cache lines to other chips without writing to memory (this is similar to the fifth state of the MOESI protocol used by AMD).

For some time, graphics processors have improved quite significantly in computation power, driven by 3D games. Recently, this power is made available to programmers via GPGPU interfaces (General Purpose Graphics Processing Unit). For their purposes, graphics processes contain huge arrays of ALUs, driven in clusters by control units. They can be seen as using an extreme version of the vector extensions of standard processors. Intel recently announced a processor with a large number of simplified cores, but with very wide vector units (64 byte), named Larrabee. Its advantage is that each core is x86-compatible. However, as a core is probably a lot slower than a current Nehalem core, the chip is built as graphics processor. With more

transistor budget available, it can be expected that such simple, but specific cores with wider vector units are mixed with more complex ones, resulting in heterogeneous multicore processors. This also would mean that graphics units are moved into the same chip as the main processor cores, similar to the x87 floating point coprocessor in i486 times.

## Related Entries

- ▶ Cache Coherence
- ▶ Instruction-Level Parallelism
- ▶ Memory Wall
- ▶ Pipelining
- ▶ Power Wall
- ▶ Shared-Memory Multiprocessors
- ▶ Superscalar Processors
- ▶ Vector Extensions, Instruction-Set Architecture (ISA)

## Bibliographic Notes and Further Reading

A good introductory book on computer architecture is [1]. A detailed, step-by-step introduction into computer architecture and microprocessor design, using the MIPS ISA, is given in [2]. From the same authors, [3] provides more in-depth reading with practical evaluations of concepts. A good coverage of how to build large parallel machines, again with evaluation of concepts with practical applications, is provided in [4]. For coverage of current microprocessor trends, see [5].

The best reference about the Intel® 64 and IA-32 Instruction Set Architectures, including overviews of the micro-architectures, is the Software Developer's Manual together with the Optimization Reference Manual from Intel Corporation [6, 7], both available for free download from their web site. For a more introductory text on microprocessors, especially also from Intel, see [8].

On the Internet, a lot of technical information for x86 processors can be found on [9], and articles about recent chips, for example, on [10].

## Bibliography

1. Tanenbaum AS (2005) Structured computer organization, 5th Edn. Prentice Hall, Upper Saddle River, NJ, USA
2. Hennessy JL, Patterson DA (2006) Computer architecture – a quantitative approach, 4th edn. Morgan Kaufmann, San Francisco, CA

3. Patterson DA, Hennessy JL (2008) Computer organization and design: the hardware/software interface, 4th edn. Morgan Kaufmann, Burlington, MA, USA
4. Culler D, Singh JP, Gupta A (1997) Parallel computer architecture: a Hardware/Software Approach. Morgan Kaufmann, San Francisco, CA, USA
5. Microprocessor report. In-stat, reed business information. Reed Elsevier. <http://www.mdonline.com>
6. Intel® 64 and IA-32 architectures software developer's manual, October 2009
7. Intel® 64 and IA-32 architectures optimization reference manual, October 2009
8. Stokes J (2007) Inside the machine: an illustrated introduction to microprocessors and computer architecture. No Starch Press, Inc., San Francisco, CA, USA
9. <http://www.sandpile.org>
10. <http://arstechnica.com>

## Intel® Parallel Inspector

PAUL PETERSEN

Intel Corporation, Champaign, IL, USA

### Synonyms

Data race detection; Deadlock detection; Illegal memory access

### Definition

Intel® Parallel Inspector is a product that integrates with the Microsoft Parallel Studio to provide support to software developers in analyzing parallel software for problems relating to the correct execution of an application.

### Discussion

#### Introduction

Intel® Parallel Inspector is designed as a component of the Intel® Parallel Studio, a suite of software development tools to aid in the creation of parallel applications. Software development involves many different aspects from the creation of specifications, to writing the implementation code, testing, debugging, and tuning the resulting application. All of these aspects are true when writing any kind of software, with additional challenges arising when writing software which exploits concurrency via threading or parallelism. Parallel Inspector serves as an intelligent assistant to notify the developer about defects in the parallel application that could

cause incorrect behavior. In particular, Parallel Inspector focuses on a few core problems. These are misuse of synchronization which can result in data races or deadlocks, and incorrect memory accesses resulting in illegal or invalid memory references in parallel applications.

## Requirements for Parallel Inspector

Intel® Parallel Inspector works with the Microsoft Visual Studio 2005 or 2008 development environment on the Windows XP, Vista, or Windows 7 operating system. It can either be installed by itself or as part of the Parallel Studio bundle.

Intel® Parallel Inspector operates by analyzing the dynamic behavior of a native code application via dynamic instrumentation. The typical application is written using the C or C++ language. It is preferable to analyze applications that are built with debugging symbols to map the dynamic behavior to the source code. It is also helpful to tailor the inputs driving the test cases to be small and focused on the portions of the applications that the developer wishes to analyze. Using the smallest inputs possible, avoid redundant work during the analysis where similar memory access patterns are repeatedly verified.

## Defects in Parallel Software

The defining characteristic of threaded software is the potential for concurrent access to memory. To be correct, these memory accesses need to follow certain rules:

1. When reading or writing, the memory address must be valid.
2. When reading, contents of the memory address must be initialized.
3. The memory access must be properly synchronized.

Each of these rules is examined in turn to see how Parallel Inspector can assist the software developer.

### Invalid Memory Accesses

First, consider if a memory address is valid. In the memory address space for an application, different parts of the address space are allocated for various purposes. For example, part is allocated for the instructions comprising the code of the application. Part is/Parts are allocated as the stack(s) for the threads. Part is/Parts are allocated for the heap(s) used by the application. Part is/Parts are reserved by the operating system for its

own usage. Typically, other regions of the application's address space are unallocated, and memory references to the unallocated regions of the address space are considered to be illegal. Similarly, requests to write into read-only pages are also illegal (such as attempting to write to the code pages).

### Uninitialized Memory Accesses

Second, consider if it is legal to access a memory address. Writes to any valid memory address are legal even if the memory is uninitialized. However, reads from uninitialized memory addresses are illegal as the value read may be nondeterministic and cause the behavior of the application to be undefined. A specific challenge for uninitialized memory access detection is the reading from uninitialized memory of a value that is ultimately not used. For example, a word containing 4 bytes might only have its lower byte initialized. A common programming pattern would be to load all 4 bytes via a word load instruction, and then mask off all but the lowest byte. By throwing away the high bytes in the mask operation the uninitialized memory cannot affect the behavior of the application, even though it was physically read.

Similarly the same concept can apply to larger structures. Consider a structure containing a "char" variable followed by a "double" variable

```
struct Ex1 {
 char c;
 double d;
};
```

Alignment rules may require that field "d" be aligned on an 8 byte boundary. This forces 7 padding bytes to be inserted after "c", yielding a structure with 16 bytes.

The difficulty arises from structure assignment.

```
struct Ex1 x, y;
y.c = 'a';
y.d = 3.14;
...
x = y;
...
```

Often structure assignment is implemented by copying all 16 bytes, of which 7 of these bytes are uninitialized. Often, heuristics can filter these memory copy operations and avoid issue false-positive reports.

## Synchronization Usage

The last category of problems relates to synchronization of memory accesses.

The best case is where memory is not shared between threads, and thus no synchronization is needed. However, often this is not possible as the shared memory locations are needed for communication of data between threads.

Three cases arise when considering synchronization.

1. Incorrect lock usage can cause deadlock
2. Synchronization is not used, or not used consistently
3. Synchronization is used, but at an incorrect granularity

## Deadlock

Incorrect lock usage can expose the potential for a deadlock at runtime. A deadlock occurs when two or more threads each owns a lock that another thread is attempting to acquire, and will not release its lock until it acquires its next requested lock. This situation forms a cycle in the lock ownership/acquisition graph, as illustrated in the following example:

Thread 1:

```
AcquireLock(A)
AcquireLock(B)
```

...

Thread 2:

```
AcquireLock(B)
AcquireLock(A)
```

...

The lock acquisition graph for Thread 1 is  $A \rightarrow B$ . For Thread 2 it is  $B \rightarrow A$ . Merging these lock graphs graph results in the cycle  $A \rightarrow B \rightarrow A$ , which indicates the potential for deadlock in the application.

## Data Races

The second case is that synchronization is not used, or that it is used inconsistently. An example of this situation is:

Thread 1:

```
...
AcquireLock(A)
X = X + F(1)
ReleaseLock(A)
```

Thread 2:

```
...
X = X + F(1)
```

Thread 3:

```
...
AcquireLock(B)
X = X + F(1)
ReleaseLock(B)
```

In this example, the first update to  $X$  is protected by the lock  $A$ , but the second update to  $X$  is not protected by any lock. Therefore, it is possible that the operations of Thread 1 and Thread 2 interleave in a way that alters the applications behavior. Similarly, Thread 3 has the problem that it is using a different lock than Thread 1, which means that mutual exclusion is not enforced between any of Threads 1, 2, or 3.

## Atomicity Violations

The third case is when synchronization is used at the wrong granularity. In a serial program, all operations between I/O statements appear atomic to an outside observer, since the I/O statements are the only mechanism by which to interact or observe the execution of a serial program. In a multi-threaded program this is not true. The multiple threads that may exist can potentially interact or observe the state of other threads in the application at any point during the execution. Because of this, developers need to be aware of the concept of atomicity. Consider a statement such as:

$X = F(X)$

Let  $X$  be an arbitrary object, and  $F$  be an arbitrary function applied to  $X$ . During this computation, various parts of  $X$  may change to intermediate states that are then used to compute the final value of  $X$ . If no other entity can observe or modify any of these intermediate states, then  $F()$  is an atomic function, and the atomicity of the transformation  $X = F(X)$  has been preserved.

Consider the following example:

```
struct Node {
 struct Node *next;
 double data;
};

Node *head;
int length;
```

```

void Initialize()
{
 head = 0;
 length = 0;
}
void Add(Node *p)
{
 p->next = head;
 head = p;
 length = length + 1;
}
Node *Remove()
{
 Node *p = head;
 length = length - 1;
 head = p->next;
 return p;
}
double GetAverage()
{
 Node *p = head;
 double answer = 0;
 for (int I = 0; I < length; ++I) {
 answer += p->data;
 p = p->next;
 }
 return (sum / length);
}

```

Assume that the application requires that the invariant that “length” is exactly the number of elements in the linked list. The GetAverage() function depends on this to compute the correct answer. Both the Add() and Remove() functions change the value of “length.” If either function is called in parallel with the GetAverage() function, then an inconsistent state of the data structure may be observed.

To fix this, the example must enforce atomicity to maintain the invariant that the value of length is exactly the same as the number of nodes in the list.

Intel® Parallel Inspector is not designed to detect atomicity violations in an otherwise well-synchronized parallel program. Instead, Parallel Inspector detects the lack of synchronization. To reason about atomicity with Parallel Inspector, the recommended approach is to remove synchronization suspected to be at the wrong granularity, find the set of data races this

creates, and then reason about your algorithm to determine if the algorithm is correct if the atomicity region is large enough to enclose all affected memory accesses.

## Intel® Parallel Inspector Defect Model

When Parallel Inspector detects a defect in the parallel software, it records this fact. The record contains the static and dynamic location of the occurrence of the defect as well as supporting information that may be necessary to help the developer understand how the defect occurred or what may be necessary to fix the defect.

Many detected defects may be related to the same root cause in the application. For example, if the developer forgot to synchronize the update to a particular object, then that update may cause a data race with all other uses of that object.

Similarly, if the same object is referenced by several threads, then this set of data races must be fixed by acquiring the same synchronization object to cover all usages of that object.

To aid the user in organizing the defects, Parallel Inspector defines three organizational concepts: an Observation, a Problem, and a Problem Set.

A defect in the application is composed of Observations. An Observation is defined as a fact about an action performed by the application. This may be a memory Read action, a memory Write action, a Lock Acquire action, a Memory allocation action, or possibly a Memory deallocation action. Each Observation has a representative call stack, and specific location (instruction) in the application which triggered this action. Some Observations are fundamental to a defect. For example, in a data-race defect, the memory accesses (either Read or Write) trigger the existence of this defect. Other Observations are incidental. For example, it may be useful to know where the application allocated the object involved in the data race, but typically it is not the allocation which triggered the detection of the defect.

A Problem contains one or more Observations related to the same defect. A Problem also merges all of the dynamic instances of a defect into a single instance of the defect. For example, a utility function may access a shared variable that is unsynchronized, which causes

a data race. This function may be called from many different locations in the application, representing many different dynamic occurrences. This first level of defect organization focuses the user on the source code of the application and collapses all of these dynamic instances into a single Problem.

The second level of organization performed by Parallel Inspector is to organize Problems into Problem Sets. Two Problems are grouped into the same Problem Set when they share a significant Observation. For example, consider Problem 1 that writes to an object at location S1, and reads the object by a different thread at location S2. If another Problem 2 exists that also writes the object at location S1, but reads the object by some other thread at location S3, then Problem 1 and 2 are placed into the same Problem Set. What this indicates is that the pair-wise relationship of a data race between two threads should be extended to a group-wise relationship between a set of threads and a set of memory location. This implies that since S1 is related to S2 and S1 and also related to S3, then it is quite likely that S2 is also related to S3, and needs to be considered when developing the solution to fix the data race.

These three levels of organization of application defects in Parallel Inspector allow the developer to interact with the reported information in appropriate ways.

For example, a “Detail” view is provided that organizes the defects according to the Observations. This allows quick access to any Problem or Problem set based on characteristics of any Observation contained in a Problem Set. For example, the developer may want to know which data races are connected with an object allocated at a specific location in the program. Or, the developer may be concerned about which Observations are reported to be involved with a specific binary module (.DLL or .EXE file) or with a specific function or file. If the developer recently edited these functions or files, or recompiled this module the defect may relate to a modification of the application recently performed by the developer.

## Implementation

Intel® Parallel Inspector is designed to observe an application as it executes. This has the benefit that the tools have accurate information about the state of

the application as it runs, but has the disadvantage of being limited to the subset of the application behavior covered by the workload used during testing. Since Parallel Inspector relies on dynamic program behavior to detect defects, it is primarily a diagnostic tool aiding to pinpoint the root cause of a defect.

Parallel Inspector uses the Pin Dynamic Instrumentation system [6]. This system is a set of APIs which translate and instrument the native assembly language instructions. Pin operates on a trace of instructions translating them with a native-code to native-code JIT compiler, and inserting instrumentation which drives an online analysis engine. This instrumentation observes modification to memory (load or store actions), as well as memory allocation and deallocation, threading and synchronization actions.

Two different analysis engines are included in Parallel Inspector. The Memory Inspection analysis engine maintains a bitmap describing the valid and initialized state of the address space. Allocation actions create valid address regions; write actions create initialized address regions. Subsequent memory accesses are checked against this bitmap to determine if they are invalid or uninitialized. The memory allocation and deallocation operations are matches to help determine which memory regions have never been deallocated, and further analysis is done to determine which memory regions are inaccessible (i.e., a memory leak).

The Thread Inspection analysis engine employs a “happens-before” [4] methodology for determining where a data race may exist in the application. The “happens-before” relationship is computed by observing the synchronization performed by the application. Two threads are ordered by a “happens-before” arc when the second thread’s execution is conditional on the first thread’s execution. For example, the first thread releases a lock, which the second thread subsequently acquires. In this execution of the program, any memory reference that occurs before the first thread releases the lock must “happen-before” any memory reference that occurs after the second thread acquires the lock. Using this methodology, a data race occurs whenever the same memory location is accessed (at least one write) by two threads, and there does not exist a “happens-before” ordering relationship between these two memory accesses.

A special class of memory access is flagged by the Thread Inspector as deserving additional attention. This class of memory accesses is known as “cross thread stack accesses.” When a thread accesses memory belonging to the stack of another thread, typically it does this by dereferencing a pointer published by the thread that owns the stack. The challenge is not only enforcing mutual exclusion to stack locations when they are modified, but the existence of the stack location needs to be maintained during while threads access that stack location. In a strict parent–child relationship, where the parent publishes one of its stack locations to its children, and then waits for the children to terminate, then it is guaranteed that the lifetime of the stack location encloses the lifetime of the remote thread which has obtained a reference to that stack. In any other situation, the developer must be vigilant in understanding the memory reference pattern to determine if it is a legal parallel programming pattern.

Additionally, Thread Inspector determines the existence of actual and potential deadlocks caused by the synchronization executed by the application. A lock hierarchy acquisition graph is inferred from the lock acquire and release actions observed in an applications execution. The analysis engine checks for cycles in this graph to detect if a deadlock exists.

## Related Entries

- [Intel® Parallel Studio](#)
- [Intel® Thread Profiler](#)
- [Intel® Threading Building Blocks \(TBB\)](#)

## Bibliographic Notes and Further Reading

The Parallel Inspector is a product in the Intel® Parallel Studio. This product is the evolution of ongoing work, which was initially released from Kuck & Associates, Inc (KAI) as the product Assure, and then subsequently refined and released as the Intel® Thread Checker after the acquisition of KAI by Intel in 2000.

The main evolution of this technology has been to move from source-to-source instrumentation, to static binary instrumentation, to finally the Pin-based dynamic binary instrumentation that is used in the Parallel Inspector.

## Bibliography

1. Gabb H, Kakulavarapu P (eds) (2005) Developing multithreaded applications: a platform consistent approach, v2.0 Feb 2005
2. Wang L, Xu X (2007) Parallel software development with Intel threading analysis tools. *Intel Tech J* II(4):287–297
3. Banerjee U, Bliss B, Ma Z, Petersen P (2006) Unraveling data race detection in the Intel® thread checker. Presented at the first workshop on Software Tools for Multi-core Systems (STMCS), in conjunction with IEEE/ACM international symposium on code generation and optimization (CGO), Manhattan, New York, 26 March 2006
4. Banerjee U, Bliss B, Ma Z, Petersen P (2006) A theory of race detection. In: International symposium on software testing and analysis, Proceedings of the 2006 workshop on parallel and distributed systems: testing and debugging, Portland, pp 69–78
5. Intel® Parallel Inspector Help (2009) [http://software.intel.com/sites/products/documentation/studio/inspector/en-us/2009/ug\\_docs/index.htm](http://software.intel.com/sites/products/documentation/studio/inspector/en-us/2009/ug_docs/index.htm)
6. Hazelwood K, Lueck G, Cohn R (2009) Scalable support for multi-threaded applications on dynamic binary instrumentation systems. In: Proceedings of the 2009 international symposium on memory management ISMM '09, Dublin, Ireland, 19–20 June 2009. ACM, New York, pp 20–29. DOI= <http://doi.acm.org/10.1145/1542431.1542435>

## Intel® Parallel Studio

PAUL PETERSEN

Intel Corporation, Champaign, IL, USA

## Definition

Intel® Parallel Studio is a product that integrates with Microsoft\* Visual Studio IDE that extends the IDE with facilities to assist in the creation of software for multi-core computers.

## Introduction

Intel® Parallel Studio is a product that encompasses the life cycle of parallel software development. If you are starting from a serial application, the tools provided can generate high performance executables, inform you of where the hotspots exist in your algorithms, and find memory leaks and invalid memory access that may be corrupting your application’s execution. Continuing in the parallel software life cycle, tools exist that assist the user in determining the most appropriate places to introduce parallelism. Once the application

is enabled for parallel execution, the remaining tools in Parallel Studio come into play. Problems unique to parallel programming such as deadlocks, data-races, synchronization overhead, or insufficient concurrency are easily exposed through the parallel analysis tools.

## Requirements for Parallel Studio

Intel® Parallel Studio works with the Microsoft Visual Studio 2005, 2008, or 2010 development environment on the Windows XP, Vista, or Windows 7 operating system.

## Background

*Intel Parallel Studio* is a software development product created by Intel that uses the Microsoft Visual Studio Integrated Development Environment. Its purpose is to enable the development of programs for parallel computing. Parallel programming enables software programs to take advantage of multi-core processors from Intel and other processor vendors.

Parallel Studio consists of four component parts, each of which has a number of features:

- Parallel Composer consists of a high performance C++ compiler, a number of performance libraries (Integrated Performance Primitives), Threading

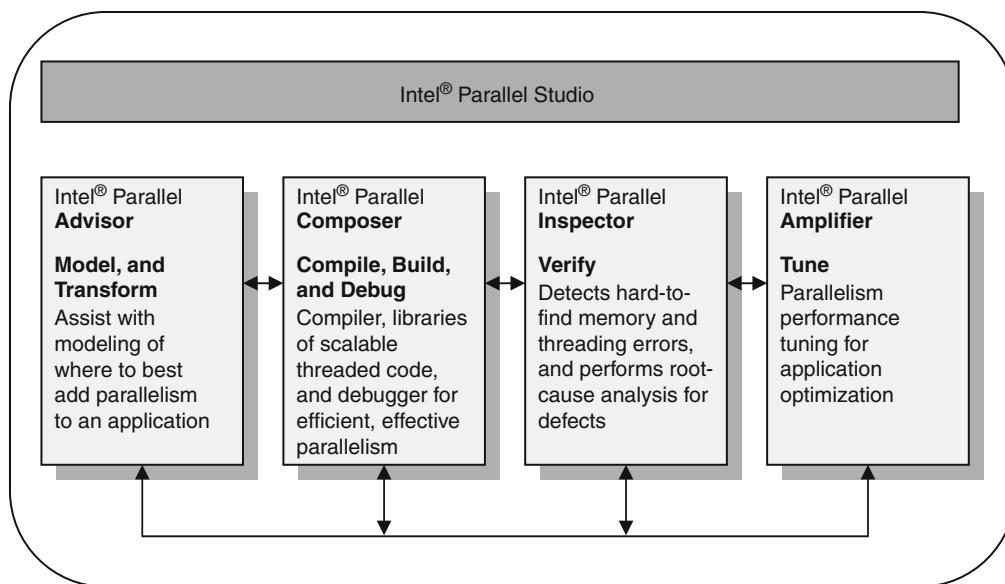
Building Blocks, and an extension to the Visual Studio debugger for parallel programs.

- Parallel Advisor assists the developer in the analysis and modeling of where to best add parallelism to an application.
- Parallel Inspector serves as an intelligent assistant to notify the developer about defects in the parallel application that could cause incorrect behavior. In particular Parallel Inspector focuses on a few core problems. These are misuse of synchronization that can result in data-races or deadlocks, and incorrect memory accesses resulting in illegal or invalid memory references in parallel applications.
- Parallel Amplifier helps the developer to understand the performance implications of their application providing profiling support to determine the hotspots, bottlenecks due to synchronization, and an analysis of the concurrency present in the application.

Parallel Studio is focused on native code development for C/C++ programming.

## Workflow

Parallel software development builds on top of sequential software development in a number of ways. First you must understand the opportunities for



Intel® Parallel Studio. Fig. 1 Intel parallel studio product components

parallelism. Second, you must express the parallelism in the application. Third, you must verify that your expression of parallelism does not have unexpected defects. Finally, you must determine if you are getting the expected benefit from the use of parallelism.

Intel Parallel Studio (Fig. 1) provides the capabilities you need to tackle all four of these areas. Starting with a high performance C/C++ compiler, adding parallelism-specific analysis for opportunities, correctness, and performance gives the developer a head start.

## Related Entries

- ▶ [Intel® Parallel Inspector](#)
- ▶ [Intel® Thread Profiler](#)
- ▶ [Intel® Threading Building Blocks \(TBB\)](#)

## Bibliographic Notes and Further Reading

The Intel® Parallel Studio brings together products to aid the developer in producing parallel programs for Microsoft® Windows.

## Bibliography

1. Intel® Parallel Studio Help (2009) <http://software.intel.com/en-us/articles/intel-parallel-studio-documentation/>

## Intel® Thread Profiler

MARK DEWING, DOUGLAS ARMSTRONG  
Intel Corporation, Champaign, IL, USA

### Definition

The Intel® Thread Profiler is a tool to assist in optimizing multithreaded applications. It shows developers where various performance bottlenecks and impediments to scalability occur in the program.

### Discussion

#### Introduction

Intel® Thread Profiler collects data relevant for multithreaded performance and helps the user understand the cause of various performance problems.

Intel® Thread Profiler can collect data on programs written using OpenMP® or using native threading APIs (Windows® API or POSIX® threads). “\*Other names and brands may be claimed as the property of others”.

### Threading Methodology

Threading for performance is a process with several steps. This process starts with discovering possible parallelism in an existing codebase or algorithm and expressing that parallelism through some threading library or framework (e.g., OpenMP, Intel® Threading Building Blocks, native threading APIs). Next, the threaded code must be debugged and analyzed for correctness with testing and possibly with tool support (e.g., Intel® Parallel Inspector). Then the performance of the code is analyzed and the code tuned. This process is iterative – changes suggested by performance tuning need to be tested for correctness as well.

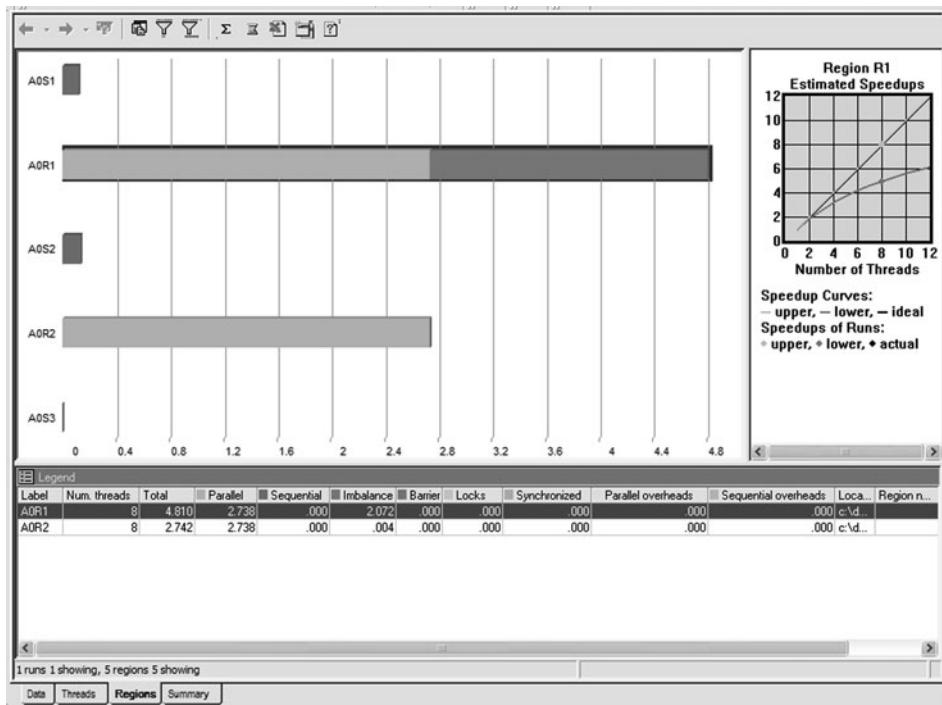
Intel® Thread Profiler can provide assistance in the performance tuning part of the process. A typical usage model is to run the target application under Intel® Thread Profiler and start looking at regions where the concurrency is lower than expected. Next, particular synchronization objects are examined to see which ones are responsible for the most waiting. Finally, the source locations where those objects are used can be examined to understand the root cause of the performance problem.

### Implementation Details

To collect data on OpenMP programs, Intel® Thread Profiler uses a version of the OpenMP runtime library that collects statistics, and writes them to a file. For native threads, binary instrumentation is used to intercept various threading and blocking APIs. A graphical interface integrated into the VTune™ Performance Environment reads the data file and displays it for interactive analysis (VTune is a trademark of Intel Corporation in the USA and other countries).

### Profiling OpenMP

The OpenMP mode of Intel® Thread Profiler displays the amount of time spent in parallel regions and serial regions. It also shows the amount of imbalance in parallel regions, and the amount of overhead due to synchronization and locks. This can be displayed per region, as well as an aggregated summary (Fig. 1).



Intel® Thread Profiler. Fig. 1 OpenMP region view in Thread Profiler

The tabs show a breakdown of the time in categories:

- Summary – time for the whole program
- Regions – time per serial or parallel region
- Threads – time per thread
- Data – grid view of performance data

The screen shot shows the regions view with two parallel regions – one with a large amount of imbalance and one with almost no imbalance.

### Profiling and Tracing Native Threads

When analyzing applications using native threads, different data is collected and displayed to the user, as shown in Fig. 2.

### Timeline Pane

The lower pane in Fig. 2 is the timeline view, where the threads are on the vertical axis and time is on the horizontal. The activity of each thread is shown (either waiting or in a ready-to-run state). Transition lines show connections between threads – where one thread releases a lock and another thread acquires that lock.

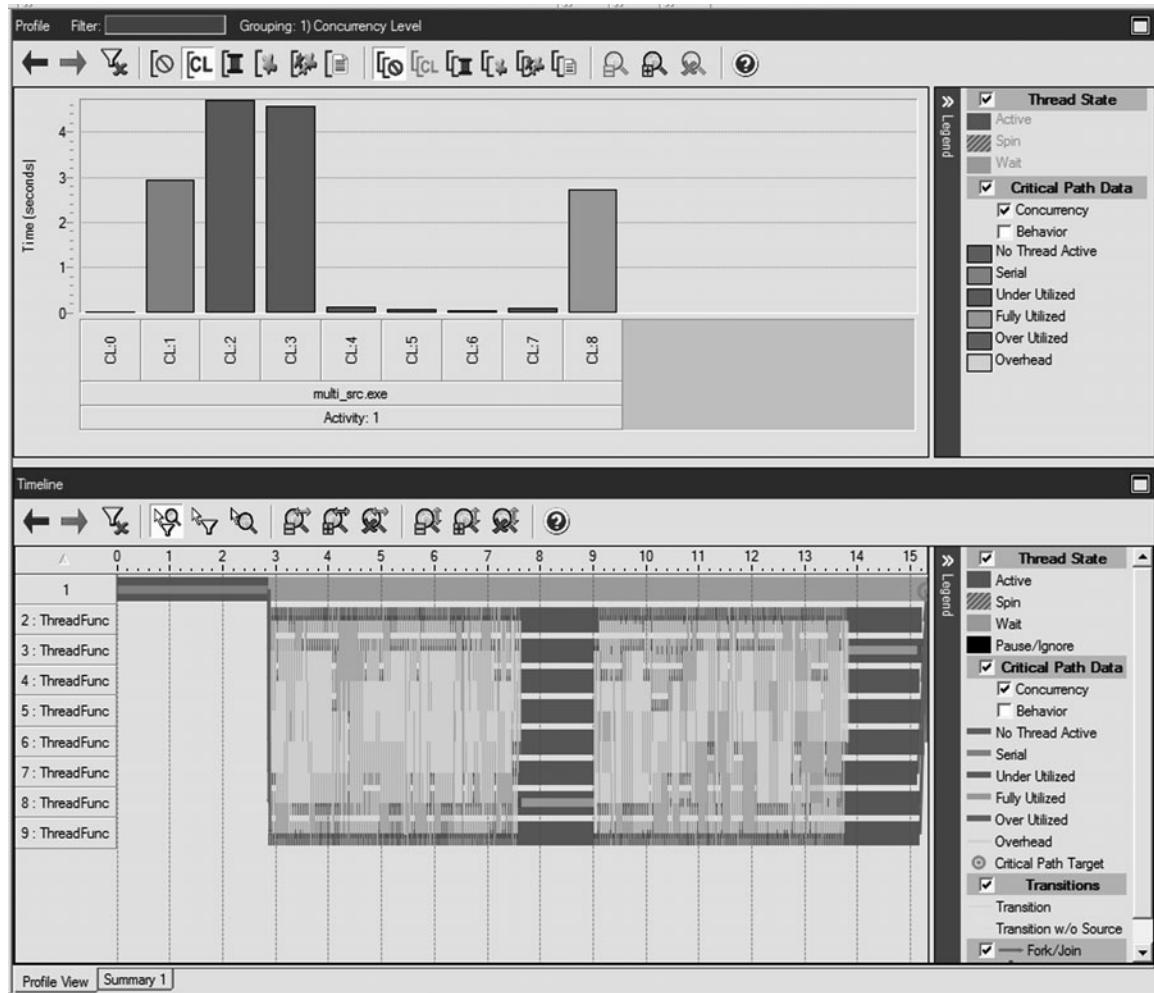
There are several thread states:

- Active – the thread is either running on a core, or is ready to run and is waiting in the run queue.
- Wait – the thread is blocked waiting for some resource to become available (synchronization or I/O).
- Spin – the thread is actively waiting in a synchronization construct. This thread state is only detected in libraries or user codes that have added API calls to inform Thread Profiler of spin waits.

The bars in profile view and line segments in timeline view can also be colored by concurrency and/or behavior.

The concurrency types are:

- No Thread Active – none of the application's threads are active.
- Serial – only a single thread is active.
- Underutilized – the number of active threads is less than the number of cores in the system.
- Fully Utilized – the number of active threads is equal to the number of cores on the machine. This is the ideal case for best utilization of CPU resources.
- Overutilized – there are more active threads than cores on the machine. This can lead to some



Intel® Thread Profiler. Fig. 2 Intel® Thread Profiler display for native threads

performance loss due to excess context switches and thrashing the cache.

The behavior applies to time segments along the critical path

- Critical Path – The thread on the critical path is active, and not holding a resource needed by the next thread on the critical path (compare with “impact time”).
- Blocked on critical path – The thread on the critical path is waiting for an external event – typically a system event (a timer or file I/O) or a user event (mouse or keyboard). (If it were waiting on another thread, that thread would have been on the critical path instead).

- Impact on critical path – The current thread is active, but the next thread on the critical path is waiting. Typically, this means the current thread is holding a lock or other resource while the next thread waits.

### Profile Pane

The upper pane in Fig. 2 is the profile view, which displays wait time and time on the critical path, grouped in various ways – by concurrency, thread, object, object type, or source location. The groupings in more detail are

- None – shows the overall times for the region visible in timeline view
- Concurrency – time a given number of threads are active in the system

- Thread – list of threads created in the process and the amount of time each thread spends active or waiting
- Object Type – waiting caused by objects, grouped by type (all the mutexes in one group, all the critical sections in another, etc)
- Object – waiting caused by individual objects (individual mutexes, critical sections, etc)
- Source Location - time associated with the source location where the wait occurs and the source location where the object is released.

### Source View

The user can navigate to the source code where the wait occurred. The source code where the object is released is also shown.

### Critical Path

The critical path links transitions between threads into a continuous path from the beginning to the end of the program. The critical path is visible in the timeline view, and data aggregated over the critical path is shown in the profile view. If the performance is improved anywhere along the critical path, the program's run time will be reduced. Improving the performance elsewhere is less likely to reduce the program run time (it may affect load or CPU usage, however).

## Tuning Threaded Code

### Imbalance in Parallel Work

Imbalance occurs when one or more threads in a parallel region are idle waiting for other threads in the parallel region to finish working. In the OpenMP

profiler, there is a specific time category for imbalance. In the native threads mode of Intel® Thread Profiler, imbalance can be seen visually in timeline view, or it manifests as underutilized time in the profile view.

The two regions in Fig. 1 are duplicates of the same code and differ only in how the work is scheduled on different cores. The amount of work varies per iteration, and in the first loop, work is assigned statically to different cores. That is, each core executes the same number of loop iterations. The second loop assigns the work dynamically to each core, and results in a much better load balance.

### “Hidden” Synchronization

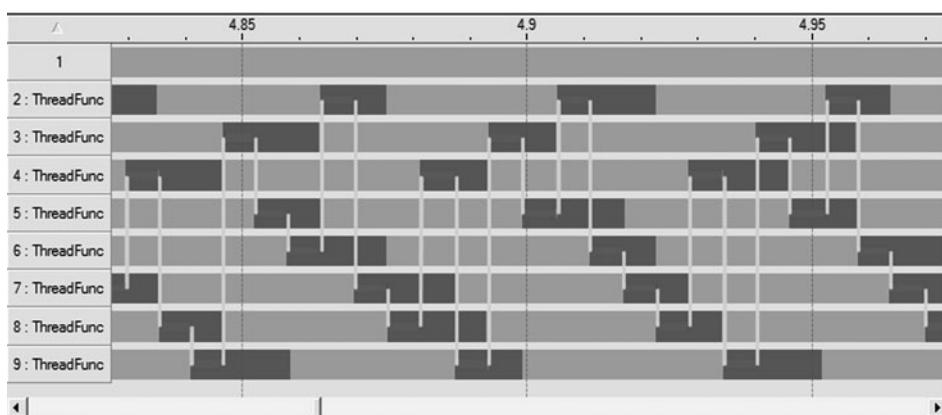
Often the developer is unaware of the parallel behavior of various subsystems and what locks they may hold. Intel® Thread Profiler reveals where these subsystems may be serializing or reducing the concurrency of the higher-level code.

### Excessive Synchronization

If a lock is held and released frequently, the threads may spend an excessive amount of time locking overhead, reducing the amount of time available for actual work.

### Excess Hold Times and Lock Convoys

Threads may hold locks longer than necessary, and this will manifest in underutilized time. This may also lead to a lock convoy, which is easily spotted in the timeline view (Fig. 3).



Intel® Thread Profiler. Fig. 3 Lock convoy in timeline view

### Hot Locks

A hot lock will show up in the object grouping in the profile view – the hot lock will accrue a large amount of waiting time. Further grouping by source location can pinpoint where and how the lock is being used.

### Release History

Intel® Thread Profiler 1.0 was first released in 2003 in conjunction with Intel® Thread Checker for Windows. This version of Intel® Thread Profiler only included support for OpenMP.

Intel® Thread Profiler 2.0 was released in 2004 and added support for native threads. Intel® Thread Profiler 3.0 was released in 2006 and featured an enhanced user interface.

Intel® VTune™ Amplifier XE 2011 was released in 2010, replacing the functionality of Intel® Thread Profiler with a tighter integration of Intel® Thread Profiler and Intel® VTune™ Analyzer capabilities.

### Bibliography

1. Gabb H, Kakulavarapu P (eds) (2005) Developing multithreaded applications: a platform consistent approach, v 2.0. Intel Corporation, Santa Clara, CA
2. Wang L, Xu X (2007) Parallel software development with Intel Threading Analysis tools. Intel Technical Journal 11(4):287–297
3. Intel® Thread Profiler Help, v 3.1 (2007) Intel Corporation, Santa Clara, CA

## Intel® Threading Building Blocks (TBB)

ARCH D. ROBISON

Intel Corporation, Champaign, IL, USA

### Synonyms

[Intel® TBB](#)

### Definition

Intel® Threading Building Blocks (Intel® TBB) is a C++ library for shared-memory parallel programming.

### Discussion

#### Introduction

Intel® Threading Building Blocks (Intel® TBB) is an open-source C++ library for parallel programming of

multi-core processors. It is notable as a commercially supported library for parallel programming targeting mainstream developers. It has won two “Jolt Productivity Awards.” The library has evolved since its initial release in 2006. This entry describes Intel® TBB version 3.0 released in 2010.

The library supports both high-level and low-level specifications of parallel control flow. At the high level are parallel “algorithms,” similar in spirit to ISO C++’s `<algorithm>` library. These range from simple parallel control structures such as parallel iteration to more complex subjects such as parallel sorting. The algorithms may be nested. At a slightly lower level the programmer can specify tasks, which are less structured than the parallel algorithmic operations. For both of these, the run-time deals with mapping the algorithms or tasks onto hardware threads.

For low-level parallel programming, the library provides portable abstractions for threads, mutexes, condition variables, and atomic operations. Since memory allocators are often a significant bottleneck in programs, the library provides a scalable memory allocator.

A key point is most parallelism in Intel® TBB is optional, never mandatory. The programmer specifies *potential* parallelism. The run-time dynamically decides how much potential parallelism becomes real parallelism. For example, the run-time creates no parallelism if only a single thread is available. Consequently, the programmer must not depend on actions running concurrently, otherwise deadlock may ensue. The only mandatory concurrency in Intel® TBB is its implementation of the C++0x class `std::thread`, which is a thin wrapper around an OS thread.

Intel® TBB also provides support for organizing data. It has concurrent containers that can be safely and efficiently operated on by multiple threads. Its abstractions for thread-local storage permit threads to operate on their own private portion of data, while also providing a global view of all of the data.

Intel® TBB supports parallel programming exclusively via library mechanisms, without language extensions. Related approaches such as Cilk (see ►Cilk) and OpenMP (see ►OpenMP) rely on language extensions, which permit more concise syntax. However, the extensions require compiler support. Hence the library approach has an advantage of

not limiting what compiler is used. C++0x lambda functions simplify syntactic issues of the library approach.

## Terminology

The notation **[a,b)** denotes a half-open interval that includes **a** and excludes **b**.

A C++ functor is an object that defines **operator()** as a member. Some examples in this entry create functors via C++0x lambda expressions. The lambda expressions used in this entry start with **[&]**, **[=]**, or **[]**. Below is an example.

```
template<typename F, typename X >
void Apply(F f, X x) {f(x);}
int Foo() {
 int b = 3;
 Apply([&] (int x) {b+=x}, 4);
 return b;
}
```

The lambda expression **[&] (int x) {b+=x;}** returns a function object. The **[&]** specifies that local variables (such as **b**) are captured by reference. Thus, the call to **Apply** updates the variable **b** declared in **Foo**, and **Foo** returns 7.

Evaluation of the lambda expression is similar to invoking the constructor **Functor1(a)**, where class **Functor1** is defined as:

```
class Functor1 {
 B& b_ref;
public:
 void operator()(X x) {b_ref+=x;}
 Functor1(B& b): b_ref(b) {}
};
```

Replacing **[&]** with **[=]** causes local variables to be captured by value. **[]** disallows any capture. A lambda expression can omit the argument list if there are no arguments. If the body of the lambda returns a value, the return type is declared after the parameter list, like this:

```
[](int i)->float {float x=1.0/i; return x;}
```

The return type can be omitted if the body is only a return statement.

Some examples use the C++0x shorthand **auto** for declaring local variables of inferred type. For example

```
auto x = y;
```

declares **x** to be of the same type as **y**.

## Parallel Algorithms

Most of the parallel algorithms in Intel® TBB are C++ templates that act as parallel control structures. These structures compose functors in a particular parallel pattern. The simplest pattern is **parallel\_invoke(f,g)**, which executes functors **f** and **g**, possibly in parallel if resources permit. Extended variants allow more functor arguments.

Parallel loops (See ►Loops, Parallel) are supported by several templates. The basic form **parallel\_for(range, func)** applies functor **func** over an iteration space defined by **range**. For example, the following expression computes **a[i]+=b[i]** for **i** in the half-open interval **[0,n)**:

```
parallel_for(blocked_range<int>(0,n),
[&](blocked_range<int> r) {
 for(int i=r.begin() il=r.end(); ++i)
 a[i] += b[i];
});
```

The **range** argument **blocked\_range<int>(0,n)** describes a one-dimensional iteration space **[0,n)** over type **int**. The **parallel\_for** partitions that space into subranges, and applies the functor to each subrange, in parallel if resources permit.

The **parallel\_for** template operates over any recursively divisible range. Table 1 lists the signatures required of such a range. For a range type **R**, the initialization **R r2(r1,split())** splits **r1** into two pieces. One piece is **r2**. The other piece is **r1**. For instance, if **r1** is a **blocked range** initially representing **[0,8)**, then after **R r2(r1,split())** executes, **r1=[0,4)** and **r2=[4,8)**. The argument **split()** is a dummy argument that distinguishes a splitting constructor from a copy constructor.

**Intel® Threading Building Blocks (TBB). Table 1** Signatures that a recursively divisible range must support

| Signature                | Semantics                               |
|--------------------------|-----------------------------------------|
| <b>R(const R&amp;);</b>  | Copy constructor                        |
| <b>~R();</b>             | Destructor                              |
| <b>R.empty();</b>        | True if range is empty; false otherwise |
| <b>R.is_divisible();</b> | True range can be split into two parts  |
| <b>R(R&amp;, split);</b> | Splitting constructor                   |

Intel® TBB provides **blocked\_range**, **blocked\_range2d**, and **blocked\_range3d** respectively for one-, two-, and three-dimensional ranges. Programmers may define their own ranges. An interesting implementation technique that demonstrates the power of recursive ranges is the internal implementation of template **parallel\_sort(first,last,compare)**. It uses **parallel\_for** over a special range that represents the key sequence. The **parallel\_for** recursively subdivides the key sequence until **is\_divisible()** returns false. Each subdivision invokes the range's splitting constructor, which is defined to do a partitioning step of quicksort. The functor argument to the **parallel\_for** deals with the base case of the recursion. It serially sorts key sequences that are not subdivided. Hence **parallel\_for** sometimes goes far beyond iteration by expressing a parallel divide-and-conquer pattern.

Intel® TBB has a shorthand form for the common case of a one-dimensional space of integral type. The call **parallel\_for(lower,upper,step,func)** is a parallel equivalent of:

```
for(auto i=lower; i<upper; i+=step) func(i);
```

The template **parallel\_do(first,last,func)** has two cases. The first case acts as a parallel variant of **std::for\_each**, and performs the parallel equivalent of:

```
for(auto first i=first; i!=last; ++i) func(*i);
```

The second case acts as a parallel while loop over a work list to be exhausted, but which may have items added to it during the work. This case is distinguished by **func** taking two arguments. The second argument must be of type **parallel\_do\_feeder<T>**, which has a method **add** that feeds another item to the work list. The initial work list is **[first,last)**. For example, let the following structure represent a node in a tree:

```
struct Node {
 int data;
 Node* left;
 Node* right;
};
```

The following code walks such a tree with root node **Root**, adding 1 to each field data.

```
parallel_do(& Root, & Root+1,
[](Node& n, parallel_do_feeder<Node>& f) {
 if(n.left) f.add(*n.left);
```

```
 if(n.right) f.add(*n.right);
 n.data += 1;
});
```

The initial work list consists of **Root**. The functor takes a node and adds its children to the work list. The **parallel\_do** terminates when all work items have been processed.

## Reduction and Scan

The template **parallel\_reduce** performs parallel reduction (See Reductions). It has several forms. The form discussed here is **parallel\_reduce(range,identity,func, reduction)**. The **range** argument is a recursively divisible range. The **identity** argument is the identity element of the reduction operation. The functor **func** has signature **range × value<sub>1</sub> → value<sub>2</sub>**. It should do a reduction over iterations space **range** using **value<sub>1</sub>** as the initial reduction value. The functor **reduction** has signature **value<sub>1</sub> × value<sub>2</sub> → value<sub>3</sub>**. It should reduce **value<sub>1</sub>** and **value<sub>2</sub>** to **value<sub>3</sub>**. The reduction operation must be associative, but does not have to be commutative.

The following example shows **parallel\_reduce** used to reduce the first **n** elements of an array **x** of type **T**. It assumes that **T()** constructs the identity element for **+**.

```
parallel_reduce(blocked_range<int>(0,n),
T(),
[](blocked_range r, T initial) -> T {
 for(int i=r.begin(); i!=r.end(); ++i)
 initial+=x[i];
 return initial;
},
[](T a, T b) {return a+b;}
);
```

Like **parallel\_for**, template **parallel\_reduce** recursively subdivides the range. It uses the functor **func** to evaluate leaves and to perform left-to-right serial evaluation of subranges. It uses the functor **reduction** to merge results from subranges processed in parallel. Overall, it strives to keep each processor busy doing left-to-right evaluation of subranges, using just enough parallel evaluation to keep processors busy [9].

The template **parallel\_scan** computes prefix sum for any associative operation (See ▶ **Prefix**). Because parallel computation of prefix sum requires two passes, but serial evaluation requires only one pass, the template

uses a hybrid algorithm that does parallel evaluation only if idle threads are available [5].

## Partitioning Hints

An optional partitioner argument can be specified as a hint for optimizing execution. The default strategy is **auto\_partitioner**, which applies a heuristic for picking the sizes of the subranges, based on the number of hardware threads and distribution of the work-load [9]. The other two partitioners are **simple\_partitioner** and **affinity\_partitioner**.

Specifying **simple\_partitioner** causes the range to be subdivided as finely as the range argument permits. The following example is similar to the previous example, except that it specifies a grain size of 1,000 for the **blocked\_range** and a **simple\_partitioner**, which causes the **parallel\_for** to subdivide the range into subranges no bigger than 1,000.

```
parallel_for(blocked_range<int>(0,n,1000),
[&] (blocked_range<int> r) {
 for(int i=r.begin(); i!=r.end(); ++i)
 a[i] += b[i];
},
simple_partitioner());
```

Specifying **affinity\_partitioner** hints that performance might benefit from mapping loop iterations to threads similarly to previous executions of the loop or a similar loop. Codes typically benefit from **affinity\_partitioner** when the dataset touched by the parallel loop fits in the aggregate cache of a multi-processor system. The following example uses an **affinity\_partitioner** to hint that the **parallel\_for** should map iterations consistently to threads on each invocation.

```
affinity_partitioner ap;
for(int t=0; t<100; ++t)
 parallel_for(blocked_range<int>(0,n),
 [&] (blocked_range<int> r) {
 for(int i=r.begin(); i!=r.end(); ++i)
 a[i] += b[i];
 },
 ap);
```

The **affinity\_partitioner** object carries information between executions of the **parallel\_for**, and so must stay live between executions. Hence it is declared outside the **t** loop.

## Parallel Pipeline

The **parallel\_pipeline** template executes a linear pipeline of filters. Each filter is a functor. The first filter generates values. Each subsequent filter maps input values to output values. The last filter consumes its input values. The pipeline terminates when all items have been processed.

The filters can be **parallel**, **serial\_out\_of\_order**, or **serial\_in\_order**. The pipeline invokes filters concurrently subject to the following constraints:

- A given serial filter can process only one item at a time.
- Any two **serial\_in\_order** filters process items in the same respective order.

Parallelism is achieved in two ways. First, parallel filters can process any number of items in parallel. Second, different filters can run concurrently.

For instance, a three-stage pipeline might have a **serial\_in\_order** stage for reading chunks of input, a **parallel** stage for processing each chunk independently, and a **serial\_in\_order** stage for writing the output. The chunks will be written in the same order that they are read. For another example, a three-stage pipeline consisting of a **parallel**, **serial\_in\_order**, and a **parallel** stage is isomorphic to a *Doacross* loop (See ▶Loops, Parallel), where the intervening **serial\_in\_order** stage handles the cross-iteration dependency.

## Task Scheduler

The parallel algorithms build upon a task scheduler. The scheduler is non-preemptive and based on work stealing. Tasks may be *spawned* or *enqueued*. Enqueued tasks are used when first-come first-serve behavior is desired, typical of parts of programs that deal with reacting to external events such as user input. Recursively spawned tasks are used to achieve scalability of computations that run to completion.

Typically, a parent task spawns child tasks and then continues when the child tasks finish. The task interface supports two approaches for the waiting. In the *blocking-style* idiom, the parent explicitly waits for the children to complete. In the *continuation-passing* idiom, the parent specifies a continuation task to be executed when the children complete. The continuation-passing idiom is generally more tedious to write, but enables

the same space and time bounds as Cilk (See ►Cilk), because in continuation-passing style, even though tasks may wait on other tasks, no *thread* waits on another thread.

The following code demonstrates the blocking-style idiom. It derives a class **RecursiveTask** from the Intel® TBB base class **tbb::task**, and overrides method **execute()** to specify the actions of the task. The task applies function **Work** across the values in a half-open interval  $[l, u)$  via recursive decomposition.

```
class RecursiveTask: public tbb::task {
 int lower, upper;
 tbb::task* execute() {
 // Override virtual method task::execute()
 if(upper-lower==1) {
 Work(lower);
 } else {
 int midpoint = lower+(upper-lower)/2;
 // 3 = two children + one for the wait.
 set_ref_count(3);
 // Create and spawn two child tasks
 spawn(*new(allocate_child()) Recursive
 Task(lower, midpoint));
 spawn(*new(allocate_child()) Recursive
 Task(midpoint, upper));
 // Wait for my children to complete
 wait_for_all();
 }
 return NULL; // Effect of non-null value
 explained
 later.
}
public:
 // Construct task that evaluates Work(i) for i∈[l,u)
 RecursiveTask(int l, int u): lower(l),
 upper(u) {}
};
```

The **wait\_for\_all** is written separately to simplify exposition. Often the spawning of the last child and waiting are written using the shorthand **spawn\_and\_wait\_for\_all**, which optimizes away some scheduler overhead.

The following code shows how the root task for the parallel recursion is spawned. The net effect of the code is to evaluate **Work(i)** over  $[0,n)$ .

```
void TreeSpawn(int n) {
 if(n>0)
 tbb::task::spawn_root_and_wait(
 *new(task::allocate_root())
 RecursiveTask(0,n));
}
```

Root tasks are always waited on in a blocking manner, so that algorithms can present an ordinary call/return interface to users that does not return until work is completed. However, for efficiency Intel® TBB algorithms often employ continuation-passing style internally below the root task. The following code shows a continuation-passing variant of **RecursiveTask**.

```
class RecursiveTask: public task {
 int lower, upper;
 task* execute() {
 // override virtual method task::execute()
 if(upper-lower==1) {
 Work(lower);
 return NULL;
 } else {
 int midpoint = lower+(upper-lower)/2;
 // Allocate continuation. In this example, the
 // continuation exists only for
 // synchronization, and thus can be a
 // tbb::empty_task.
 task* c = new(allocate_continuation())
 tbb::empty_task;
 // Continuation has two children.
 c->set_ref_count(2);
 // Create right child of c and spawn it.
 spawn(*new(c->allocate_child())
 RecursiveTask(midpoint, upper));
 // Reuse *this as left child of c.
 recycle_as_child_of(*c);
 upper = midpoint;
 // Bypass scheduler - specify *this as next
 // task to run.
 return this;
 }
 }
}
```

```
public:
RecursiveTask(int lower_, int upper_):
 lower(lower_), upper(upper_) {}
};
```

The example shows two common optimizations for recursive spawning. First, the parent recycles itself as its left child, which avoids the overhead of creating another task and copying state from parent to child. Second, the parent does not spawn its left child, but instead returns a pointer to it. By convention, the scheduler executes that task next, skipping the overhead of pushing/popping the deque to be described shortly.

To improve cache reuse, it is sometimes desirable to have a task *y* run on the same thread on which a previous related task *x* ran. The task scheduler interface supports this preference via two methods. One method records where *x* ran, as an abstract **affinity\_id**. The other method lets the programmer specify that the scheduler should try to run *y* in the place with the same **affinity\_id**, unless load balancing requires otherwise. This approach of “replay affinity” does not require knowledge of the machine topology.

The current implementation of the task scheduler uses work stealing. Each thread has both a deque (double-ended queue) of *spawned* tasks and a mailbox of advertisements. The advertisements are for tasks in other deques that have an **affinity\_id** for the owner of the mailbox. A thread spawns a task by pushing it onto its deque. If the task has an **affinity\_id**, then the thread also sends an advertisement to the mailbox for the thread with **affinity\_id**. The threads share a queue-like structure that holds *enqueued* tasks.

A thread chooses the next task to run by the first rule below that applies:

1. Execute the task returned by last executed task.
2. Pop a task from the thread’s deque, choosing the *most recently pushed* task.
3. Dequeue an advertisement from the thread’s mailbox and steal the advertised task.
4. Pop an enqueued task from a global queue-like structure, in approximately first-in first-out order.
5. Randomly choose another thread and pop the *least recently pushed* task from it.

Each rule corresponds to a particular concern. Rule 1 enables tail recursion by tasks. Rule 2 encourages locality. Rule 3 implements “replay affinity.” Rules 4 and 5 balance load.

Rules 2 and 5 are particularly important for nested parallelism. Consider traversing a tree. If done fully parallel, the traversal can become a parallel breadth-first traversal, taking space proportional to the size of the tree. It is better to use just enough of the potential parallelism to keep the hardware threads busy, and let each busy hardware thread do sequential depth-first traversal of some subtree. Rule 2 corresponds to sequential evaluation order. Rule 5 expands actual parallelism just enough to keep an otherwise idle processor busy.

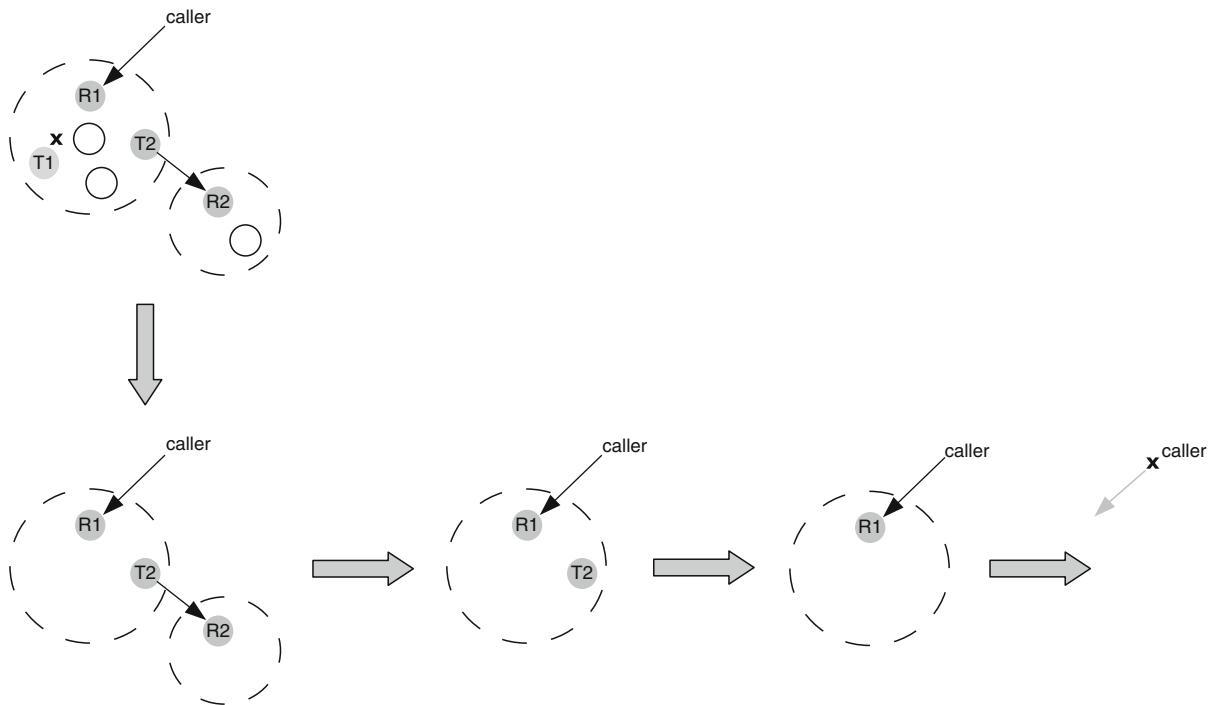
## Exceptions and Cancellation

When a parallel algorithm throws an exception, related work should be canceled and the exception propagated to the thread that initiated the algorithm. Intel® TBB exception handling performs these actions based on **task\_group\_context** nodes. Each task belongs to a **task\_group\_context** node. Typically all tasks of a parallel algorithm instance belong to the same node. The nodes form a tree. The tree of **task\_group\_context** nodes provides a structured summary of the nesting of parallel algorithms.

An algorithm typically creates a **task\_group\_context**, and then invokes **spawn\_root\_and\_wait()** on a root task associated with that **task\_group\_context**. When a running task throws an exception, the following happens:

1. The library temporarily captures the exception.
2. The library cancels the task’s **task\_group\_context** node and its descendent nodes in the tree.
3. The library cancels all tasks in each canceled node. A canceled task skips invoking its **execute()** method unless it is already running, in which case it runs to completion. A task’s destructor always runs, even if it is canceled, thus permitting proper cleanup.
4. After all canceled tasks are destroyed, the library rethrows the exception to the caller that was waiting on the root task.

Figure 1 shows propagation of an exception *x*. The small circles denote tasks. A colored circle denotes a



Intel® Threading Building Blocks (TBB). Fig. 1 Propagation of an exception  $x$  up through a task tree

task that is running. Each large dashed circle denotes a **task\_group\_context** node associated with the tasks inside it. The sequence shows:

1. Task T1's method `execute()` throws exception  $x$ , which is temporarily caught by the library.
2. Tasks are canceled in the **task\_group\_context** containing T1 and its descendant **task\_group\_context**.
3. Task T2 waits for root task R2 to complete.
4. Root task R1 waits for task T2 to complete.
5. The library rethrows  $x$  to the caller of `spawn_root_and_wait(R1)`.

The waiting in steps 3 and 4 are necessary because the only implicit cancellation point is just before task execution begins. To improve response time to cancellation, a task can explicitly poll its **task\_group\_context** and return early.

A **task\_group\_context** also can be used to explicitly cancel a parallel algorithm, from either inside or outside the algorithm. Thus, the **task\_group\_context** tree provides a structured way to process cancellation and exception handling, even though the underlying

task graphs may have much less structure and in the case of continuation-passing style, are decoupled from the thread stacks.

## Task Groups

Intel® TBB provides a higher-level task interface **task\_group** that simplifies the writing of blocking-style tasking, particularly in conjunction with C++0x lambda expressions. The following example uses **task\_group** to execute `Work(i)` for  $i \in [\text{lower}, \text{upper}]$ .

```
void Recurse(int lower, int upper) {
 if(upper<=lower) {
 // Do nothing
 } else if(upper-lower==1)
 Work(lower);
 else {
 int midpoint = lower+(upper-lower)/2;
 tbb::task_group g;
 g.run([&] {Recurse(midpoint,upper);});
 g.run_and_wait([&] {Recurse(lower,
 midpoint);})
 }
}
```

## Containers

Sometimes threads must inspect and modify a common container of data. The containers provided by the C++ standard library generally do not permit concurrent modification, and thus actions on them must be serialized. This serialization can become an impediment to scalability. Intel® TBB provides five containers that permit concurrent inspection and modification. The containers are independent of the Intel® TBB task scheduler. They may be used in conjunction with any threading package.

A **concurrent\_queue** is an unbounded queue that permits multiple producers and consumers to operate on the same queue. There is no support for blocking pop operations. Consumers use **try\_pop**, which returns a failure status if the queue is empty.

A **concurrent\_bounded\_queue** adds support for blocking and putting an upper bound on the queue size. The blocking push operation waits if the queue is full. The blocking pop operation waits if it is empty.

A **concurrent\_vector** is a vector that supports concurrent access and growing. For example, multiple threads can execute **v.push\_back(item)** on the same **concurrent\_vector v** without corrupting it. The growth operations do not move existing items, so unlike for **std::vector**, it is safe to access existing elements while the vector is growing. Besides **push\_back**, there are two other growth operations:

- **grow\_by(n)** appends n consecutive elements.
- **grow\_to\_at\_least(n)** appends zero or more consecutive elements to make the vector at least size **n**.

All growth operations return an iterator pointing to the first element appended. In the case where **grow\_to\_at\_least(n)** appends no elements, it returns an iterator pointing to just beyond the nth element.

A **concurrent\_unordered\_map** is a hash table of **(key,value)** pairs that supports concurrent **insert** and **find** operations, along with safe concurrent traversal during these operations. The interface closely resembles the C++0x **unordered\_map** interface.

A **concurrent\_hash\_map** is a thread-safe hash table of **(key,value)** pairs that supports concurrent **insert**, **find**, and **erase** operations, but not concurrent traversal. A fundamental problem of concurrent **find** and **erase** operations on the same key is that they may race in

a way such that the **find** operation returns a reference to an item that is promptly destroyed by the **erase** operation. Class **concurrent\_hash\_map** addresses the problem by combining lookup with locking. The find and insert operations return the found **(key,value)** pair via an **accessor** object. The accessor acts as a pointer to the item with an implicit lock on it. An **accessor** implies a writer lock. A **const\_accessor** implies a reader lock. Concurrent erasure of an item waits until the item has no accessors pointing to it.

## Thread Local Storage

Intel® TBB abstracts thread-local storage via container-like objects that have a separate element per thread. Method **local()** returns a reference to a thread's element, which is created lazily upon first access by its associated thread. Copying such a container copies all of its elements, retaining the element to thread mapping.

There are two such container classes. The more general class is **enumerable\_thread\_specific**. The example below shows how it can be used for reduction.

```
typedef enumerable_thread_specific<double> ets_type;
ets_type local_sums;
parallel_for(0, n, [&](int i) {
 local_sums.local() += a[i];
});
double global_sum = 0;
for(ets_type::const_iterator i=result.begin();
 i!=result.end(); ++i)
 global_sum += *i;
```

Alternatively, the final reduction can be written more concisely using method **combine**:

```
double global_sum = local_sums.combine
 (std::plus<double>());
```

Class **combinable<T>** is similar, but omits the iterator support. It exists for compatibility with Microsoft's Parallel Patterns Library.

## Memory Allocator

Because malloc scales poorly on some systems, Intel® TBB provides a scalable memory allocator. It also provides an allocator for cache-aligned allocations. All the allocators have interfaces similar to the

**C++ std::allocator.** The library also provides a C level interface that is analogous to the traditional malloc interface. Intel® TBB provides support for automatically replacing calls to the system-defined allocators with calls to the scalable allocator.

The implementation of the scalable allocator uses techniques similar to Hoard [2]. Each thread maintains its own size-segregated lists of free memory blocks. If a thread frees a block that was allocated by another thread, the freeing thread sends the block back to the originating thread.

## Mutexes

Intel® TBB provides a variety of mutex classes, which all implement a common set of signatures for ordinary mutual exclusion. The common set of signatures permits code to be parameterized with respect to the mutex type. The different types are tailored to different semantic or performance needs; for example, efficiency under light contention versus fairness under heavy contention. Most of the mutex classes have corresponding reader-writer variants that provide for shared/exclusive locking.

All of the mutex classes support a locking pattern where lock acquisition is represented by construction of a **scoped\_lock** object as shown below:

```
std::list<std::string> myList;
tbb::mutex MyMutex;
void PrependElement(std::string x) {
 tbb::mutex::scoped_lock lock(MyMutex);
 myList.push_front(x); // protected by lock on
 mutex.
}
```

Destruction of the **scoped\_lock** object causes the lock to be released. The pattern has the advantage that the programmer cannot forget to release the lock, and the lock is released even if **push\_front** throws an exception.

A subset of the mutex classes support an unscoped locking interface that follows C++0x conventions [6].

## Atomic Operations

Intel® TBB provides template class **atomic<T>** for atomic operations. Type T can be any integral, enum, or pointer type. The class supports atomic read, write, fetch-and-add, exchange, and compare-and-swap. Overloaded operators provide an intuitive interface for

fetch-and-add. For example, the following code atomically decrements RefCount and executes Foo() if it became zero.

```
extern atomic<int> RefCount;
if(--RefCount==0) Foo();
```

The Intel® TBB memory model is acquire-release consistency. Atomic reads have acquire semantics. Atomic writes have release semantics. (See ▶Memory Models). The read-modify-write operations (fetch-and-add, exchange, compare-and-swap) are sequentially consistent and (unlike C++0x sequentially consistent atomics) act as memory fences with respect to other memory operations. The read-modify-write operations take an optional argument that can specify weaker ordering.

## Future Directions

The immediate future direction of Intel® TBB is driven by the C++0x draft. Intel® TBB works with standard C++98 compilers, without special support, except that it currently depends upon vendor extensions for atomic operations and memory consistency rules because C++98 lacks a memory model for multi-threading. The C++0x draft addresses this issue. C++98 prevents propagation of an exception across threads. Intel® TBB works around this limitation by propagating an approximation of the original exception. Intel® TBB implementations built on top of C++0x can propagate the original exception across threads.

C++0x also introduces lambda expressions, which were not present when Intel® TBB was initially designed. Interfaces in Intel® TBB have been evolving to work better with lambda expressions. For example, TBB 2.2, introduced overloads of **parallel\_reduce** that specifically target ease of use with lambda expressions.

Intel® TBB has been evolving to support a subset of interfaces that are compatible with Microsoft's Parallel Patterns Library (PPL). Examples of existing PPL-compatible interfaces are **parallel\_invoke**, **task\_group**, as well as the short forms of **parallel\_for**.

## Related Entries

- ▶ [Cilk](#)
- ▶ [Loops, Parallel](#)
- ▶ [OpenMP](#)

## Bibliographic Notes and Further Reading

Intel® TBB continually evolves. The most up-to-date description is its reference manual [5]. The corresponding tutorial [4] describes use cases for various features.

The work-stealing approach to task scheduling is adapted from Cilk [3]. (See ►Cilk). There are important differences. In Cilk, the spawning thread immediately executes the child, and the parent is stolen. In Intel® TBB, a thread that spawns a child continues execution, while the spawned child may be stolen. The Cilk semantics are simulated via continuation-passing style. Where blocking style is used, Cilk guarantees on space and time do not apply. The algorithm templates in TBB generally use continuation-passing style internally, with a single blocking-style task at the root.

The mechanism for `auto_partitioner` does introspection on stealing [9]. The mechanism for `affinity_partitioner` [9] is derived from a locality-guided work-stealing technique described by Acar, Blelloch, and Blumofe [1].

## Bibliography

- Acar U, Blelloch G, Blumofe R (2000) The data locality of work-stealing. In: Proceedings of 12th annual ACM symposium on parallel algorithms and architectures, SPAA '00. Bar Harbor, Maine. ACM, New York, pp 1–12
- Berger E, McKinley K, Blumofe R, Wilson, P (2000) Hoard: a scalable memory allocator for multithreaded applications. In: Proceedings of 9th international conference on architectural support for Programming Languages and Operating Systems, ASPLOS 2000. Cambridge, Massachusetts. ACM, New York, pp 117–128
- Frigo M, Leiserson C, Randall K (1998) The implementation of the cilk-5 multithreaded language. In: Proceedings of the ACM SIGPLAN '98 conference on programming language design and implementation, PLDI '98. Montreal, Quebec, June 1998. ACM, New York, pp 212–223. DOI=<http://doi.acm.org/10.1145/277650.277725>
- Intel Corporation. Intel® Threading building blocks tutorial. <http://www.threadingbuildingblocks.org/documentation.php>
- Intel Corporation. Intel® Threading building blocks reference manual. <http://www.threadingbuildingblocks.org/documentation.php>
- ISO/IEC JTC1/SC22/WG21 C++ Standards Committee, Working draft, standard for programming language C++, N3092. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf>
- MacDonald S, Szafron D, Schaeffer J (2004) Rethinking the pipeline as object-oriented states with transformations. In: 9th international workshop on high-level parallel programming models and supportive environments, IPDPS 2004. Santa Fe, New Mexico, April 2004. IEEE, pp 12–21. DOI=<http://doi.ieeecomputersociety.org/10.1109/HIPS.2004.1299186>
- Reinders J (2007) Intel threading building blocks. O'Reilly Media, Sebastopol, California
- Robison A, Voss M, Kukanov A (2008) Optimization via reflection on work stealing in TBB. In: Proceedings of 22nd IEEE symposium on parallel and distributed processing. New York. DOI=<http://dx.doi.org/10.1109/IPDPS.2008.4536188>

## Interactive Parallelization

### ►FORGE

## Interconnection Network

- Buses and Crossbars
- Cray XT4 and Seastar 3-D Torus Interconnect
- Ethernet
- Hypercubes and Meshes
- InfiniBand
- Interconnection Networks
- Myrinet
- Quadrics

## Interconnection Networks

SUDHAKAR YALAMANCHILI  
Georgia Institute of Technology, Atlanta, GA, USA

## Synonyms

Multicore networks; Multiprocessor networks

## Definition

High-performance computing architectures utilize interconnection networks as the communication substrate between the processor cores, memory modules, and I/O devices.

## Discussion

### Introduction

In their most general form, interconnection networks are a central component of all computing and communication systems from the internal interconnects of chip-scale embedded architectures to geographic-scale systems such as wide area

networks and the Internet. This section focuses on interconnection networks as they are used in multiprocessor and multicore systems. Specifically, the section addresses basic systems concepts and principles for interconnecting processors, cores, and memory modules. These concepts and principles are not exclusive to this application domain and in fact have also found application in networks used in storage systems, system area networks, and local area networks, albeit with distinct engineering manifestations tailored to their particular domains.

Interconnection networks have their roots in telephone switching networks and packet switching data networks and, in fact, the earliest multiprocessor networks were adaptations of those networks. They have since evolved to incorporate new principles and concepts that accommodate the distinct engineering and performance constraints of multiprocessor and multicore architectures. In particular, the lack of standards such as that found in the wide-area and local-area communities created opportunities for meeting performance goals via customized solutions, which in turn has been largely responsible for the diversity of implementations. However, the shared goal of high-performance multiprocessor communication has led to a common set of principles and concepts organized around a network stack that can be viewed as being comprised of the following layers: *routing layer*, *switching layer*, *flow control layer*, and *physical layer*. These layers collectively operate to realize network performance and correctness properties, for example, deadlock freedom. The physical layer comprises signaling techniques for synchronized transfer of data across links. Switching techniques determine *when* and *how* messages are forwarded through the network. Specifically, these techniques determine the granularity and timing with which resources such as buffers and switch ports within a router are requested and released. For example, in packet switching a packet is forwarded only after it has been received in its entirety at a router whereas in virtual cut through switching bytes of a packet can be forwarded once routing information is available and the corresponding output port is free. Forwarding and reception are overlapped. The design of switching techniques is closely coupled to the behavior of flow control protocols, which ensure the availability of buffering resources as packets traverse the network.

Finally, routing algorithms determine the path taken by a message through the network.

This section describes basic interconnection network concepts covering topology, routing, correctness, and common metrics. The discussion includes the routing layer and periodically exposes dependencies on the lower level layers particularly with respect to deadlock freedom.

## Topology

Routing of communication between nodes is the most basic property of an interconnection network and is intimately tied to the pattern of interconnections between nodes where a node may be a processor, a core, a memory module, or an integrated processor-memory module. Such an interconnection pattern is referred to as the *topology* of the network and defines the set of available paths between communicating nodes. A node may be a source or destination of a message that is encapsulated in one or more packets each of which is comprised of a *header* that contains routing and control information and an optional *body* that contains data. A crossbar switch provides a direct connection between any pair of network nodes. It is also the most expensive interconnect in terms of hardware and power. Thus, as system size grows we encounter a variety of topologies that represent various performance and cost trade-offs. The most common topologies can be broadly classified into shared medium, direct, indirect, and hybrid.

## Shared Medium Networks

The bus is the most common shared medium network and the first widely used interconnect in multiprocessor architectures. A bus can have several hundred signals organized as address, data, and control. Only one node is allowed to transmit at a time as determined by the *arbitration policy* that arbitrates between concurrent requests for the bus.

Such shared medium buses do have several advantages for low node count systems. They provide an efficient broadcast medium in support of one-to-all or one-to-many communication patterns. The shared bus also serves to serialize all traffic between nodes. When used as a processor-memory interconnect, this property is exploited in cache-based multiprocessors to support coherence protocols for shared memory multiprocessors [13]. Routing on the

bus is straightforward since all devices snoop the address lines. However, if the bus is used to connect devices of differing speeds, such as processors to memory, bus utilizations can degrade rapidly if a transaction (e.g., reading from memory) holds the bus for the duration of the transaction. As processor and memory speed disparities grew, low utilizations led to *split phase transactions* where requests and responses are split into two separate transactions. As a result, multiple memory transactions can be interleaved on the bus increasing bus utilizations.

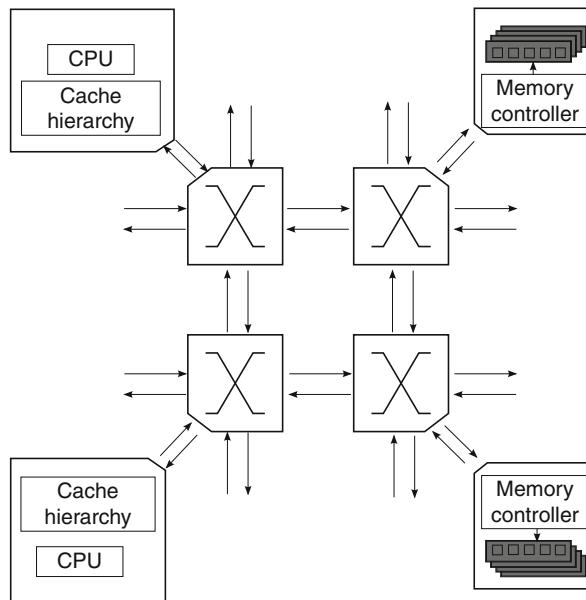
Physical constraints limit bus lengths and consequently system sizes. This has also become true for chip multiprocessors, and thus the use of buses can be expected to be limited to smaller sized systems at future technology nodes.

### Direct Networks

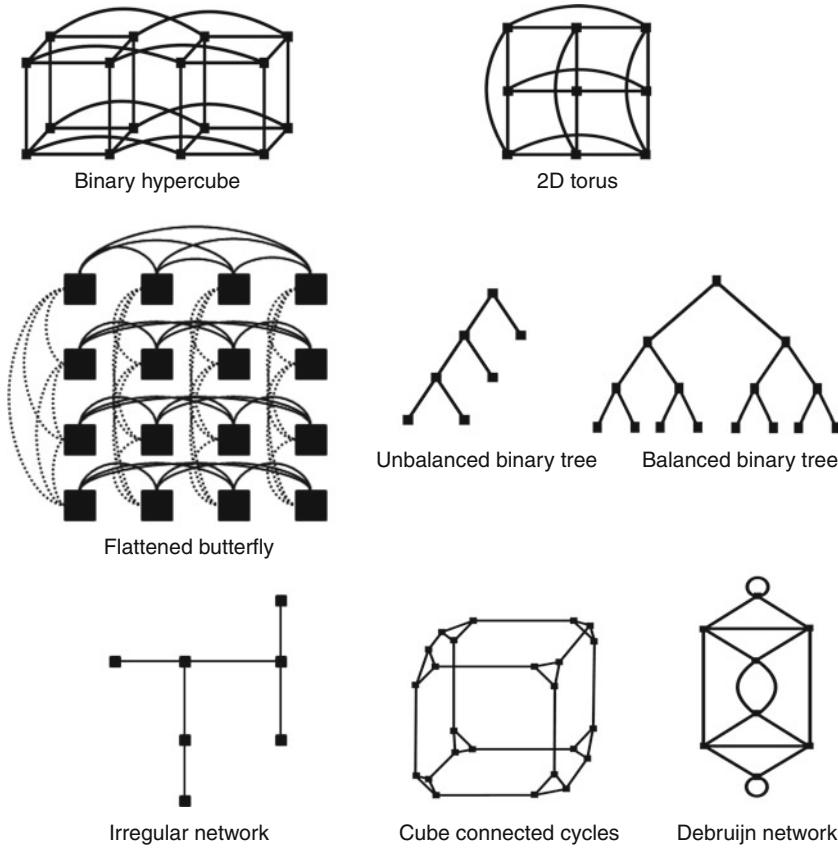
To achieve scalability beyond that which is feasible with shared medium networks, nodes are interconnected by *point-to-point links*. A node includes a router whose input and output links are connected to routers at some set of *neighboring nodes* producing a direct network. An example of a processor node in a 2D mesh network is illustrated in Fig. 1 where one of the router links connects within the local node. Each unidirectional link realizes the *injection* and *ejection* channels.

The topology refers to the overall interconnection pattern between nodes (actually between the routers to which the node is connected). A common topology, particularly for on-chip networks, is a 2D mesh as shown in Fig. 1. Formally, an  $n$ -dimensional mesh has  $k_0 \times k_1 \times \dots \times k_{n-2} \times k_{n-1}$  nodes, where  $k_i$  is the number of nodes along dimension  $i$  and is referred to as the *radix* for dimension  $i$ , and where  $k_i \geq 2$  and  $0 \leq i \leq n-1$ . Each node  $X$  is identified by  $n$  coordinates,  $(x_{n-1}, x_{n-2}, \dots, x_1, x_0)$ , where  $0 \leq x_i \leq k_i - 1$  for  $0 \leq i \leq n - 1$ . Two nodes are connected if their coordinates differ by 1 in only one dimension. Except for nodes on the boundary, all nodes are directly connected to  $2n$  other nodes. Thus, we can have a  $3 \times 4$  mesh where each interior node is connected to four other nodes or a  $6 \times 4 \times 8$  3D mesh where each interior node is connected to six other nodes. Now, messages between a pair of nonadjacent nodes are routed through the intermediate routers.

A symmetric generalization of the mesh is a  $k$ -ary  $n$ -cube [4]. Unlike the mesh, nodes at the edge have a link that “wraps around” the dimensions as shown in Fig. 2 for the 3-ary 2-cube. Now all nodes have the same number of neighbors. Formally, a  $k$ -ary  $n$ -cube has  $k_0 \times k_1 \times \dots \times k_{n-2} \times k_{n-1}$  nodes,  $k_i$  nodes along each dimension  $i$  and two nodes  $X$  and  $Y$  are neighbors if and only if  $y_i = x_i$  for all  $i$ ,  $0 \leq i \leq n - 1$ , except for dimension  $j$ , where  $y_j = (x_j \pm 1) \bmod k$ . Every node has



**Interconnection Networks. Fig. 1** An example of network nodes



**Interconnection Networks.** Fig. 2 Example direct network topologies

$2n$  neighbors. When  $n = 1$ , the  $k$ -ary  $n$ -cube collapses to a bidirectional *ring* with  $k_0$  nodes. When the radix in all dimensions are equal to  $k$ , we have  $k^n$  nodes. When  $k = 2$ , the topology becomes the binary *hypercube* or binary  $n$ -cube as shown in Fig. 2 which illustrates a 16-node binary hypercube.

A generalization of the  $k$ -ary  $n$ -cube is the *generalized hypercube* where rather than being connected to the immediate neighbors in each dimension, each node is connected to every node in each dimension. In graph theoretic terms, as we proceed from meshes through  $k$ -ary  $n$ -cubes to generalized hypercubes, additional links are added creating a denser interconnect. From a practical standpoint, as additional links are created, feasible channel widths change since the number of I/Os at an individual router have some maximum value. Finally, in all of the cases above, we note that the radix in each dimension can be distinct. These networks are often referred to as *orthogonal networks*

recognizing the independence of the routing behavior in each dimension (see ►Routing (Including Deadlock Avoidance)).

The landscape of direct interconnection network topologies is only limited by the imagination and many variants have been proposed and will no doubt continue to be proposed to match engineering constraints as technologies evolve. The preceding are the most common classes of networks used in current generation systems. Figure 2 illustrates some other common examples from the current time frame. A 2D flattened butterfly topology used in on-chip networks is equivalent to a 2D generalized hypercube. Various forms of balanced and unbalanced trees have been proposed motivated by the logarithmic distance properties between nodes. Others such as de Bruijn networks [16], star networks [2], and cube connected cycles [15] present different trade-offs between network diameter (longest path between nodes) and wiring

density. We can anticipate other direct networks to evolve in the future.

A final observation is that all of the topologies in the preceding discussion exhibit some form of regularity in structure, the edge routers in the mesh notwithstanding. A final class of networks that can be distinguished are irregular networks. Rather than a mathematical description, such networks are typically represented as graphs of interconnections and can be constructed to form any type of topology. An example is shown in Fig. 2. These networks are usually motivated by packaging constraints or customization, for example a specific topology in embedded systems where the topology is customized to the specific communication interactions between nodes. As a result of their irregularity these networks rely on distinct classes of routing techniques that are described in Routing.

### Indirect Networks

In general, direct topologies support interconnections where some routers may simply serve as connections to other routers and are not attached to any node such as a processor, core, or memory. These are the class of indirect networks. Like direct topologies, these may be organized into regular or irregular configurations.

A large class of regular indirect networks are multistage interconnection networks (MINs) an example of which is illustrated in Fig. 3. Nodes are connected to the inputs and outputs of the network. Each  $2 \times 2$  router – typically referred to as a switch in these topologies – can route a packet from one of its input ports to one of its output ports. There are  $N = 2^n$  input ports,  $N = 2^n$  output ports, and  $\log_2 N$  stages of switches with  $N/2$  switches in each stage. The interconnections between stages of switches are important – they ensure that a single path exists between every input port and every output port. In particular, consider the interconnection between stages 1 and 2 in Fig. 3. The connections can be described by the function

$$\sigma(x_{n-1}x_{n-2}\dots x_1x_0) = x_{n-2}\dots x_1x_0x_{n-1}$$

For example, output 5 of stage 0 is connected to input 3 of the next stage of switches (as shown in Fig. 3) and similarly output 6 is connected to input 5. This connection can be thought of as analogous

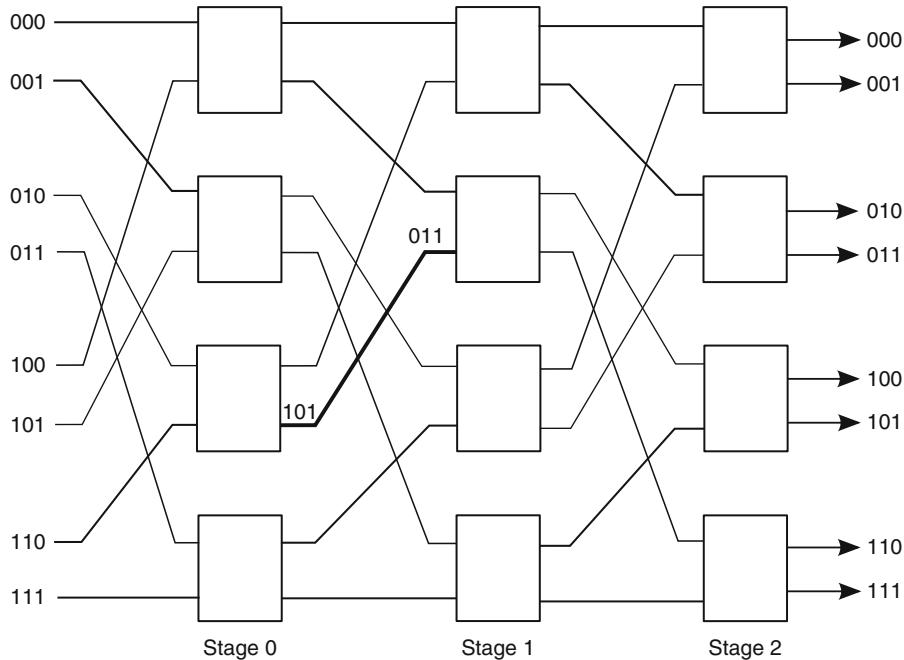
to perfectly shuffling a deck of  $N$  cards, where the deck is cut into two halves and shuffled perfectly, that is, the inputs are shuffled to the outputs. This function is referred to as the *shuffle* function and the network in the figure is the well-known Omega network. The preceding is a compact way of expressing the connections between stages. Note that every output (of a stage) is connected to exactly one input (of the next stage of switches), and therefore these connections can be represented as a *permutation* of the inputs. Consequently, such networks are referred to as *permutation networks*. In general, such networks can be constructed with  $t \times t$  switches and correspondingly  $n = \log_2 N$  stages for connecting  $t^n$  nodes.

Not all stages need to have the same interconnect pattern. For example, consider the following interconnect function.

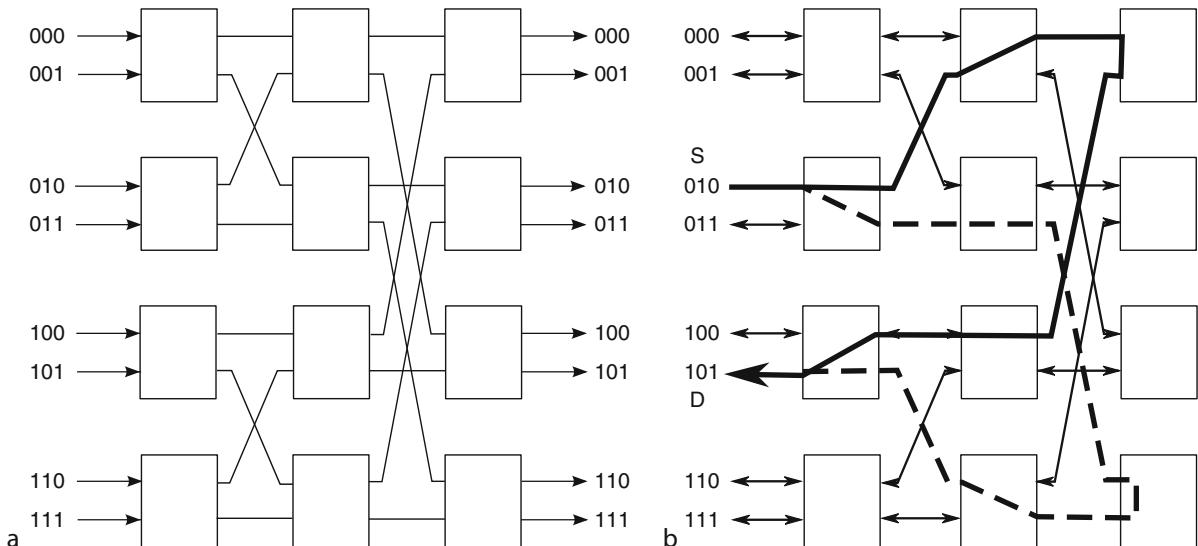
$$\beta_i(x_{n-1}\dots x_{i+1}x_ix_{i-1}\dots x_1x_0) = x_{n-1}\dots x_{i+1}x_0x_{i-1}\dots x_1x_i$$

The preceding function refers to Butterfly function forming the corresponding network of the same name illustrated in Fig. 4a. Note each stage uses a distinct butterfly ensuring every input has a path to every output. Many such MINs have been defined based on different permutation functions (see [11, 19] for a sampling). Perhaps, the oldest and best known among these is the Clos Network [3]. The Clos network is derived from a recursive decomposition of the crossbar with the first level decomposition shown in Fig. 5a. A crossbar is decomposed into a 3-stage network. Consider a network with  $N = r \times n$  inputs. The first stage comprises of  $r, n \times m$  switches while the center stage comprises  $m, r \times r$  switches. The last stage comprises  $r, m \times n$  switches. Each switch at the first stage has a connection to every switch in the middle stage. Each switch in the center stage has a connection to every switch at the last stage. Thus, there are  $m$  paths between every input and output. Each of the individual switches can be further and recursively decomposed with  $n = m = 2$  and  $r = N/2$  until the network is reduced to  $(2\log_2 N) - 1$  stages of  $2 \times 2$  switches. This network is known as the Benes network.

The MINs are characterized based on their ability to simultaneously establish connections between inputs and outputs. If a path through the MIN prevents other paths between free inputs and outputs from being set up, they are referred to as *blocking networks*. MINs



**Interconnection Networks.** Fig. 3 A MIN with a shuffle interconnection between stages

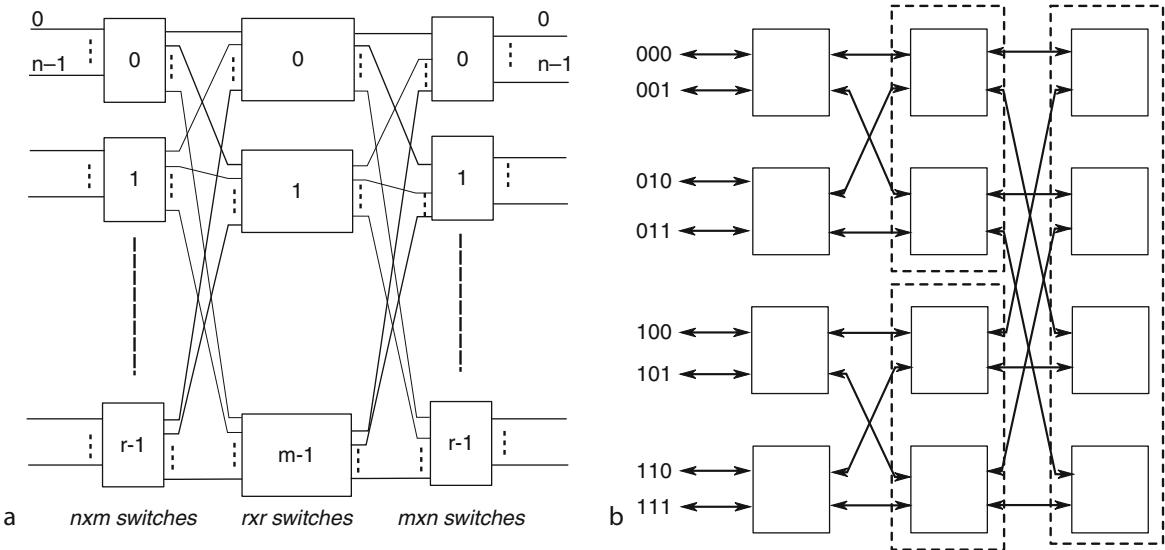


**Interconnection Networks.** Fig. 4 Some Example MINs. (a) The butterfly network. (b) A bidirectional MIN network

for which any free input can be connected to any free output are referred to as *non-blocking* while MINs where any free input can be connected to any free output by rearranging (rerouting) existing paths are referred to as *rearrangeable*. The preceding MINs with  $\log_2 N$  stages are blocking. Note that adding extra stages

introduces multiple paths between inputs and outputs until the network becomes rearrangeable and eventually non-blocking. Clos networks are non-blocking when  $m \geq 2n - 1$  and rearrangeable when  $m \geq n$ .

Finally, the preceding discussion was based on MINs with unidirectional links routing packets from input to



**Interconnection Networks. Fig. 5** Some additional MINs. (a) Crossbar decomposition of the clos network. (b) An example fat tree

output. However, MINs can operate with bidirectional links and packets can be routed to “any” output port of an intermediate switch leading to routing paths shown in Fig. 4b. Such MINs are referred to as *bidirectional MINs*. A careful look at the structure of bidirectional MINs will reveal that there exist multiple paths between any source–destination pair, an example of which is shown in the figure.

A fat tree is an indirect network that is topologically a balanced binary tree topology with the modification that the bandwidth of the links increases as one gets closer to the root (hence the label “fat”) [14]. This is typically achieved by adding more links in parallel closer to the root. Routing is straightforward in a fat tree – a message is routed up to the nearest common ancestor and then down to the destination. Thus, local communication is restricted to sub-trees. Fat trees are universal in the sense that it is the most efficient network for a given amount of hardware (see [14] for definitions of efficiency and universality). A butterfly BMIN with turnaround routing can be viewed as a *fat tree* – consider the dashed boxes of Fig. 5b as single internal switches in a fat tree. In this case, it can be viewed as having four  $2 \times 2$  switches in the first stage, two  $4 \times 4$  switches in the second stage, and a single  $8 \times 8$  switch in the last stage (the root of the tree). Modern fat tree topologies have evolved into a parametric family

as a function of (a) the number of levels in the tree, (b) the number of switches at each level, and (c) the switch sizes.

As with direct networks, the preceding MINs are considered regular, and it is possible to consider a range of irregular indirect networks of arbitrary topologies with nodes connected to an arbitrary subset of switches. As before, this has a clear impact on the routing protocols.

### Hybrid Networks

As the name suggests, hybrid networks are customized networks that couple various topological concepts described so far. For example, we might have a mesh of clusters where nodes within a cluster are interconnected by a high-speed bus. Or, we may have a hierarchy of ring interconnections. Hybrid networks are often motivated by the target packaging environment and incorporation of new technologies. For example, continued progress in optical communication, especially silicon nanophotonics, will lead to networks where interconnects and topologies are optimized for a combination of optical and electrical signaling.

### Routing

The routing algorithm establishes a path through the network and can be distinguished by the number

of candidate paths that can be produced between a source and destination. *Deterministic* routing protocols provide a single path between every source–destination pair. *Oblivious* routing algorithms produce routes independent of the state of the network, for example independent of local congestion. While deterministic routing algorithms are oblivious, the converse may not be true. For example, the source may cycle through alternative (deterministic) paths to the destination for successive packets in an attempt to balance network traffic across all links while not being influenced by the current state of any router. Another example is Valiant's two-phase routing protocol [20] where a packet is first transmitted to a randomly chosen node and then routed deterministically to the destination. *Adaptive* routing algorithms operate in response to network conditions in making routing decisions at each router by selecting from a set of alternative output ports at intermediate routers and thereby exposing multiple alternative paths to a destination. All of the preceding routing algorithms may cause a packet to follow a *minimal* path to the destination or permit *non-minimal* paths.

Implementations of routing algorithms admit to several alternatives. In *distributed routing*, each router implements a routing function that provides an output port(s) for each packet based on some combination of destination and/or input channel and/or source. Several implementations of distributed routing are feasible including table look-up or finite state machine implementations. In *source routing*, the source node prepends an encoding of the entire path to the destination, for example, as a sequence of output ports at each intermediate router. Each router along the path simply queries this header to determine the local output port along which to forward the packet. Source routing is invariably deterministic.

Many refined characterizations of routing algorithms have also emerged over the years. In *multiphase* routing algorithms, packets employ different routing algorithms during different phases of transmission. In *optimistic routing* algorithms, packets can freely choose any path to the destination since deadlock freedom is ensured via recovery rather than avoidance (see ►Deadlocks). Alternatively, in *conservative routing* algorithms, packets advance only when progress to the destination along the selected path is

assured. Principle features of routing algorithms and an accompanying taxonomy can be found in [11].

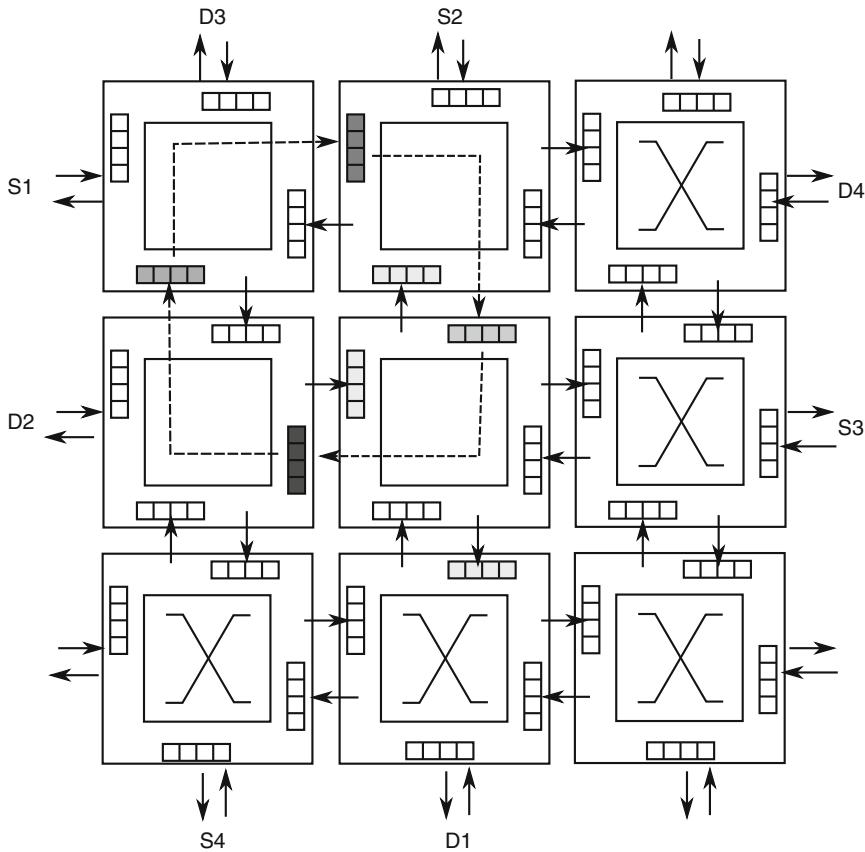
## Deadlock and Livelock

For any given topology, the routing algorithms must realize deadlock-free and livelock-free operation. *Deadlock* refers to the network state where some set of messages are indefinitely blocked waiting on resources held by each other. *Livelock* refers to a packet state, where a packet is routed through the network indefinitely and is never delivered. The presence of faulty components can lead to both deadlock and livelock behavior if mechanisms to prevent them are not implemented.

An example of a deadlocked configuration of messages is shown in Fig. 6 where messages are being transmitted between sources  $S_i$  and destinations  $D_i$  in a 2D mesh using a deterministic shortest path routing and input buffered switches (nodes attached to each router are not shown). Every message is waiting on a buffer used by another message producing a cycle of dependencies where each message is transitively waiting on itself. When routing is adaptive more complex configurations of deadlocked messages can occur where each message is blocked waiting on multiple buffers occupied by multiple messages and the set of messages cannot make progress since each message is transitively waiting on itself along all feasible output ports at the intermediate router.

*Deadlock avoidance* mechanisms are designed such that packets request and wait for resources (e.g., buffers) in such a way as to avoid the formation of deadlocked configurations of messages or otherwise guarantee progress. Routing algorithms for a topology are designed in conjunction with switching techniques to avoid deadlock. The most commonly employed principle, which is also common to deadlock avoidance in operating systems, is to provide a global ordering of resources and for all packets to request resources in this order. Specific applications of this principle are topology specific and are described in ►Routing (Including Deadlock Avoidance).

In contrast, *deadlock recovery* mechanisms promote optimistic routing of messages in the absence of any deadlock avoidance rules [1]. Deadlocked configurations of messages are detected and recovery mechanisms are invoked that reestablish forward



**Interconnection Networks. Fig. 6** An instance of deadlock

progress of packets. The application of deadlock recovery is motivated in network configurations where the probability of deadlock occurrence is very low. This is often the case in adaptively routed networks with large numbers of virtual channels.

All deadlock freedom proofs rely on the *consumption assumption* – messages reaching a destination are eventually consumed. However, the end points can introduce dependencies between incoming packet channels and outgoing packet channels since there may be dependencies between message types, for example processing of a message may lead to the injection of new messages. Consequently, it is possible to introduce a dependency between injection and ejection buffers external to the networks and which is not addressed by routing restrictions in the network. This can potentially introduce cyclic dependencies and therefore deadlock referred to as *protocol* or *message-dependent* deadlock [18]. For example, in shared memory systems

message traffic consists of request packets (for cache lines) that produce response traffic (cache lines). Traffic is often separated into two virtual networks – the request network and response network. Thus, protocol handlers at the end points can ensure that each network satisfies the consumption assumption and that the end point is never indefinitely blocked waiting on injection (full) or ejection (empty) channels.

### Deadlock-Free Routing

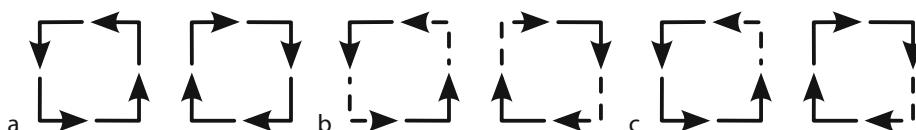
The literature on packet data networks is rich with techniques for deadlock-free routing (avoidance). This section will be focused on principles and common techniques that are utilized in multiprocessor interconnection networks that in one form or another enforce ordering constraints to ensure deadlock freedom.

The simplest example of deadlock free routing is in 2D meshes using X-Y routing where a packet is routed in the X dimension eliminating the X-offset (find the right column) before routing in the Y dimension (eliminate the Y-offset) – also known as *dimension order routing*. This avoids deadlocked configurations since no message traversing the Y dimension can request a channel in the X dimension and create a cycle of dependencies as illustrated in Figure 6. This deadlock-free routing algorithm is naturally extended to  $n$  dimensions where all packets follow the same order of dimensions in eliminating offsets in each dimension. The *Turn Model* generalizes dimension order restrictions on routing by observing that packets must be prevented from moving from one class of channels to another, that is, performing a *turn* [12]. In dimension order routing in 2D meshes packets are prevented from turning from the Y dimension to the X dimension. Many Turn-based deterministic and adaptive routing algorithms have been developed in the literature. One of the principle attractions of the Turn Model is that it can be applied to networks without virtual channels. Figure 7a shows all possible turns for a 2D mesh. Figure 7b shows those turns (dashed arrows) that are restricted for X-Y routing. Note that two turns are restricted in each cycle. However, restricting the use of a single turn in each cycle is sufficient to avoid deadlock. An example is illustrated in Figure 7c.

Deadlock-free routing in irregular networks can be achieved with similar ordering constraints. For example, a distinguished node can be identified in the network (e.g., the root of a spanning tree). A labeling algorithm labels all channels as either being *up* and directed toward the root or *down* and being directed away from the root. Packet routing is now constrained to use all up channels followed by all down channels. A packet cannot turn from a down channel to an up channel avoiding the creation of cyclic dependencies [17].

The wrap-around links in multidimensional tori introduce hardware cycles within a dimension and therefore dimension order routing is not deadlock free (consider the case where each node in a row is simultaneously transmitting to a destination  $h$ -hops away in the same row). For deterministic routing the work of Dally and Seitz [6] showed that the necessary and sufficient condition for deadlock-free routing is an acyclic channel dependency graph – intuitively if a packet can be routed from channel A to channel B, there is a dependency from A to B. Acyclic dependency graphs are achieved in tori with the presence of two virtual channels per physical channel – say VC0 and VC1 – and placing restrictions on how these channels may be used. A single link in each dimension is chosen (sometimes called the dateline link). The routing algorithm constrains the packets being routed across this link to use one virtual channel class. Coupled with the routing rules across other links, cycles in the channel dependency graph and hence deadlock is avoided. The requirement of acyclic channel dependency graphs can be implemented in many ways by creatively restricting the use of virtual channels in regular and irregular networks.

The theory of deadlock free adaptive routing protocols was developed by Duato [7, 8]. Intuitively, cycles may exist in the channel dependency graph as long as there exists an acyclic subgraph that enables all messages to be routed between any source destination pair – sometimes referred to as an *escape path* or *escape channel* at a router. Virtual channels provide a natural medium for implementing adaptive routing. In the case of tori, each physical link supports three virtual channels – VC0, VC1, and VC2. The first two provide the acyclic subgraph necessary for deadlock free routing. VC2 is the adaptive channel that can be used by a packet to cross any physical link to the destination. The theory has been developed for wormhole, packet, and virtual cut-through implementations [7–10]. In general,



**Interconnection Networks. Fig. 7** Possible and restricted turns in a 2D mesh. (a) Possible turns in a 2D mesh; (b) Restricted turns for X-Y routing; (c) Minimum turn restrictions for deadlock freedom

by suitably specifying routing restrictions, adaptive routing protocols can be developed for regular and irregular networks.

## Metrics

The construction of an interconnection network is a process of trade-offs between various network design parameters such as topology, channel widths, and switch degree and the physical constraints of the target implementation environment such as on-chip vs. in-rack vs. inter-rack. The consequences of the choice of network parameters are captured in several common metrics.

A practical constraint encountered by interconnection networks is the wiring density. Since networks must be implemented in three dimensions, for an  $N$  node network the available wiring space can be seen as growing as  $O(N^{\frac{2}{3}})$  which corresponds to the area of the surface of a 3D cube containing the  $N$  nodes. Network traffic can grow as  $O(N)$ . As system sizes are scaled, wiring limits are eventually encountered. A common metric to capture wiring limits is the *bisection width* of a network defined as the minimum number of wires that must be cut when the network is divided into two equal sets of nodes. The intuition is that in the worst case, every node on one side of the minimum cut will be communicating with every node on the other side of the cut maximally stressing the communication bandwidth of the network. For regular topologies such as  $k$ -ary  $n$ -cubes the minimum bisection width is orthogonal to the dimension with the largest radix. For irregular topologies, the min-cut defines the bisection width. Bisection bandwidth is usually computed as the bandwidth across these wires implicitly considering only data signals. The bisection widths of several topologies where each link between adjacent nodes is  $W$  bits wide are listed in [Table 1](#).

**Interconnection Networks. Table 1** Examples of bisection width and node size of some common net-works ( $t \times t$  switches are assumed in the MIN)

| Network               | Bisection width | Node size |
|-----------------------|-----------------|-----------|
| $k$ -ary $n$ -cube    | $2Wk^{n-1}$     | $2Wn$     |
| Binary $n$ -cube      | $\frac{NW}{2}$  | $nW$      |
| $n$ -dimensional mesh | $Wk^{n-1}$      | $2Wn$     |
| MIN                   | $NW$            | $2tW$     |

A second physical constraint is the number of data signals on a router referred to as the *node degree*, *node size*, or *pin-out*. The importance of node degree stems from the fact that it is a practical limit on routers. Further, the topology, bisection width, and node degree are closely related. Consider the implementation of a  $k$ -ary  $n$ -cube with the bisection width constrained to be  $N$  bits. The maximum channel width determined by this bisection width is  $W$  bits and is given by

$$\text{Bisection width} = 2Wk^{n-1} = N \Rightarrow W = \frac{k}{2} \quad (1)$$

The channel width directly impacts message latency and hence performance. The specific relationships between channel widths, node degree, and bisection bandwidth vary significantly whether the network is implemented on-chip, intra-rack, or inter-rack. These metrics also provide a uniform basis for comparison between various topologies.

## Related Entries

- ▶ [Networks, Direct](#)
- ▶ [Networks, Multistage](#)
- ▶ [Cray XT4 and Seastar 3-D Torus Interconnect](#)

## Bibliographic Notes and Further Reading

Each of the major topics discussed in this section have been subjected to a rich and diverse development. Entries can be found in the encyclopedia for all levels of the interconnection network stack. At the lowest level, the entry on *Flow Control* elaborates the issues and solutions for managing buffer resources as packets progress through a network. One level up, *Switching Techniques* are closely coupled with flow control and interact with higher-level routing protocols that are covered in the entry on *Routing (Including Deadlock Avoidance)* to determine no-load latency properties of an interconnection network. However, workloads can place highly nonuniform demands on the network. The entry on *Congestion Management* introduces a variety of techniques for managing such nonuniform demands and enhancing performance. A very important special case of traffic patterns is collective communication, and various approaches toward support for such communication patterns can be found in the entry on *Network Support for Collective Communication*. High-performance network designs are especially challenging when the network must be resilient to

faults. Basic issues and approaches can be found in the entry on *Fault Tolerant Networks*. Finally, the injection and ejection interfaces to the network must be designed to deliver network performance to the end point hosts. Basic architectural and operational principles are covered in *Network Interface*.

A combined coverage of fundamental architectural, theoretical, and system concepts and a distillation of key concepts can also be found in two texts [5, 11]. Advanced treatments can be found in papers in most major systems and computer architecture conferences with the texts contributing references to many seminal papers in the field.

## Acknowledgments

Feedback and comments from Michelle Rasquinha and Jeffrey Young are gratefully acknowledged.

## Bibliography

1. Anjan KV, Pinkston TM (1995) An efficient fully adaptive deadlock recovery scheme: DISHA. In: Proceedings of the 22nd international symposium on computer architecture, IEEE Computer Society, Silver Spring, MD, pp 201–210
2. Akers SB, Krishnamurthy B (1989) A group-theoretic model for symmetric interconnection networks. *IEEE Trans Comput C-38(4)*:555–566
3. Clos C (1953) A study of non-blocking switching networks. *Bell Syst Tech J* 32(5):406–424
4. Dally WJ (1990) Performance analysis of k-ary n-cube interconnection networks. *IEEE Trans Comput C-39(6)*:775–785
5. Dally WJ, Towles B (2004) Principles and practices of interconnection networks. Morgan Kaufman, San Francisco, CA
6. Dally WJ, Seitz CL (1987) Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans Comput C-36(5)*:547–553
7. Duato J (1993) A new theory of deadlock-free adaptive routing in wormhole networks. *IEEE Trans Parallel Distrib Syst* 4(12):1320–1331
8. Duato J (1995) A necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks. *IEEE Trans Parallel Distrib Syst* 6(10):1055–1067
9. Duato J (1995) A theory of deadlock-free adaptive multicast routing in wormhole networks. *IEEE Trans Parallel Distrib Syst* 6(9):976–987
10. Duato J (1996) A necessary and sufficient condition for deadlock-free routing in cut-through and store-and-forward networks. *IEEE Trans Parallel Distrib Syst* 7(8):841–854
11. Duato J, Yalamanchili S, Mi L (2003) Interconnection networks. Morgan Kaufmann, San Francisco, CA
12. Glass CJ, Ni LM (1992) The turn model for adaptive routing. Proceedings of the 19th International Symposium on Computer Architecture, Queensland, Australia, pp 278–287
13. Patterson D, Hennessey J (2004) Computer architecture: a quantitative approach. Morgan Kaufmann, Palo Alto, CA
14. Leiserson CE (1985) Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans Comput C-34*:892–901
15. Preparata FP, Vuillemin J (1981) The cube-connected cycles: a versatile network for parallel computation. *Commun ACM* 24(5):300–309
16. Samatham MR, Pradhan DK (1989) The de Bruijn multiprocessor network: A versatile parallel processing and sorting network for VLSI. *IEEE Trans Comput C-38(4)*:567–581
17. Schroeder MD, Birrell AD, Burrows M, Murray H, Needham RM, Rodeheffer TL (1991) Autonet: a high-speed, self-configuring local area network using point-to-point links. *IEEE J Select Areas Commun* 9(8):1318–1335
18. Song YH, Pinkston TM (2003) A progressive approach to handling message-dependent deadlock in parallel computer systems. *IEEE Trans Parallel Distrib Syst* 14(3):259–275
19. Wu CL, Feng TY (1980) On a class of multistage interconnection networks. *IEEE Trans Comput C-29*:694–702
20. Valiant LG (1982) A scheme for fast parallel communication. *SIAM J Comput* 11:350–361

## Internet Data Centers

### ► Data Centers

## Inter-Process Communication

- Collective Communication
- Collective Communication, Network Support for

## I/O

XIAOSONG MA

Oak Ridge National Laboratory, Raleigh, NC, USA  
North Carolina State University, Raleigh, NC, USA

## Synonyms

High-performance I/O

## Definition

Parallel I/O refers to input/output operations where multiple processes working together in executing a parallel application concurrently access secondary storage devices to exchange data between the main memory and files. The discussion in this entry will

be focused on parallel I/O in the context of HPC (High Performance Computing), especially scientific computing on large-scale parallel machines.

## Introduction

I/O is an integral part of computing, scientific and commercial both alike. For parallel scientific applications, the purpose of their execution is to generate data through simulation, or to analyze data collected from observation instruments or computer simulations (which usually will in turn produce results stored in data files). With today's supercomputers, 100,000s of processes may work in parallel in executing a single parallel program and accessing the secondary storage system. Unlike other environments with high I/O *concurrency*, such as commercial settings where a large number of client requests are processed by web server clusters or data centers, the HPC parallel I/O workloads are generated from much more tightly coupled/synchronized computation and often demand high data bandwidth in addition to low access latency. Moreover, parallel I/O plays an important role in HPC fault tolerance – by providing efficient checkpointing and restart operations.

Parallel I/O faces tremendous performance and scalability challenges in the current era of Peta-scale computing. First, the well-known performance gap between the computing components (CPU and memory) and the secondary storage devices (hard disks) keeps widening. Although disk speed has also been continuously growing in the past decades, hard disks appear slower and slower. Second, such performance gap is worsened by the limitation in parallelism. On state-of-the-art supercomputers, there are typically orders-of-magnitude more I/O client nodes compared to I/O server nodes, creating complex resource sharing and interference problems when data travel from individual compute nodes' memory to shared interconnections, I/O servers, and storage devices. In particular, this challenge is intensified by the trend of having more CPU cores within each compute node, when the clock rate improvement predicted by the Moore's law cannot sustain due to semiconductor and heat management limitations. Such increased computation parallelism aggravates the I/O intensity and contention levels, making careful I/O

subsystem design crucial to delivering a balanced overall performance.

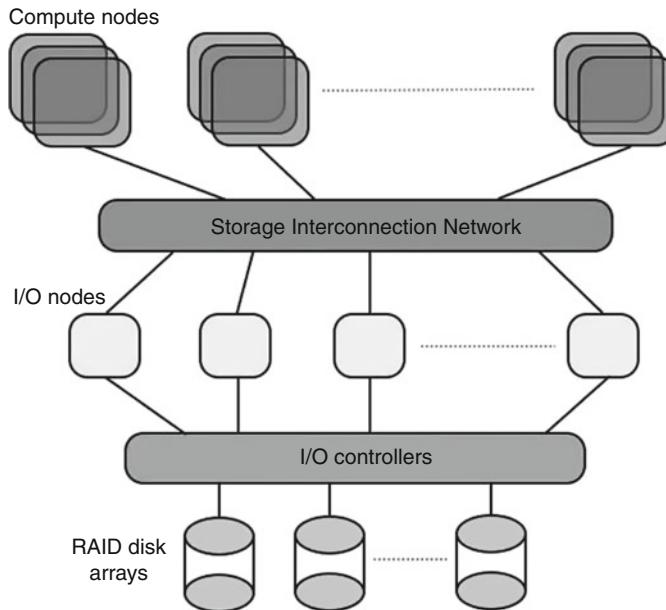
## Hardware Environments

Parallel I/O, in a more general sense, can be carried out in most distributed storage systems, such as in typical commercial/R&D settings, when storage and I/O hardware are shared by multiple users or client applications. Examples include the NAS (Network Attached Storage) and SAN (Storage-Area Networks) environments.

In the context of HPC, where the term “parallel I/O” is mostly used, there are several common hardware and networking features in addition to distributed accesses to shared storage devices. HPC I/O environments, designed for efficient file accesses performed by closely synchronized parallel applications (especially parallel simulations), often possess highly homogeneous hardware at both the client and server sides, as well as strictly hierarchical and symmetrical network structure.

[Figure 1](#) demonstrates a representative hardware and interconnection architecture designed for high-performance parallel I/O. Such an I/O environment can be viewed as a multilevel client-server architecture. On today's supercomputers (see configurations of state-of-the-art machines on the Top500 list [39]), the client side consists of 10,000s or more *compute nodes*, where users run their parallel programs, typically in batch mode. These programs issue I/O requests through the client layer of the parallel file system, which also resides on the compute nodes. Each node is equipped with its own main memory, shared by multiple cores. Smaller clusters may have node-attached local secondary storage (hard disks), while such a configuration has become rare on supercomputers these days.

The nodes are connected with each other through one or more interconnection networks (such as InfiniBand or Myrinet), to communicate with each other. Message passing has been the dominant inter-process communication scheme in the past 2 decades and currently remains the main stream. The majority of parallel applications today adopts the MPI (Message Passing Interface) [21, 22] programming model. Also connected to the compute nodes are *I/O nodes*, which are I/O servers handling client I/O requests and run the server



I/O. Fig. 1 Sample High Performance Computing (HPC) I/O hardware architecture

layer of the parallel file system. With modern supercomputers, it is not rare that there are redundant interconnection networks among nodes (typically with different types of topology), for better performance and fault tolerance. One of these networks can potentially be assigned as the main I/O interconnection.

On supercomputers these days, the ratio of compute to I/O nodes often varies between dozens to hundreds. Unlike in the case of compute nodes, the existence and operation of I/O nodes are typically transparent to application programmers. However, the combination of parallel I/O libraries (such as MPI-IO) and parallel file systems, both to be discussed in the next section, allows user programs to perform parallel I/O and access shared files via the I/O nodes implicitly through MPI interfaces.

While small or medium clusters often have disks directly attached to the I/O nodes, on large-scale supercomputers they are connected with another interconnection network, to form a SAN. A group of controllers can be placed between the I/O node layer and the disk layer, connecting hundreds or more I/O nodes to 10,000+ hard disks. For performance and fault tolerance, these disks are usually organized into RAID [31] groups. Data striping is performed at the file system level, and further at the storage device level.

## Parallel I/O Software

### Overview of Parallel Application Execution and I/O Scenario

On today's typical HPC platforms, parallel I/O is carried out in multiple stages of parallel jobs. Such jobs are executed mostly in batch mode, submitted through job scripts and managed by batch job schedulers/managers such as LSF, PBS, MOAB, and LoadLeveler. Jobs submitted wait in the job queue until dispatched by the scheduler, to execute on the requested number of processors/cores. Usually such a partition of computation hardware (compute nodes, including CPUs and memory, and in some cases local secondary storage resources such as hard disks and nonvolatile memory devices) is occupied by the job until it finishes or its execution time has exceeded the maximum specified in the job script. On the other hand, concurrent jobs on a parallel computer need to compete with each other in using shared resources, such as the interconnection networks and the share/parallel file system.

With parallel simulations, the dominant type of applications running on 100s to 100,000s processor cores, the execution of a job typically goes through many computation timesteps in an iterative manner. Often there is certain correspondence between a

computation timestep with a physical timestep, therefore the simulation computes the development of the object or phenomenon being studied along the time dimension. With most simulation codes, the execution alternates between *computation phases* and *I/O phases*, by having one I/O timestep every  $n$  computation timesteps. Application developers or users can configure the frequency of such periodic I/O operations by adjusting  $n$ .

There are two major types of data written during a parallel simulation run: *checkpointing data* that save the current status of the computation, so that the computation can be restarted after a crash, a failure, or at a given intermediate point of simulation, and *result data* that save an intermediate “snapshot” of the simulation, for future analysis and visualization. Concatenating multiple snapshots generated by an application run allows scientists to view a movie demonstrating the process being simulated. In contrast, often no more than two consecutive checkpoints are needed for fault-tolerance purposes. Obviously, in choosing a desired periodic I/O frequency, users have to consider the trade-off between I/O cost and resolution. More frequent I/O steps generate more detailed view of the simulation and allow for less redundant recomputation in case of hardware/software failures, but will cost more time spent on I/O during the job run and result in larger result datasets that need to be stored, migrated, and processed. In most cases, application users tune the I/O frequency so that the visible time spent on I/O operations within a run is under a certain budget, e.g., 5–10% of the total parallel execution time (wall time).

In addition to periodic checkpointing and result data output, applications may write out miscellaneous data to assist book-keeping, debugging, and performance diagnosis/analysis. Such data include application-level logs and library-level profiling data and traces. Meanwhile, for most simulation codes the demand for input is relatively small: simulation jobs typically only read input data (from files describing the initial simulation state or a certain checkpoint) during the start or restart process, to initialize data structures before the first timestep. Therefore, simulation codes are often considered write-intensive, with the exception of *out-of-core* applications, where the aggregate memory is not large enough to accommodate the entire data

footprint, requiring applications to repeatedly swap data between the main memory and the secondary storage. Non-simulation applications (such as analytics and parallel data base search codes), on the other hand, may have larger portions of input in their I/O workloads.

Besides the I/O needs during the execution of a job, staging operations are needed to move input data to and move output data from a supercomputer center. The high-performance file systems at these centers are “scratch space” for jobs’ transient use instead of long-term storage. With parallel data transfer tools such as GridFTP, input staging and output offloading can also be carried out with multiple streams accessing files simultaneously.

## HPC I/O Access Patterns

HPC applications have traditionally been using several modes for performing I/O. One brute force approach is to have every process communicate their data to/from a root process (typically process 0), which is in charge of all file I/O. This approach is simple and can easily coordinate data to/from multiple processes while accessing files with traditional, single-process I/O interfaces. The problem is also obvious: parallel I/O is not used and the I/O process is serialized through the root process, which quickly becomes the I/O bottleneck even in a midsize cluster.

Another rather straight-forward alternative to the above approach is to have each process perform their file I/O independently, to/from separate files. This approach is also easy to implement, and in many cases results in good performance in terms of the aggregate I/O throughput. By accessing nonoverlapping sets of files, many parallel processes can go ahead with I/O without coordinating with each other to decide their data’s global offsets or synchronizing through file locks. Within each file, accesses can also be largely sequential. However, this approach brings the need of creating, managing, and using a large number of small files. Such a management overhead is prohibiting for long-running, large-scale applications, which easily generate millions of files if independent file I/O is used. Also, this approach makes it more challenging to restart an application with a different number of processes.

Considering the limitations of the above two methods, many large-scale applications with significant

periodic I/O requirement choose to go with the “true parallel I/O” mode, where many processes collaboratively access one or more shared files during each I/O phase. This is often done via parallel I/O libraries (see section “Parallel I/O Libraries” below). This approach combines the benefits of the first two methods: a small number of “global” files plus concurrent file I/O accesses.

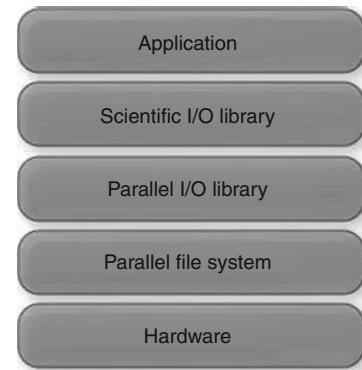
Such shared-file accesses create more interesting I/O patterns, as many processes need to map their in-memory data partitions to on-disk file range partitions, to store the global data object in layouts that facilities future restart or analysis. Therefore, the access pattern heavily depends on the domain decomposition method adopted by the parallel computation model. For regular applications that work on multidimensional arrays, commonly seen patterns are similar to HPF (High Performance Fortran) [11], such as BLOCK and CYCLIC. Fine-granule interleaving pattern maybe created by corresponding strided data distribution, or a mismatch between in-memory and on-disk data layouts. For example, a 2-D array distributed to  $p$  processes in column-major blocks may produce such an interleaving pattern when the shared array needs to be written out to disk in row-major order. Such finely interleaved accesses can be aggregated and reorganized by the parallel I/O libraries, though, to alleviate the I/O performance concern.

### Parallel I/O Software Stack

Between a parallel application and the storage media, parallel I/O requests usually have to go through a multilayer I/O software stack, across the user and the kernel space. Figure 2 illustrates such a deep I/O stack and the software layers will be discussed in the following sections. Please note that performing parallel I/O does not require all layers to be included. For example, off-the-shelf clusters often do not have parallel file systems, and applications may choose not to use high-level scientific data libraries or parallel I/O libraries.

### Parallel File Systems

Parallel file systems aim to bring to parallel computer users a file system view that is similar to traditional serial file systems on a single computer: a unified, logical storage space. It hides from programmers



I/O. Fig. 2 Sample HPC I/O stack

and users the fact that physical storage spaces are attached to distributed I/O servers, which are in turn interconnected to the compute nodes where the application programs run. Currently, parallel file systems are installed on supercomputers and medium/large clusters, while small clusters often use distributed file systems such as NFS [29].

In addition to the routine features of distributed file systems, such as file and directory management, request routing, and disk space management, parallel file systems have features and strategies that are designed to fit parallel I/O demands. One important feature is file striping, which partitions file into chunks and distribute them across I/O servers and further across disks. With intelligent striping strategies, a parallel file system can achieve reasonable I/O load balance for both large and small file accesses, as well as system extensibility when new servers and disks are added. In addition, parallel file systems take into account the HPC I/O workload (multiple parallel jobs running concurrently) in making design choices regarding file block size, metadata server configuration, distributed lock management, caching/prefetching policies, fault-tolerance mechanisms, etc.

As of now, the most commonly used parallel file systems include the proprietary GPFS [33] (available on IBM systems), and two open-source systems: Lustre [8], and PVFS [5].

### POSIX I/O

File systems used in HPC environments usually support I/O accesses via the well-known POSIX (IEEE

Portable Operating System Interface for Computing Environments) interfaces. POSIX interfaces can be used at different levels in a parallel program's execution: they can be directly called by the programmer to carry out I/O operations, and they are used eventually by higher level I/O middleware (such as parallel I/O libraries) to access disk files. With the former approach, these familiar interfaces (such as `open`, `close`, `read`, and `write`) allow programs extended from sequential codes to easily run on parallel file systems. By having multiple processes carry out I/O simultaneously, “parallel I/O” can be achieved simply through POSIX interfaces. However, they do not assist parallel program developers in achieving high-performance I/O, especially to perform coordinated accesses to shared files with high I/O concurrency [15]. Also, strict enforcement of POSIX consistency may prevent applications from applying performance optimizations.

## Parallel I/O Libraries

Parallel I/O libraries act as middleware between the underlying parallel file system (accessed via POSIX/POSIX-like I/O interfaces) and the parallel application. Most importantly, parallel I/O libraries provide applications with convenient interfaces to easily access shared files concurrently, as well as performance optimizations that often dramatically improve applications' parallel I/O efficiency.

For ease of use, parallel I/O libraries supply users with file I/O interfaces to simultaneously open/close shared files and share common file pointers, as well as to conduct both *individual* and collective I/O. In particular, collective I/O [3, 13, 27, 34, 38] allows processes to collaborate in reading/writing their portions of data from/to shared files, using data distribution descriptions that map naturally to the domain decomposition methods used widely in parallel simulation codes. For example, a parallel program that distribute a large 3-D double array across 256 processes in  $8 \times 8 \times 8$  grids can conveniently write out their sub-arrays into the global array, stored in row-major order in a shared file. With parallel I/O libraries, this can be done with a single collective I/O call (which also forces implicit synchronization in a way similar to a barrier call). The complicated data reorganization and correspondent inter-process communication are

therefore hidden from the programmer. There are also interfaces for programmers to define more sophisticated or irregular data distribution, such as with indirection arrays.

Several collective I/O architectures have been proposed in the past, including DDIO (Disk Directed I/O) [13] and SDIO (Server Directed I/O) [34]. The current prevalent architecture is 2PIO (Two Phase I/O) [3], where each I/O operation is separated into communication phases and I/O phases, for performing inter-process data exchange/reorganization and file I/O, respectively.

Mechanisms such as collective I/O are also important performance optimizations, for example, aggregating small, noncontiguous I/O requests into large, contiguous ones. The resulting long, sequential accesses are favored by hard disks, the dominant storage media used in secondary storage. Also, avoiding finely interleaved accesses alleviates the false sharing problem, as well as the consequent expensive locking/synchronization costs. In addition, parallel I/O libraries employ other performance optimizations, such as I/O aggregation (which reduces the number of processes that make I/O calls, which helps to reduce I/O contention in the face of large-scale runs where thousands or more processes are used to execute a parallel application), data sieving (which avoids small, non-contiguous I/O by in-memory data manipulations, either “picking out” useful data segments or masking holes in output through read-modify-write operations), and I/O hints (which pass information from the application to help the parallel I/O library in configuring its internal settings, such as available buffer size, aggregation parameters, and file consistency requirements).

To this date, the most popular parallel I/O libraries follow the MPI-IO specification, as a part of the MPI-2 Standard [23]. MPI-IO specifies the semantics of MPI-IO interfaces, which carry a unified style with familiar MPI calls for inter-process communication. While MPI-IO has been implemented with several MPI distributions, the most widely used and studied implementation is ROMIO [38], developed at the Argonne National Laboratory, as a part of MPICH, a popular open-source MPI library. ROMIO adopts the 2PIO design and adopts many collective I/O optimizations mentioned above.

## Scientific Data Format Libraries

Parallel I/O libraries such as ROMIO provide application programmers with the convenience and performance for coordinated, efficient multi-process I/O operations. However, many scientists running large-scale simulations and performing subsequent analysis/visualization of the computation results do not read/write data in plain binary formats. Instead, they often use high-level I/O libraries to create and access data in special scientific data formats. These file formats bring several important features. First, they produce self-explanatory and self-contained files that encompass both raw datasets (most commonly multidimensional arrays) and their accompanying metadata items. Such metadata describe attributes associated with the raw datasets, such as variable name, unit, type, array dimension and size, and other information needed to interpret and analyze the data. Second, they facilitate binary portability through automatic file conversion when the files are moved between architectures using different floating point presentation. This is important as scientific data files and programs need to be designed to survive generations of diverse HPC platforms. Finally, such data formats often support good file extensibility, allowing files to grow easily when datasets are expanded or more datasets are stored within a file.

Different scientific communities have different preferences over format choices, but more and more scientists and their applications converge to two most popular formats these days, HDF [12] and netCDF [24]. Both formats possess the aforementioned advantages, at the cost of performance: file access latency and bandwidth are often significantly inferior compared to those with binary files. In recent years, both formats have added support for parallel accesses, by developing parallel access interfaces on top of the MPI-IO parallel I/O library.

## Parallel I/O Today

Due to the unsolved and incoming challenges of delivering scalable, high-speed parallel I/O on large-scale supercomputers, there continue to be research and development efforts on parallel I/O system design and optimizations. In particular, the recent trend is going toward more integrated explorations and development, where application developers/users, I/O

middleware designers, supercomputer center managers and system administrators, file system designers, and finally supercomputer vendors gather together to participate in next-generation HPC I/O subsystem design.

One example of such coordinated design is the recent ADIOS parallel I/O middleware [17], which was developed jointly between Oak Ridge National Laboratory and Georgia Tech through tight interaction with state-of-the-art parallel scientific simulations. ADIOS is able to facilitate convenient result/checkpoint data output through intuitive group-based interfaces, with the help of an auxiliary XML configuration file. Also, ADIOS can perform aggressive asynchronous I/O as well as data staging operations [40]. This allows analytics tasks typically performed as post-processing steps to be carried out efficiently on the fly along the data output pathway, avoiding the need for multiple rounds of input and output. Another example is the close collaboration among designers across the many layers in the deep HPC I/O stack, such as the Lustre Center of Excellence that connects vendors and file system designers directly to supercomputing centers and application users.

Recently researchers and developers are also working on streamlining parallel I/O operations during parallel job runs with other pre-job and post-job data activities, such as data staging/offloading to/from supercomputing centers, long-area data migration, and data analytics. The traditional “in-job” parallel I/O becomes one link in the end-to-end scientific data workflow, while other components of the workflow have also been increasingly exploiting parallelism of various forms. Ongoing work is examining mechanisms and tools to allow part of these data processing/analysis tasks into the “in-job” data pathway. For example, long-desired features such as in-situ visualization not only completes important data post-processing, but also enables valuable runtime job steering capabilities.

Parallel I/O beyond the environment of supercomputing centers can potentially get more help from ongoing efforts such as the creation of pNFS (parallel NFS). There are also efforts underway to improve software reuse and data sharing, for example, through efforts to make popular scientific data formats such as netCDF and HDF more compatible with each other.

## Future Directions

Parallel I/O faces new challenges as well as new opportunities in the coming years, when supercomputing is moving from Peta-scale to Exa-scale and beyond.

With next-generation supercomputers, the computation parallelism will continue to rise and intensify the pressure on the I/O subsystem. The unprecedented high degree of I/O parallelism will be reflected in the number of processes making I/O calls and sharing files, the number of metadata operations to be processed concurrently, the number of disks and disk controllers, and the number/volume of data files.

On larger systems, parallel I/O challenges will be highlighted in several aspects. First, the increasing number of cores per node demands more scalable interconnection and storage device layers. Also, it is not clear whether the memory capacity and bandwidth would become a severe problem again, when shared among dozens or more cores within the same node. If true, this will make it harder to apply optimizations developed in the past decade to exploit unused memory to perform asynchronous I/O or intelligent client-side caching/prefetching. Second, with millions of disks to manage, software and hardware reliability will continue to be a problem in maintaining the overall system availability and usability. In particular, it is predicted that with extreme-scale systems, disk recovery through RAID coding will become a constant event and requires careful handling to avoid overall parallel I/O speed degradation. More elaborate metadata server and I/O node fall-over schemes may be needed too. Third, the large amount of data produced on extreme-scale systems need to be managed and processed. In particular, timely data movement and analysis tools will be crucial in the end-to-end scientific computing cycle. Finally, with the increasing system scale and complexity, it becomes ever more challenging to monitor and analyze parallel I/O performance. Effective benchmarking, tracing, and profiling mechanisms are needed for researchers to provision resources for large centers' storage subsystem, and to understand, configure, and optimize its performance in everyday operations. Performance visualization and automatic analysis/debugging tools will also facilitate the identification and solving of I/O

problems, particularly in the direction of cross-layer trace correlation and causal study.

One side problem of the increasing scale of supercomputers is that it becomes harder and harder for academic researchers to deploy and evaluate their parallel I/O design or improvements on production-scale platforms. In particular, server-side and kernel-level modifications often have to be constrained to experiments on small or medium research clusters, where the configuration and bottlenecks are drastically different from large-scale machines. The advances in cloud computing and virtual machine technologies, though, may enable these researchers to emulate supercomputer environments and perform large-scale experiments.

Meanwhile, there are also emerging opportunities for new practice or even paradigm shifts in parallel I/O systems and software in the future years and decades. One ongoing exploration is examining the feasibility and methodology for incorporating new storage devices into the HPC I/O hierarchy, as destination storage devices or intermediate caching layers. Examples of such devices of interest include Solid State Drives (SSDs) and phase-changing memory. These nonvolatile memory devices provide a compromise between capacity and speed between the main memory and secondary storage. Also, free of moving parts, these devices offer more reliable and energy-efficient I/O solutions compared to hard disks. However, there are unique challenges associated with deploying these new devices as the current I/O stack is designed based on hard disks' operation characteristics. Also, for HPC I/O, the write durability of devices such as SSDs may make their deployment costly, considering the write-intensive workloads on supercomputers. Scientific data centers, in this sense, may be better candidates for employing such read-friendly storage media for fast parallel I/O.

## Related Entries

► [MPI \(Message Passing Interface\)](#)

► [MPI-IO](#)

## Bibliographic Notes and Further Reading

For interested readers, John May authored a comprehensive book on parallel I/O for HPC [20], published in 2001. It gives a detailed overview of

HPC I/O hardware and software layers, as well as detailed discussions on standard parallel I/O interfaces (MPI-IO) and high-level scientific data libraries (netCDF and HDF5). A more recent overview of an HPC storage system instance is given by researchers at Oak Ridge National Laboratory [30], which describe the hardware, network, and software settings of the Spider parallel file system that is deployed at their computing center. Spider connects the storage hardware on several computing platforms centered around Jaguar, the Cray system that ranks as the world's largest supercomputer as of early 2010.

There have been several studies that investigate HPC I/O characteristics. In particular, a series of investigations on this topic were published in the 1990s [9, 10, 14, 35]. A recent study on Peta-scale I/O workloads was conducted by Argonne National Laboratory researchers [6].

A large amount of work has been conducted in parallel I/O performance optimization, especially on improving the performance of collective I/O. These optimizations include automatic I/O library parameter tuning [7], buffering/caching techniques [16, 18], file domain partitioning methods and handling of irregular/noncontiguous access patterns [4, 28, 37], exploiting overlap between computation and I/O [19, 23], and application-specific optimizations [26, 32].

The John May book mentioned above also discussed two specific uses of parallel I/O beside job input and result output: I/O in parallel out-of-core applications (to swap data between memory and disk due to limited main memory space), and checkpointing (to write out data periodically for fault tolerance and future restart). Several studies focused on parallel I/O for out-of-core codes (e.g., [25, 36]). Due to the increasing system size and per-node memory capacity, out-of-core applications are less common these days. For checkpointing, application-level parallel checkpointing has mostly shared characteristics with result output, though recent studies have explored new ways of performing checkpointing (such as in-memory checkpointing through file system interfaces [1] and checkpointing-specific storage design [2]).

## Bibliography

1. I-Kiswany S, Ripeanu M, Vazhkudai SS, Gharaibeh A (2008) std-chk: a checkpoint storage system for desktop grid computing. In: 28th IEEE International Conference on Distributed Computing Systems (ICDCS), Beijing, 2008, pp 613–624
2. Bent J, Gibson GA, Grider G, McClelland B, Nowoczynski P, Nunez J, Polte M, Wingate Meghan (2009) Plfs: a checkpoint filesystem for parallel applications. In: SC 2009, Portland, 2009
3. Bordawekar R, Rosario J, Choudhary A (1993) Design and evaluation of primitives for parallel I/O. In: Proceedings of Supercomputing '93, Portland, 1993
4. Broom B, Fowler R, Kennedy K (2001) KelpIO: a telescope-ready domain-specific I/O library for irregular block-structured applications. In: Proceedings of the 2001 IEEE International Symposium on Cluster Computing and the Grid, Brisbane, 2001
5. Carns P, Ligon W III, Ross R, Thakur R (2000) PVFS: a parallel file system for Linux clusters. In: Proceedings of the Fourth Annual Linux Showcase and Conference, Atlanta, 2000
6. Carns P, Latham R, Ross R, Iskra K, Lang S, Riley K (2009) 24/7 characterization of petascale I/O workloads. In: Proceedings of 2009 Workshop on Interfaces and Architectures for Scientific Data storage, New Orleans, September 2009
7. Chen Y, Winslett M (2000) Automated tuning of parallel I/O systems: an approach to portable I/O performance for scientific applications. IEEE Trans Softw Eng 26(4):363–383
8. Cluster File Systems, Inc. (2002) Lustre: a scalable, high-performance file system. <http://www.lustre.org/docs-whitepaper.pdf>, 2002
9. Crandall P, Aydt R, Chien A, Reed D (1995) Input/output characteristics of scalable parallel applications. In: Proceedings of Supercomputing '95, San Diego, 1995
10. Galbreath N, Gropp W, Levine D (1993) Applications-driven parallel I/O. In: Proceedings of Supercomputing '93, Portland, 1993
11. High Performance Fortran Forum (1994) High performance fortran language specification. Center for Research on Parallel Computation, Rice University, Houston, Nov 1994
12. HDF5 – A New Generation of HDF. <http://hdf.ncsa.uiuc.edu/HDF5/doc/>
13. Kotz D (1994) Disk-directed I/O for MIMD multiprocessors. In: Proceedings of the Symposium on Operating Systems Design and Implementation, New Orleans, November 1994
14. Kotz D, Nieuwejaar N (1994) Dynamic file-access characteristics of a production parallel scientific workload. In: Proceedings of Supercomputing '94, Washington, 1994
15. Lang S, Latham R, Kimpe D, Ross R (2009) Interfaces for coordinated access in the file system. In: Proceedings of the Workshop on Interfaces and Architectures for Scientific Data Storage, New Orleans, 2009
16. Liao WK, Ching A, Coloma K, Choudhary AN, Ward L (2007) An implementation and evaluation of client-side file caching for mpi-io. In: 21th International Parallel and Distributed Processing Symposium (IPDPS), Long Beach, 2007, pp 1–10
17. Lofstead JF, Klasky S, Schwan K, Podhorszki N, Jin C (2008) Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In: Sixth International Workshop on Challenges of Large Applications in Distributed Environments (CLADE), Boston, 2008, pp 15–24

18. Ma X, Lee J, Winslett M (2006) High-level buffering for hiding periodic output cost in scientific simulations. *IEEE Trans Parallel Distrib Syst* 17(3):193–204
19. Ma X, Winslett M, Lee J, Yu S (2003) Improving MPI-IO output performance with active buffering plus threads. In: Proceedings of the International Parallel and Distributed Processing Symposium, Nice, 2003
20. May J (2001) Parallel I/O for high performance computing. Morgan Kaufmann Publishers, San Francisco
21. Message Passing Interface Forum (1995) MPI: message-passing interface standard, June 1995
22. Message Passing Interface Forum (1997) MPI-2: extensions to the message-passing standard, July 1997
23. More S, Choudhary A, Foster I, Xu MQ (1997) MTIO: a multi-threaded parallel I/O system. In: Proceedings of the Eleventh International Parallel Processing Symposium, Geneva, 1997
24. NetCDF Documentation. <http://www.unidata.ucar.edu/packages/netcdf/docs.html>
25. Nieplocha J, Foster I (1996) Disk resident arrays: An array-oriented I/O library for out-of-core computation. In: Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation, Annapolis, Oct 1996, pp 196–204
26. Nieplocha J, Foster I, Kendall R (1998) ChemIO: high-performance parallel I/O for computational chemistry applications. *Int J Supercomput Appl High Perform Comput* 12(3):345–363, Fall 1998
27. Nieuwejaar N, Kotz D (1997) The galley parallel file system. *Parallel Comput* 23(4):447–476, 1997
28. No J, Park S, Carretero J, Choudhary A (2002) Design and implementation of a parallel I/O runtime system for irregular applications. *J Parallel Distrib Comput* 62(2):193–220
29. Nowicki B (1989) NFS: network file system protocol specification. Network Working Group RFC1094, 1989
30. Oral S, Wang F, Dillow D, Shipman G, Miller R (2010) Efficient object storage journaling in a distributed parallel file system. In: Eighth USENIX Conference on File and Storage Technologies, San Jose, 2010
31. Patterson D, Gibson G, Katz R (1988) A case for redundant arrays of inexpensive disks (RAID). In: Proceedings of the ACM SIGMOD Conference, Chicago, 1988
32. Ross R, Nurmi D, Cheng A, Zingale M (2001) A case study in application I/O on Linux clusters. In: Proceedings of Supercomputing '01, Denver, 2001
33. Schmuck F, Haskin R (2002) GPFS: a shared-disk file system for large computing clusters. In: Proceedings of the First Conference on File and Storage Technologies, Monterey, 2002
34. Seamons KE, Chen Y, Jones P, Jozwiak J, Winslett M (1995) Server-directed collective I/O in Panda. In: Proceedings of Supercomputing '95, San Diego, 1995.
35. Smirni E, Aydt R, Chien A, Reed D (1996) I/O requirements of scientific applications: an evolutionary view. In: Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing, Syracuse, 1996
36. Thakur R, Choudhary A (1996) An extended two-phase method for accessing sections of out-of-core arrays. *Sci Programming* 5(4):301–317, 1996
37. Thakur R, Gropp W, Lusk E (1999) Data sieving and collective I/O in ROMIO. In: Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation, Annapolis, Feb 1999
38. Thakur R, Gropp W, Lusk E (1999) On implementing MPI-IO portably and with high performance. In: Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems, Atlanta, May 1999
39. Top500 supercomputer sites. <http://www.top500.org/>
40. Zheng F, Abbasi H, Docan C, Lofstead J, Liu Q, Klasky S, Parashar M, Podhorszki N, Schwan K, Wolf M (2010) Predata – preparatory data analytics on peta-scale machines. In: 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Atlanta 2010

## iPSC

The iPSC machines were hypercube-connected multicomputers designed and built by Intel Corporation in the 1980s. Four models were developed: iPSC/1, iPSC/2, and iPSC/860

## Isoefficiency

### ► Metrics

## JANUS FPGA-Based Machine

RAFFAELE TRIPICCIONE

Università di Ferrara and INFN Sezione di Ferrara,  
Ferrara, Italy

### Definition

JANUS is a massively parallel application-driven machine, developed in the years 2006–2008 to support Monte Carlo simulations of spin-glass systems, a compute-intensive application in statistical physics. JANUS – fully based on FPGA technology – is a large array of processing elements that work under the supervision of a traditional host-machine; they are firmware-configured on-the-fly to run an application code developed in a hardware description language. Each processing element, when configured for spin-glass simulations, implements a massively-parallel architecture, midway between a graphic processing unit and a many-core CPU, in which several hundreds small processing cores concurrently run a SIMD application. For the specific application for which it was developed, JANUS offered approximately a 1,000 $\times$  performance gain over systems available at the time it was brought on line.

### Discussion

#### Overview

Application-driven computing systems have been used in many cases in computational physics in the last 20 years. These systems have been extremely successful in diverse compute-intensive areas, such as condensed matter simulations [1], Lattice QCD [2–4], and the simulation of gravitational systems [5]. The main reason for these successes has invariably been that the architectural requirements for the algorithms relevant for those problems were – at the time these machines were developed – at strong variance with those targeted by off-the-shelf CPUs or systems, so tailoring the architecture

to the algorithm yielded huge performance gains over commercial state-of-the art solutions. JANUS belongs to this class of machines: it was developed in the years 2006–2008, in order to simulate the behavior of glassy materials in condensed matter physics with Monte Carlo techniques; it has delivered – for that specific problem – performances more than three orders of magnitudes higher than possible with traditional systems available at the time it was developed. JANUS was developed by a collaboration of Italian and Spanish Universities (Università di Roma “La Sapienza,” Università di Ferrara, Universidad de Zaragoza, Universidad Complutense de Madrid, Universidad de Extremadura, and Instituto de Biocomputacion y Fisica de Sistemas Complejos (BIFI) in Zaragoza). The team had a strong and complementary background in the areas of physics of complex systems and in dedicated computer systems for lattice QCD.

A major challenge in condensed matter physics is the understanding of glassy behavior [6]. Glasses are materials of the greatest industrial relevance (aviation, pharmaceuticals, automotive, etc.) that do not reach thermal equilibrium in human lifetimes. This sluggish dynamics is a major problem for the experimental and theoretical investigation of glassy behavior, placing numerical simulations at the center of the stage. Glasses pose a formidable challenge to state-of-the-art computers: studying the surprisingly complex behavior governed by deceptively simple dynamics equations still requires inordinately long execution times.

Spin glasses (SG) are widely used theoretical models of disordered magnetic alloys widely regarded as prototypical glassy systems [7, 8]; the investigation of these systems today is largely based on Monte Carlo simulations (see e.g., [9] for a comprehensive review). The associated computational challenge is huge; in many SG models, the dynamical variables – so-called spins – are discrete and sit at the nodes of discrete  $d$ -dimensional lattices. In order to make contact with experiments, it is necessary to follow the evolution of a

large enough 3D lattice, say  $80^3$  sites, for time periods of the order of 1 s. Since one Monte Carlo step – the update of all the  $80^3$  spins in the lattice – roughly corresponds to  $10^{-12}$  s, some  $10^{12}$  such steps are needed, that is,  $10^{18}$  spin updates. Statistics on several ( $\approx 10^2$ ) copies of the system has to be collected, adding up to  $\approx 10^{20}$  Monte Carlo spin updates. Performing this simulation program in a reasonable time frame (say, less than 1 year) requires a computer system able to update on average one spin in approximately 1 ps. At the time JANUS was proposed, clever use of traditional systems allowed to reach simulation performances several hundred times lower.

In order to bridge this huge gap, a large amount of parallelism is available and easily identified in the associated algorithms (see later for some details); the JANUS architecture has been designed to carefully exploit at large fraction thereof, basically handling at the same time spin variables sitting at non-neighboring sites of the lattice. Each JANUS processor is able to process almost 1,000 spins in parallel; this immediately causes a major memory access bottleneck that has been circumvented by appropriate memory organization and access strategies and – at the technological level – by the use of embedded memory. Altogether, an intra-node degree of parallelism of order  $1,000 \times$  is compounded by a further factor ( $100 \times$ ) by trivially processing in parallel a corresponding number of copies of the system, so the required system-level performance has been reached.

Implementing this large degree of parallelism has been technologically possible because the data-path associated to the handling of each spin has limited logical complexity, being based on logic operations and integer arithmetics (as opposed to floating-point arithmetics) and because control can be readily shared across collaborating data-paths.

JANUS is fully based on Field Programmable Gate Array (FPGA) technology. FPGAs are inherently slow and trade flexibility with complexity (under general terms, any system of  $n$  gates can be emulated by an FPGA of complexity  $n \log_2 n$ , and  $\log_2 n$  is a “large” number when  $n \approx 10^6 - 10^7$ ). These disadvantages are offset by more dramatic speedup factors allowed by architectural flexibility. Also, development time of an FPGA-based system is short. Early attempts in this

direction had been made several years before by a subset of the JANUS team within the SUE project [10]. An ASIC approach, that would obviously further boost performance, was not considered by the JANUS developers as the longer development times and larger development costs made it a nonviable option.

## Spin Glasses

Basic ingredients of spin glasses (SG) are frustration and randomness (see Fig. 1). The dynamical variables, the spins, represent atomic magnetic moments (the elementary magnets whose combined effects make up the magnetism of the sample as a whole). Spins sit at the nodes of a crystal lattice (Fig. 1 – left), and take only two values,  $\sigma_i = \pm 1$ , corresponding to two possible orientations for the atomic magnet.

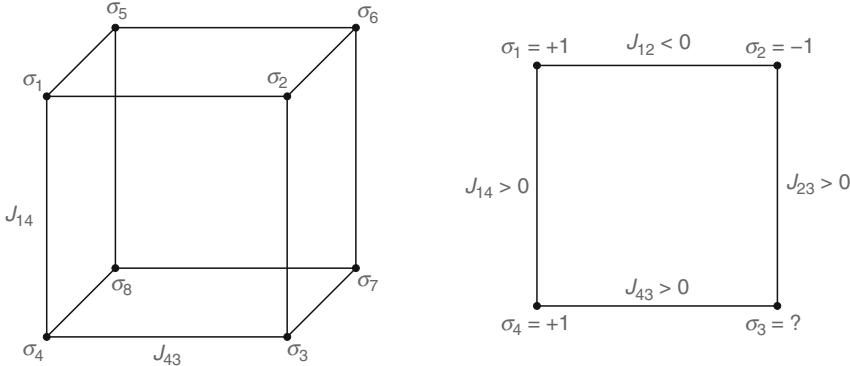
Depending on material details that wildly change from site to site, a pair of neighboring spins may lower their energy if they take the same value (and have therefore a tendency to do so). In this case, the *coupling constant*  $J_{ij}$ , a number assigned to the lattice link that joins spins  $\sigma_i$  and  $\sigma_j$ , is positive. However, it could happen with roughly the same probability that the two neighboring spins prefer to anti-align (in this case,  $J_{ij} < 0$ ).

A lattice link is *satisfied* if the two associated neighboring spins are in the energetically favored configuration. In spin glasses, positive and negative coupling constants occur with the same frequency, as the spatial distribution of positive or negative  $J_{ij}$  in the lattice links is random; this causes *frustration*. Frustration means that it is impossible to find an assignment for the spin values,  $\sigma_i$ , such that all links are satisfied (the concept is sketched in Fig. 1 – right, and explained in more details in the caption). For any closed lattice circuit such that the product of its links is negative, it is impossible to find an assignment that satisfies every link. In spin glasses, the probability that a given circuit is frustrated is 50%.

In a typical SG model, the Edwards–Anderson model [11], the energy of a configuration (i.e., a particular assignment of all spin-values) is given by the Hamiltonian function

$$H = - \sum_{\langle ij \rangle} \sigma_i J_{ij} \sigma_j, \quad (1)$$

where summation is taken on all pairs of nearest neighbor sites on the lattice. The coupling constants  $J_{ij}$  are



**JANUS FPGA-Based Machine. Fig. 1** *Left:* The spin-glass lattice is obtained by periodically repeating the unit cell. Dynamical variables, the spins, take values  $\sigma_i = \pm 1$ . Each pair of neighboring spins have either a tendency to take the same value or the opposite one. This is controlled by a coupling constant ( $J_{ij}$ ) attached to the lattice link joining the two spins.  $J_{43} < 0$  implies that the link is satisfied if  $\sigma_4 \cdot \sigma_3 < 0$  while  $J_{43} > 0$  demands that  $\sigma_4 \cdot \sigma_3 > 0$ . *Right* (frustration example): Consider the front plaquette of the cell at left, for the given values of the couplings, and try to find an assignment of the four spins such that all four links are satisfied. If one starts with  $\sigma_1 = +1$  and goes around the plaquette clockwise,  $J_{12} < 0$  requires that  $\sigma_2 = -1$  and then  $J_{23} > 0$  implies  $\sigma_3 = -1$ . On the other hand, going anticlockwise,  $\sigma_4 = +1$  (because  $J_{14} > 0$ ) and also  $\sigma_3 = +1$ , since  $J_{43} > 0$ ! Nothing changes if the initial guess is  $\sigma_1 = -1$ : also in this case conflicting assignments for  $\sigma_3$  are reached

chosen randomly to be  $\pm 1$  with 50% probability, and are kept fixed. A given assignment of the  $\{J_{ij}\}$  is called a *sample*. Some of the physical properties (such as internal energy density, magnetic susceptibility, etc.) do not depend on the particular choice for  $\{J_{ij}\}$  in the limit of large lattices. However, for the relatively small simulated systems it is useful to average the results obtained on several samples, that is, on different assignments of the  $\{J_{ij}\}$ .

It turns out that frustration makes it hard to answer even the simplest questions about the model. In three dimensions, experiments and simulations provide evidence that a spin-glass ordered phase is reached below a critical temperature  $T_c$ . In the cold phase ( $T < T_c$ ) the spins *freeze* in some disordered pattern, presumably related to the configuration of minimal energy.

For temperatures (not necessarily much) smaller than  $T_c$  spin dynamics becomes exceedingly slow. In a typical experiment one quickly cools a spin glass below  $T_c$ , then waits to observe the system evolution. As time goes on, the size of the domains where the spins coherently order in the (unknown to us) spin-glass pattern, grows slowly. Domain growth is sluggish: even after 8 h of this process, for a typical spin-glass material at a

temperature  $T = 0.72T_c$ , the domain size is only around 40 lattice spacings.

The smallness of the spin-glass ordered domains precludes the experimental study of equilibrium properties in spin glasses, as equilibration would require a domain size of the order of  $\sim 10^8$  lattice spacings. The good news is that an opportunity window opens for numerical simulations. In fact, in order to understand experimental systems, it is sufficient to simulate lattices not much larger than the typical domain size. This is the physics motivation behind the development of JANUS.

### Monte Carlo Simulations of Spin Glasses

Spin glasses are studied numerically with Monte Carlo techniques that ensure that the system configurations  $C$  are sampled according to the Boltzmann probability distribution

$$P(C) \propto e^{-\frac{H(C)}{T}}, \quad (2)$$

describing the equilibrium distribution of configurations for a system at constant temperature  $T$ .

One typical Monte Carlo algorithm is the Heat Bath algorithm, which assumes that at any time any spin has to be in thermal equilibrium with its surrounding

environment, meaning that the probability for a spin to take value +1 or -1 is determined only by its nearest neighbors, follows the Boltzmann distribution; the energy of a spin at site  $k$  of a 3D lattice of linear size  $L$  is

$$E(\sigma_k) = -\sigma_k \sum_{\langle km \rangle} J_{km} \sigma_m, \quad (3)$$

where the sum runs over the six nearest neighbors of site  $k$ . One can derive that the probability follows the Boltzmann distribution: in particular, the probability for the spin to be +1 is

$$\begin{aligned} P(\sigma_k = +1) &= \frac{e^{-E(\sigma_k=+1)/T}}{e^{-E(\sigma_k=+1)/T} + e^{-E(\sigma_k=-1)/T}} \\ &= \frac{e^{\phi_k/T}}{e^{\phi_k/T} + e^{-\phi_k/T}}, \end{aligned} \quad (4)$$

$$\phi_k = \sum_{\langle km \rangle} J_{km} \sigma_m, \quad (5)$$

where  $\phi_k$  is usually referred to as the *local field* acting on site  $k$ .

At the computational level, the corresponding algorithm takes the following steps:

1. Pick one site  $k$  at random.
2. Compute the local field  $\phi_k$  (Eq. 5).
3. Assign to  $\sigma_k$  the value +1 with probability  $P(\sigma_k = +1)$  as in Eq. 4; this can be done by generating a random number  $R$ , uniformly distributed in  $[0, 1]$ , and setting  $\sigma_k = 1$  if  $R < P(\sigma_k = 1)$ , as given by Eq. 4, and  $\sigma_k = -1$  otherwise.
4. Go back to step 1.

A full Monte Carlo step consists in the iteration of the above scheme for  $L^3$  times. Many Monte Carlo steps evolve the system toward statistical equilibrium.

Physical spin variables can be mapped into bits by the following transformations

$$\begin{aligned} \sigma_k &\rightarrow S_k = (1 + \sigma_k)/2, \\ J_{km} &\rightarrow \hat{J}_{km} = (1 + J_{km})/2, \\ \phi_k &\rightarrow F_k = \sum_{\langle km \rangle} \hat{J}_{km} \oplus S_m = (6 - \phi_k)/2, \end{aligned} \quad (6)$$

( $\oplus$  is the exclusive-or operation) so several (not all) steps in the algorithm involve logic (as opposed to arithmetic) operations.

The procedure is easily programmed for a computer; some comments are in order:

1. High-quality random numbers are necessary to avoid dangerous spurious spatial correlations

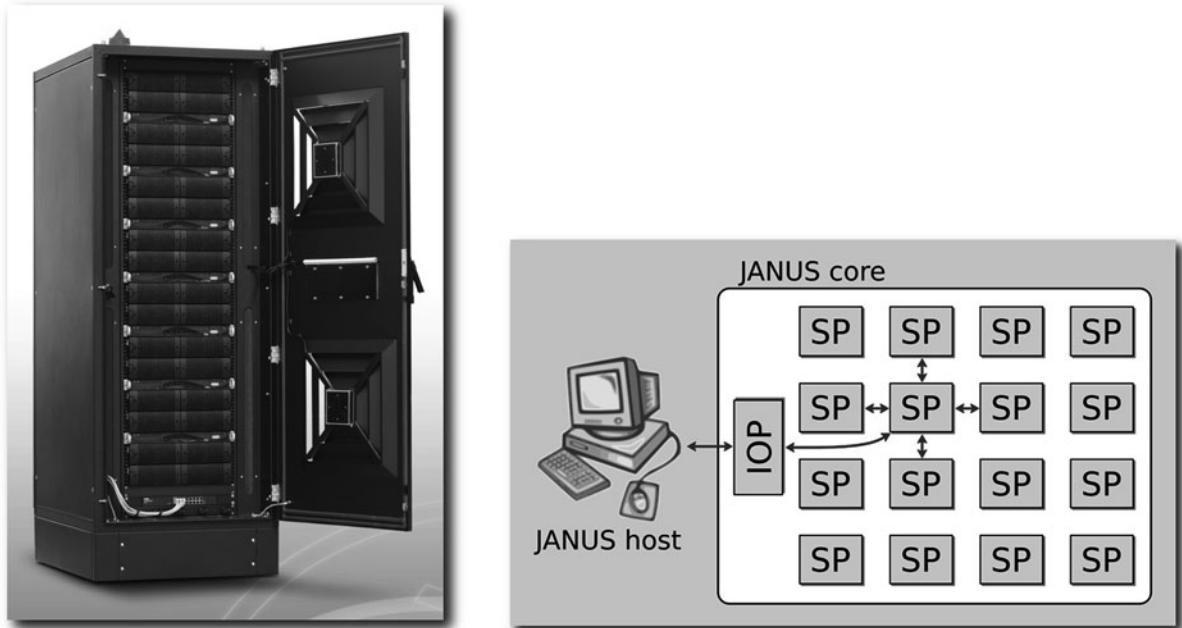
between lattice sites, as well as temporal correlations in the sequence of generated spin configurations.

2. The local field  $\phi_k$  can take only the seven even integer values in the range  $[-6, 6]$ , so probabilities  $P(\sigma_k = +1) = f(\phi_k)$  may be stored in a small look-up table.
3. The kernel of the program is the computation of the local field  $\phi_k$ , involving just a few arithmetic operations on discrete variable data.
4. Under physically reasonable assumptions, the simulation retains the desired properties even if Monte Carlo steps are implemented by visiting each lattice site exactly once, in any deterministic order, provided no two interacting spins are handled at the same time.
5. Several sets of couplings  $\{J_{km}\}$  (i.e., several *different samples* of the system) need to be generated; an independent simulation has to be performed for every sample, in order to generate properly averaged results.
6. One usually studies the property of a spin-glass system by comparing two or more statistically independent simulations of the *same sample*, starting from uncorrelated initial spin configurations (the two copies of a sample are usually referred to as *replicas*).

The last three points identify the parallelism available in the computation, which can be trivially exposed for points 5 and 6 above. For point 4 a more accurate analysis is required. In fact, one labels all sites of the lattice as *black* or *white* in a checkerboard scheme: all black sites have their neighbors in the white site set, and *vice versa*: one can then in principle perform the three steps of the algorithm on all white or black sites in parallel. These remarks identify the available parallelism in the algorithm. The next section describes the architecture that JANUS adopts in order to exploit as much as possible of this opportunity.

## JANUS: The Architecture

JANUS is a heterogeneous massively parallel system built on a set of FPGA-based reconfigurable computing cores connected to a host-system of standard processors. Each JANUS-core contains 16 so-called simulation processors (SP) and 1 input/output processor (IOP). The SPs are logically arranged at the vertexes of a 2D mesh and have direct low-latency high-bandwidth



**JANUS FPGA-Based Machine.** Fig. 2 *Left:* The JANUS system installed at BIFI, Zaragoza (Spain) has 16 JANUS-cores and 8 JANUS-hosts. *Right:* Simple block diagram of a minimal JANUS system, with one JANUS-host and one JANUS-core

communication links with nearest neighbors in a toroidal topology. Each SP is also directly connected with the IOP, while the latter communicates with an element of the host-system, as shown in the right side of Fig. 2. SPs and IOPs are based on Virtex4-LX200 FPGAs manufactured by Xilinx, that – at the time the system was developed – were the densest FPGA devices available from industry. The largest available JANUS machine (installed at BIFI – Zaragoza, since early 2008) has eight hosts (with a disk-system of 4 TBytes) and 16 JANUS-cores (Fig. 2, left side). More details on this machine are in [12–14].

The IOP handles the JANUS-host I/O interface, providing functionalities needed to configure the SPs for any specific computational task, to move data sets across the system and to control SP operations. The IOP processor exchanges data with the JANUS-host on two Gigabit-Ethernet channels, running a standard raw-Ethernet communication protocol. This means that JANUS can only be used in a very loosely-coupled mode with its host, as the JANUS-host bandwidth is rather low and the associated latency is large.

The SP is the computational element of the JANUS system. It contains uncommitted logic that can be configured via the IOP as needed to run

computing-intensive kernels. The SPs of each JANUS-core can be configured to run in parallel 16 independent programs, or to run 1 single program partitioned among the 16 FPGAs, exchanging data as appropriate across the nearest-neighbor toroidal network.

JANUS is programmed developing two different programs: a standard C program running on the host system that typically builds on a low-level communication library to exchange data to and from the SPs, and a code (written in a hardware description language, such as VHDL) defining the application running on the SPs.

In this approach, the FPGAs must be carefully configured and optimized for application requirements. Tailoring any specific application to JANUS is a lengthy and complex procedure, hopefully rewarded by huge performance gains. Obviously, high level programming frameworks able to automatically split an application between standard and reconfigurable processors and to generate VHDL code for the configurable segment would be welcome for this machine, but the tools currently available typically do not deliver the level of optimization needed for the high performance application run by JANUS; all application codes executed by JANUS so far have been handcrafted in VHDL.

It is critical for any successful JANUS implementation that as much as possible internal resources of the FPGA are exploited in a coordinated way (see [15] for a detailed description). It turned out that the underlying architecture of the Virtex4 FPGAs is almost optimal for the mix of computational operations involved in SG simulations, while other applications that were tried on the machine, especially in the area of optimization, e.g. graph coloring, did not have a comparable impact. So, JANUS, that had been advertised in earlier publications as a potentially general-purpose reconfigurable architecture, has been extensively used only for SG simulations.

Here just a few highlights of the architecture configured inside each SP node are given. The first issue that had to be addressed is memory organization, a potential performance bottleneck. LX200 FPGA devices come with many small embedded RAM blocks, which may be combined and stacked to naturally reproduce a 3D array of bits representing the 3D spin lattice. Memory banks embedded in the FPGA device are large enough to store the full simulation data structure, that – in turn – makes huge memory bandwidth possible. If one considers an SG simulation on a lattice of the size discussed earlier ( $80^3$  sites) one standard JANUS implementation configures embedded memory blocks with 80 bit width to represent the linear size of the spin lattice: 10 such memories with 10 bit addressing are enough to store the whole lattice. Addressing all memories with the same given address  $z$  allows to fetch or write a  $80 \times 10$  portion of an entire  $80^2$  lattice plane corresponding to spins with Cartesian coordinates ( $0 < x < 79, 0 < y < 9, z$ ). The next portion of the same plane is addressed by  $z + 80$ , referring to spins with coordinates ( $0 < x < 79, 10 < y < 19, z$ ), and so on. It is useful for performance to simulate two real replicas, so a good strategy allocates one such structure for *black* spins of replica 1 and *white* ones of replica 2. A further structure is allocated for white and black spins of replicas 1 and 2 respectively. A similar storage strategy applies to the read-only storage blocks associated to the  $J_{ij}$ .

After initialization, this firmware-enabled machinery fetches all neighbor spins (addresses  $z, z+1, z-1, z+80, z-80$ ) of an entire portion of plane of spins (identified by a single memory address  $z$ ) and feeds them to the update logic. The latter returns the processed (updated) portion of spins to be uploaded to memory

at the same given address. Altogether, 800 spins are updated simultaneously, so the update logic is made up by 800 identical update cells. Each cell receives the 6 nearest neighbor bits, 6 coupling bits, and one 32-bit random number; it then computes the local field, which is an address to a probability look-up table; the random number is then compared to the extracted probability value and the updated spin is output. Look-up tables are small 32-bit wide memories instantiated as *distributed RAM*. There is one such table for each update cell. Random number generators – one to each update cell – are implemented as 32-bit Parisi-Rapuano generators [16], requiring one add and one bit-wise xor-operation for each random value.

The typical number of 800 updates per clock cycle is a trade-off between the constraints of allowed RAM-blocks configurations and available logic resources for the update machinery. This internal architecture is very similar in structure to modern graphics processing units (GPUs) or many-core CPUs: there is a large number of independent computational data paths that are fed by memory available on board. The specific application can be arranged as just one large SIMD thread, so control is shared by all data paths. This saves precious logic that is allocated to more data path instantiations; furthermore the logical complexity of each data path is significantly smaller than typical CPUs: these two points are the key to the huge performances of this system.

Total resource occupation for the design is 75% of RAM-blocks (total available RAM is  $\approx 672$  KBytes for each FPGA) and 85% of logic. The system runs at a conservative clock frequency of 62.5 MHz. At this frequency, power consumption for each SP is  $\approx 35W$ . It is interesting to note that the FPGA would operate correctly at a much higher clock rate, and the current upper limit in frequency is set by the performance of the forced-air cooling systems, a reminder that power management is becoming more and more a key performance bottleneck in high performance computing.

## JANUS Performance

JANUS performance can be assessed in at least two different ways:

- The number of effective operations per second
- The speed-up factor (mostly in wall clock time, but also on other relevant metrics) with respect to pro-

cessor clusters or “arbitrary” size (i.e., assuming that, for the given simulation, the optimal number of processor is actually deployed)

Let us first consider the number of effective operations performed by second. At each clock cycle (clock frequency is 62.5 MHz), each SP processor updates 800 spins. On a traditional architecture, this would imply at least the following instructions for each spin:

- Six loads
- *1 sum (32 bits)*
- *2 xor (32 bits)*
- *6 sum (3 bits)*
- *6 xor (3 bits)*
- *1 load LUT*
- *1 comp (32 bits)*
- Six updates of address pointers (1 mult, 1 sum )
- One store
- Jump condition

If one wants to count only algorithm-relevant operation, all load/store and address instructions must not be counted; also, the 6 xor and sum operations on short operands may be considered equivalent to one standard arithmetic operation. All in all, there are seven equivalent instructions for each spin update (shown in italic in the list above). This translates into a processing power of  $7 \times 800 \times 62.5 \times 10^6$  operations per second, which is 350.0 Giga-ops. The largest JANUS machine operates 256 processors in parallel, so it performs at the level of 89.6 Tera-ops.

It is also interesting to compare JANUS performances with those obtained on commercial CPUs, where spin-glass simulations had been performed earlier.

The main reason why SG simulations are not very fast on traditional CPUs is that the natural physical variables for the problems are bits, while CPU architectures are based on long (e.g., 32–64 bit) words. A widely used trick to boost performance (usually called multi-spin coding) uses the bits of the CPU word to code the same spin of a matching number of systems, treated in parallel by the algorithm. Using this technique, carefully optimized codes running on high-end CPUs available in 2006–2008 were able to handle on average one spin

in slightly less than 1 ns. By contrast, the JANUS implementation performs at 0.020 ns/spin. This means that  $10^{12}$  Monte Carlo steps for a system of  $80^3$  sites take approximately 100 days wall clock time, while a similar simulation on PCs would take of the order of 10 years: in conclusion, JANUS has made it possible for the first time to stretch SG simulations to experimentally relevant time scales [17]. JANUS has excellent performance also from the point of view of power requirements: each node uses  $\approx 35\text{W}$  of power, which translates into  $\approx 10$  Giga-ops/W.

## Future Directions

JANUS has marked a new standard in the simulation of complex discrete systems by allowing studies that would take years on traditional computers. The system has a very large sustained performance, an outstanding price–performance ratio, and an equally good power–performance ratio. Physics results obtained analyzing simulation results delivered by JANUS have been published in several journals, and the system is still on line, with no plans to phase it out in the near future.

From the point of view of a computer system, JANUS has shown the potential performance impact of reconfigurable computing and of many core architectures.

- From the former point of view, JANUS strongly leveraged on the very regular structure of the target algorithm: a highly optimized code carefully matching the low-level underlying FPGA architecture was developed in less than three months with reasonable human effort; on the other hand attempts with automatic VHDL generators (starting, e.g., from C programs) yielded no more than a few percent of the performance of the handcrafted code, even in JANUS almost ideal conditions: clearly, reconfigurable computing will not have a significant impact till this gap is substantially narrowed.
- As a prototypical many-core system, JANUS can be seen as an extreme version of the architecture adopted in very recent years by General Purpose GPUs (GP-GPUs); in both cases a very large number of computational threads is handled by a large set of computing cores, clustered in SIMD-controlled partitions. JANUS pushes to the extreme this trend

by making the cores very simple and by arranging just one SIMD partition. Huge performance are expected by this architecture, but memory access becomes rapidly the critical bottleneck. JANUS has solved this problem using a large number of embedded memory banks and carefully arranging data structures on them.

As a technological trend, GPGPU in the near future will probably still boost performance for floating-point based performance. JANUS is still on the leading edge for the specific mix of logic/integer-arithmetic operations associated to spin-glass systems.

## Bibliographic Notes and Further Reading

A comprehensive review on the class of physics problems at the basis of the development of JANUS can be found in [6], while [9] describes the relevant algorithms. A discussion of application-driven systems developed over the years for theoretical physics applications is available in the papers [2, 3, 5]. A recent special issue of Computing in Science and Engineering has covered this area in details [18].

## Bibliography

1. Condon JH, Ogielski AT (1985) Rev Sci Instrum 56:1691–1696; Ogielski AT (1985) Phys Rev B 32:7384–7398
2. Boyle PA et al (2005) IBM J Res Dev 49(2/3):351–365
3. Belletti F et al (2006) Comput Sci Eng 8(1):18–29
4. Goldrian G et al (2009) Comput Sci Eng 10(6):46–55
5. Makino J et al (2000) A 1.349 Tflops Simulation of Black Holes in a Galactic Center on GRAPE-6. In: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, Dallas, Article no. 43
6. See for instance Angell CA (1995) Science 267(5206):1924–1935; Debenedetti PG (1997) Metastable liquids. Princeton University Press, Princeton; Debenedetti PG, Stillinger FH (2001) Nature 410:259–267
7. Mydosh JA (1993) Spin glasses: an experimental introduction. Taylor and Francis, London
8. Young AP (ed) (1998) Spin glasses and random fields. World Scientific, Singapore
9. Amit DJ, Martin-Mayor V (2005) Field theory, the renormalization group and critical phenomena, 3rd edn. World Scientific, Singapore
10. Pech J et al (1997) Comput Phys Commun 106(1–2):10–20; Cruz A et al (2001) Comput Phys Commun 133(2–3):165–176
11. Edwards SF, Anderson PW (1975) J Phys F: Metal Phys 5:965–974; (1976) 6:1927–1937
12. Belletti F et al (2007) IANUS: Scientific Computing on an FPGA-based Architecture. In: Proceedings of ParCo2007, Parallel Computing: Architectures, Algorithms and Applications, NIC Series Vol. 38, pp 553–560
13. Belletti F et al (2006) Comput Sci Eng 8(1):41–49
14. Belletti F et al (2008) Comput Sci Eng 11(1):48–58
15. Belletti F et al (2008) Comput Sci Eng 178(3):208–216
16. Parisi V, cited in Parisi G, Rapuano F (1985) Phys Lett B 157(4):301–302
17. Belletti F et al (2008) Phys Rev Lett 101:157201
18. Gottlieb S (guest editor) (2006) Comput Sci Eng 8(3)

## Java

### ► Deterministic Parallel Java

## JavaParty

MICHAEL PHILIPPSEN

University of Erlangen-Nuremberg, Erlangen, Germany

## Definition

While Java offers both a means for programming threads on a shared-memory parallel machine and a means for programming remote method invocations in a wide-area distributed memory environment, it lacks support for NUMA-architectures or clusters, that is, architectures where locality matters because of nonuniform access times. Java also lacks support for a more data-parallel programming style.

JavaParty fills this gap by providing language extensions that add both *remote objects* and *collectively replicated objects*. The former allow the programmer to express knowledge about the locality of both the application's objects and threads in an abstract way, that is, without requiring code for low-level object placement, for explicit replication and (cache) coherency, or for low-level details of explicit message passing. The latter provide a means for working elegantly on all objects of an irregular data structure in parallel.

The beauty of JavaParty is twofold. First, the flavor of all the added language features fits nicely into Java. Second, since a preprocessor transforms the language extensions back into regular Java code, JavaParty programs run anywhere, that is, for running JavaParty

programs it is sufficient to start a regular JVM (Java Virtual Machine) for example on each (multi-core) node of a cluster.

## Discussion

### Remote Objects

#### The Concept

JavaParty class definitions may have a new label `remote` that is implemented either as a new modifier or as an annotation. An instance of a class with label `remote` is a *remote object*. It appears like regular Java objects, but they may live in remote address spaces, that is, by making a class `remote` the programmer explicitly expresses that it is acceptable if access to its instance variables or invocation of its methods may be slower than accesses to regular (local) Java objects. A remote thread is tied to an object in Java. The thread will be spawned on the remote node that hosts the thread object. JavaParty extends Java's object model to a distributed object model. Conceptually, a remote object is implemented by a local proxy object (stub) that forwards all accesses to a remote JVM that hosts the remote object. The proxy objects are generated automatically by the preprocessor, see below; they are transparent to the JavaParty programmer.

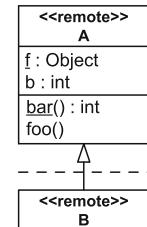
In JavaParty, garbage collection of remote objects works like that for regular (local) Java objects. Remote objects are also transparent with respect to synchronization, that is, a thread that holds a lock on a remote object may enter another synchronized method of that object even if the flow of execution has moved between different address spaces and JVMs. Static fields of a regular (local) Java class exist only once for all objects of that class. The same holds for static fields of a remote JavaParty class: all accesses to its static fields appear like in regular Java although the static fields can reside on a cluster node that may be different from the node that allocates the instance fields of a remote object. Conceptually, the static fields of a remote object are reached through another (automatically generated) proxy object that is also transparent for the programmer.

#### Transformation

From the two `remote` JavaParty classes shown in Fig. 1, the JavaParty preprocessor generates several pure Java

```
remote class A {
 static Object f;
 int b;
 static int bar() {...}
 void foo() {...}
}

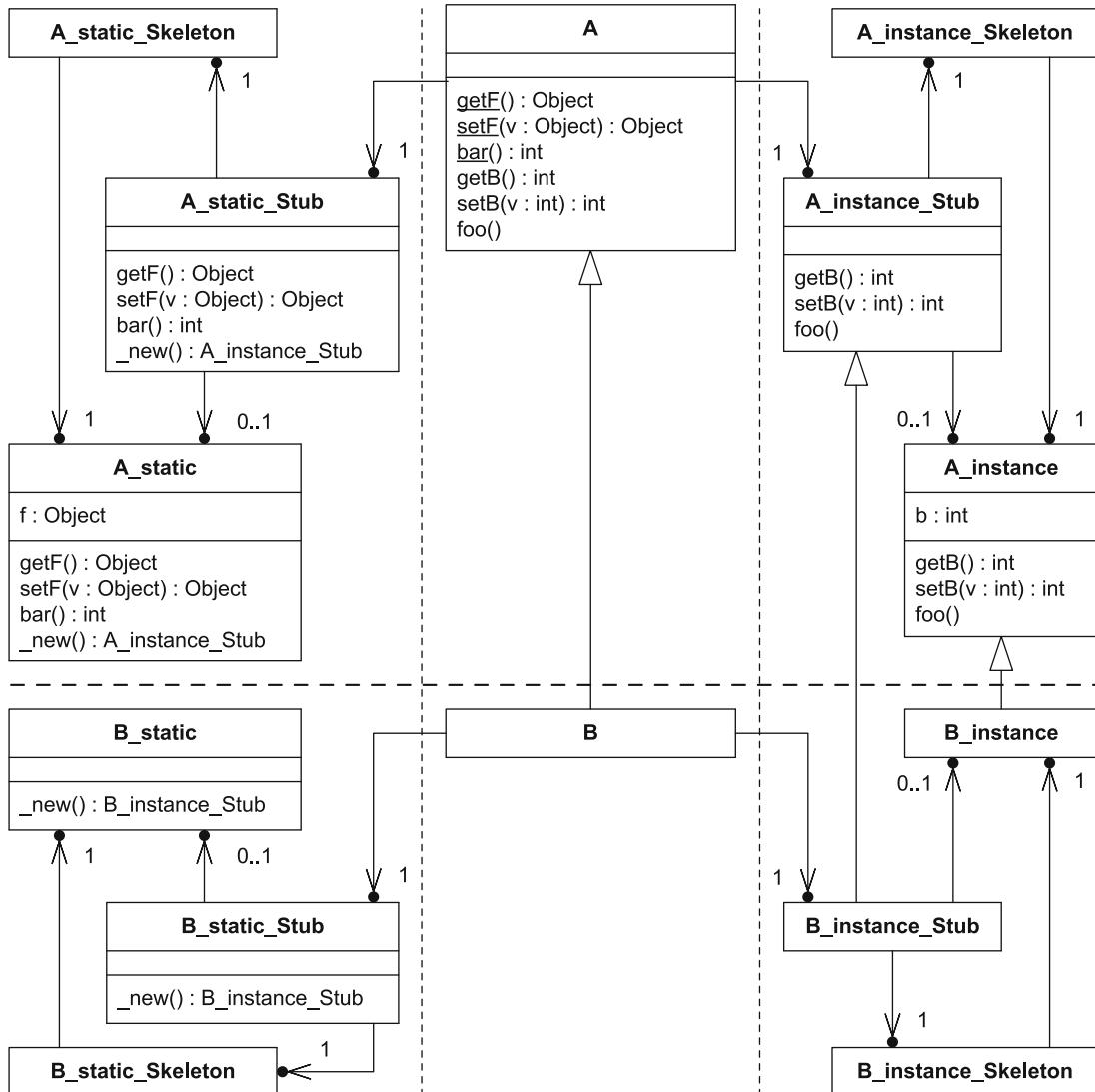
remote class B extends A {...}
```



**JavaParty, Fig. 1** Two `remote` JavaParty classes before transformation

classes. The left column of Fig. 2 depicts the classes that implement static/class fields and static/class methods, that is, you will find implementations for `f` and `bar` in `A_static`. All the instance fields and methods are implemented in class `A_instance` in the right column. On both sides, variables are replaced by setter and getter methods. At runtime, there might be several instances of class `A_instance`, one for each object of type `A` that the application creates. Since all objects might live on remote nodes, JavaParty also generates a stub and a skeleton (interface) for each of the two classes, so that it fits to Java's RMI implementation of remote methods. To create an instance of `A_instance` on a remote node, we need some way of calling a constructor on that node. Since Java's RMI does not allow this, JavaParty generates an extra method `_new` (for each of `A`'s constructors) that is called instead and that returns a reference to a remote object (or to its proxy to be more specific). Hence, at runtime there is one object of type `A_static` on each node, just to provide the constructors for creating all the necessary `A_instance` objects. But only one of these class objects actually implements the static fields and methods of `A`.

In addition to the classes that implement the instance parts and those that implement the static parts, a `remote` JavaParty class is replaced by a so-called handle class. The handle classes for the above example are shown in the middle column of Fig. 2. These classes provide the original constructors (that will call the `_new` methods), they bundle the references to the two remote objects of types `A_static` and `A_instance`, they deal with exceptions that might be caused by remote method invocations, they hide migrating remote objects, they shortcut methods calls if a potentially remote object happens to reside on



**JavaParty. Fig. 2** Simplified result of JavaParty's remote class transformation

the same node, they are used for `instanceOf` type comparisons, etc. For details and other aspects of the transformation, for example, exception handling, some additional methods, name mangling, the transformations within non-remote classes that access remote classes, etc., please check the bibliographic notes.

### Copy Semantics for Local Object Arguments in Remote Method Invocations

The main difference between regular/local Java objects and remote objects is the way arguments are passed.

For method invocations on local objects, Java copies references to local objects that are passed as arguments. In contrast, for a method invocation of a remote JavaParty object, the complete local object referred to and all the regular/local Java objects that can be reached transitively from that object are copied and shipped. This is of course necessary if the remote object resides on a different node of a cluster on which the addressed local objects are not present. For consistency, object cloning is performed even if a remote object happens to reside locally in the same JVM. There is no semantic difference for primitive type arguments. References to remote

objects are treated as such, that is, remote objects are not copied if references to them are passed along in method invocations.

## Object Distribution

The JavaParty runtime system uses heuristics to decide on which actual node of a cluster to allocate a newly created remote object and on which cluster node to execute a newly spawned remote thread. These heuristics may be guided by the results of a previous static analysis. The runtime system can automatically and transparently migrate remote objects to improve access times. If needed, the programmer can use library calls to select specific cluster nodes, to pin remote objects, or to tie a set of remote objects together so that the automatic (or manual) migration affects all of them at once. The flow of execution automatically follows the data which are being manipulated, that is, a thread executes an object's method on the node that hosts the object. Explicit or manual migration of thread objects can be used to enhance locality, since thread objects that reside at the place of the accessed objects will be faster.

## Collectively Replicated Objects

JavaParty adds `replicated` as another new class modifier/annotation for replicated objects. Whereas a remote object conceptually resides on a single remote node, a copy of a replicated object resides on *every* remote node (unless the degree of replication is restricted). Whereas a remote object is accessed remotely through a proxy object, a replicated object can only be accessed locally since it is implemented by one regular (local) Java object on each node. There is no proxy object for remotely accessing a replicated object. When an object of a replicated class is created, as its initial distribution, a copy of it with all its instance variables is created on every node of the underlying parallel computing system. Threads access this copy, called *replica*, locally on their node as they would access regular (local) Java objects, that is, there is no runtime overhead in accessing a replica.

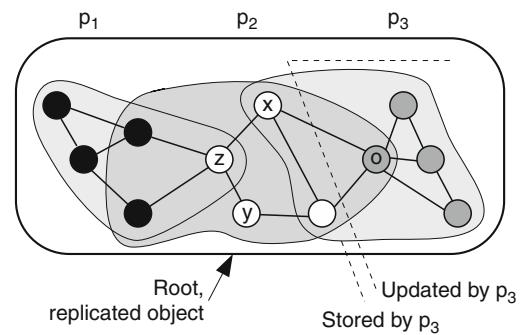
As mentioned above, JavaParty transitively clones the whole graph of referred to objects when a local argument object is shipped in a remote method invocation. Similarly, all the objects transitively referred-to from a replicated object are cloned and also exist on every node. Consider a replicated object that holds references

to a graph of local objects. Then conceptually, a copy of this object graph exists on every node.

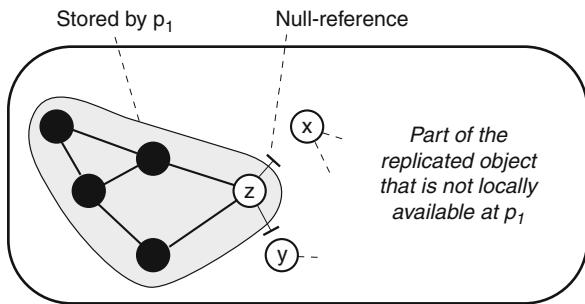
The JavaParty runtime system can choose to use a partial replication and to cut out those parts of the object graph that are not used on a particular node. This can be done automatically based on some static analysis. Alternatively, the JavaParty programmer can again use library calls to indicate cutoff objects. As an example consider the situation depicted in Fig. 3.

Here, three threads  $p_1 - p_3$  work on a replicated object graph, one thread per cluster node. Although conceptually a copy of the whole object graph is stored on all three nodes, each of the worker threads updates only a few objects (the black, white, or gray circles, respectively) based on some information in neighboring objects. A full replication that stores the whole object graph per JVM is not required. A partial replication is sufficient and more efficient with respect to memory consumption. It suffices that each JVM only keeps copies of those objects that are shown in the shaded areas. For instance, on the first JVM, where thread  $p_1$  is executed, it is sufficient to keep the part of the wire frame shown in Fig. 4.

As will be discussed below, JavaParty automatically keeps replicas in a consistent state, that is, a deep comparison of all the local objects that can be reached through the replicated object's fields must show identical values in the fields of primitive types. References to local objects may, however, be set to `null` on some nodes due to the decision to only partially replicate the data, while on other nodes the graph traversal will reach regular objects. For the example object graph, the JVM of  $p_1$  could store only a part of the replicated



**JavaParty. Fig. 3** Replicated data structure



**JavaParty. Fig. 4** Replica of node  $p_1$  in case of partial replication

object graph. Figure 4 illustrates the null references at the cutoff object  $z$ .

### Memory Model

Similar to Java where a thread is only guaranteed to see the changes made by different threads if both threads have been subject to an explicit mutual synchronization, nodes can work on their replica in isolation, that is, without an explicit synchronization no other node is ever guaranteed to see a change.

With replicated objects, a single type of synchronization that guarantees exclusive access to an object is no longer sufficient since the main reason for replication is to allow concurrent access to locally available replicas (with as little overhead as possible). Hence, JavaParty adds a concurrent read, exclusive write (CREW) locking for replicated objects:

```
shared synchronized(p) { //reader
 ...
 //read access to replica
}
...
exclusive synchronized(p) { //writer
 ...
 //update replica and notify
 ...
 //about change
}
```

This language extension allows concurrent threads in a *shared block* as long as there is no thread in an *exclusive block* at the same time. Synchronized methods can also be flagged as *shared* or *exclusive*. As usual, at some point in time, after a modifying thread leaves the exclusively locked block and before another thread enters some synchronized block on the same object, all changes must be made available to the second thread.

At the end of the exclusive block, changes are broadcast to all nodes that hold copies of the replicated object, so that other threads see the updated values, that is, a consistent state, as soon as they enter another synchronized block.

The JavaParty compiler generates code for the necessary consistency messages. This code extends Java's regular memory model to a distributed environment. In regular Java, changes made by one thread in a synchronized block are only guaranteed to be visible to another thread if the later enters a synchronized block itself. Java's `wait` and `notify` are also transformed into communication operations so that they work on replicated objects as expected even though the signaling needs to span node boundaries.

### Bulk Synchronous Collective Synchronization

JavaParty's replicated objects are called collectively replicated objects because there is a third flavor of synchronization, called *collective synchronization*. It is indicated by means of the modifier `collective`. In contrast to shared and exclusive synchronization, on every node that carries a replica exactly one thread must enter a collectively synchronized block of code. Together these threads form a *collective* that cooperatively work on the replicated state. In the above example, three threads will enter a collectively synchronized block of code to work on the replicated object graph, each on its portion of the replicated state, to collectively perform an updating step. After acquiring a collective lock, each thread is allowed to modify its portion of the replicated object's state, while concurrently, other threads modify their portions. Within the synchronized block, that is, while working on its portion of the replicated data, a thread does not see the updates performed by other threads. Only at the synchronization barrier of lock release (but not earlier), all the modifications are merged and are then made available on all the nodes. Conceptually, the end of a collective block is a synchronization barrier combined with an all-to-all broadcast that brings replicated objects back into a consistent state.

In the above example, both threads  $p_1$  and  $p_2$  can access the white object  $z$  – each thread on its own node. If either thread changes some fields of  $z$  within a collectively synchronized block, the other threads will not

see these changes immediately. Due to the bulk synchronous paradigm, the modifications are only shipped when all nodes leave the collective block. If both  $p_1$  and  $p_2$  have modified different instance fields of  $z$  concurrently, only the modified fields are exchanged. It is in general considered to be a program error if the modified portions of the replicated state overlap, that is, if both  $p_1$  and  $p_2$  alter the same instance field of  $z$  to different values. In this case it is unspecified which of the two values survives. JavaParty also provides a means of specifying merging routines that handle such concurrent updates of a single instance variable.

A naive implementation of a collective synchronization could broadcast all-to-all, diff and merge the replicated data structure. However, an efficient implementation, only ships the modified instance fields, and only ships them to those nodes that actually store the objects in case of partial replication.

## Related Entries

- ▶ [BSP \(Bulk Synchronous Parallelism\)](#)
- ▶ [PGAS \(Partitioned Global Address Space\) Languages](#)
- ▶ [Software Distributed Shared Memory](#)

## Bibliographic Notes and Further Reading

Many parallel object-oriented languages have been proposed in the past, see for example [1, 5]. JavaParty extends Java with a remote object model for distributed environments with nonuniform memory access and adds collective replication for data-parallel computation on irregular data structures in the style of bulk synchronous parallelism (BSP [8]). JavaParty runs on top of a set of regular Java virtual machines (JVMs). Alternatively, a JavaParty implementation could target a VM that spans a cluster and combines local memories into a virtual huge address space, see for example [9].

The most cited paper on JavaParty [7] covers the technical details of the transformation of remote classes into regular Java classes plus proxies and RMI invocations. There are several papers on implementation aspects that are essential to make JavaParty run fast. The most important ones are the following: The fast remote method invocation and object serialization, that is, the Karlsruhe RMI drop-in replacement for Java's remote method invocation KaRMI, is presented in [6]. Locality optimization and automatic placement

of threads and objects to reduce communication costs is addressed in [2]. JavaParty's distributed garbage collector is covered in [4]. How collective replication, especially the collective update operation at the end of collective synchronization blocks, can be implemented efficiently is covered in [3].

## Bibliography

1. Bal H, Haines M (1998) Approaches for integrating task and data parallelism. *IEEE Concurrency* 6(3):74–84 (July–September)
2. Haumacher B, Philippsen M (2000) Locality optimization in JavaParty by means of static type analysis. *Concurrency: Practice Experience* 12(8):613–628 (July)
3. Haumacher B, Philippsen M, Tichy WF (2010) Irregular data-parallelism in a parallel object-oriented language by means of collective replication. Tech. Rep. CS-2010-04, University of Erlangen-Nuremberg, Dept. of Computer Science (February)
4. Philippsen M (2000) Cooperating distributed garbage collectors for cluster and beyond. *Concurrency: Practice Experience* 12(7):595–610 (May)
5. Philippsen M (2000) A survey of concurrent object-oriented languages. *Concurrency: Practice Experience* 12(10):917–980 (August)
6. Philippsen M, Haumacher B, Nester C (2000) More efficient serialization and RMI for Java. *Concurrency: Practice Experience* 12(7):495–518 (May)
7. Philippsen M, Zenger M (1997) JavaParty - transparent remote objects in Java. *Concurrency: Practice Experience* 9(11):1225–1242 (November)
8. Valiant LG (1990) A bridging model for parallel computation. *Commun ACM* 33(8):103–111 (August)
9. Veldema R, Philippsen M (2008) Supporting huge address spaces in a virtual machine for Java on a cluster. In: Proc LCP'07, the 20th Intl. Workshop on Languages and Compilers for Parallel Computing (Urbana, IL, October 11–13, 2007). Lecture Notes in Computer Science, vol 5234, pp 187–201. Springer, Berlin

## Job Scheduling

ROLF RIESEN<sup>1</sup>, ARTHUR B. MACCABE<sup>2</sup>

<sup>1</sup>IBM Research, Dublin, Ireland

<sup>2</sup>Oak Ridge National Laboratory, Oak Ridge, TN, USA

## Synonyms

[Node allocation](#); [Processor allocation](#)

## Definition

A parallel job scheduler allocates nodes for parallel jobs and coordinates the order in which jobs are run. With

enough resources available, a system can execute multiple parallel jobs simultaneously, while other jobs are enqueued and wait for nodes to become available. The job scheduler manages the queues of waiting jobs and oversees node allocation. The goals of a scheduler are to optimize throughput of a system (number of jobs completed per time unit), provide response time guarantees (finish a job by a deadline), and keep utilization of compute resources high.

## Discussion

### Introduction

Users of a parallel system submit their jobs by specifying which application they would like to run and how many nodes they need. It is then the task of the job scheduler to find and allocate the appropriate number of nodes. This is different from a sequential system where the Operating System (OS) is responsible for scheduling processes and assignment of them to processor cores. The difference is that sequential processes are pursuing independent goals and are not working together, in parallel, to complete a common task. There is some cost associated with moving a process to another processor core, but in general the OS can assign processes to the least busy core without much concern for other processes.

Parallel jobs have different requirements, and they often run on multicomputers where each node, consisting of one or more processors and memory, runs its own copy of the OS. Parallel jobs cannot make progress until all processes of the job are assigned to CPUs. Only when these processes execute concurrently, are the resources used efficiently and the job finishes as quickly as possible. For example, if one process in a parallel job is waiting for data from another process, it cannot continue until the other process has run and computed the result that is needed by the first process. For that reason, it is important that a parallel scheduling system allocates and schedules all processors that are part of the same job, at the same time. Although a parallel job scheduler has to coordinate with the OSs running on each node, it is usually an independent software component that is not built into the OS kernel.

A node in a parallel system consists of one or more processors, each with one or more CPU cores. These cores share a network interface to communicate with

processes running on cores of other nodes. Some job schedulers allocate individual cores, but it is more common to allocate all cores on a node as a unit.

A job scheduling system, in coordination with the local OSs, ensures that a node and its processor cores are not overburdened, by limiting the number of processes assigned to each node at any given time. Therefore, a parallel job sometimes has to wait until enough nodes are available to run it. The job scheduler maintains queues to keep track of waiting jobs. When a job finishes, the nodes it was using are released and the scheduler selects the next job to run. Since each job needs a different number of nodes, assigning the available nodes to the waiting jobs is not trivial. It is sometimes necessary to leave nodes unallocated and combine them with nodes that become available later, so a larger job can run.

Managing the running and waiting jobs is done with several goals in mind. The highest priority goal is often throughput. The number of jobs a system can process in a unit of time is its throughput. On some systems, at least for some of the jobs, response time is more important. A simulation that predicts tomorrow's weather needs to finish in time for today's news. If its execution is delayed for too long, it will not meet its deadline. Another aspect of scheduling is resource utilization. Nodes should not be idle while there are still jobs to complete.

There are two main aspects to parallel job scheduling. First, the job scheduler needs to decide when a particular job needs to run. In addition to ensuring that enough nodes are available, possibly by reserving them, the scheduler may also take into consideration the priority of a job and how long it has been in the queue. The second part of scheduling is node allocation. When there are more nodes available than needed by the next job, the scheduler needs to decide which nodes should be allocated. In heterogeneous systems, node allocation may need to take into account available processor speeds and type, as well as memory sizes and other specific resources, and match them with the requirements of a job. Even in homogeneous systems, where all nodes have the same capabilities, the allocation strategy is important because it impacts performance of the system. Nodes that are located physically close to each other have lower communication latencies, and it is less likely that other messages and I/O traffic traversing the

system will delay messages between these nodes. The following sections discuss scheduling and allocation in more detail.

## Scheduling

When nodes become available, the scheduler needs to pick which job to run next. The easiest way to do that is to simply pick the next job in the queue. If that job needs more nodes than are currently available, the scheduler waits until more of the currently running jobs finish and enough nodes become available. This is called First-Come First-Serve (FCFS) scheduling.

FCFS is inefficient because it leaves nodes idle, while smaller jobs further back in the queue could be run. A method called *backfill* solves that problem. When nodes are available, but the next job in the queue is too big to fit, the scheduler searches further back into the queue to find a job that can be run on the available nodes. This can lead to *starvation* of bigger jobs, if only a few nodes are available at a time. Smaller jobs in the queue must be prevented from always bypassing the larger jobs. Methods to prevent starvation usually assign a priority to each job, or use multiple queues with different priorities. While jobs are waiting to run, their priority is increased periodically or they are moved to higher priority queues.

Another method to prevent starvation is reservation. When larger jobs are enqueued, they are allowed to reserve nodes. When these nodes become available, they remain idle or are used for short running jobs only. When all reserved nodes are ready, the larger job runs.

Both backfill and reservation work better, if the job scheduler knows how long an application will use its nodes. That knowledge makes it possible to run some shorter and smaller jobs, while the system is waiting for the remaining nodes to become available to run the larger job. This makes scheduling decisions much more complex, but utilizes the nodes of a system better. Calculating an optimal solution for large numbers of nodes and jobs is extremely time consuming. Therefore, heuristics are used to compute good approximations in an acceptable amount of time.

Scheduling systems that depend on knowing how long an application needs its nodes terminate applications that do not finish within their allocated time. The scheduler sends a signal to the application a few seconds before it removes the job from the system. This

gives applications a chance to save their current state and intermediate results to disk. They can be restarted later at which time they read their saved state and resume from where they were interrupted.

Predicting how long an application will run is difficult. Most job schedulers rely on the user to supply that information. Even for experienced users, this is not always an easy estimate since the running time of most applications is highly input dependent. Users therefore have a tendency to overestimate the time their applications need on the system to ensure that they finish before the scheduler terminates them. It turns out that overestimating is not necessarily bad. Since nodes become available sooner than the scheduler anticipated, it can often run some short jobs until the next scheduled job needs these nodes. Researchers investigating this phenomena have reported that accurate estimates help users because their jobs are serviced more predictably and faster by the job scheduler, but overestimating does not hurt the system, since it can backfill short-duration availabilities.

For some applications and workloads, it is possible to automatically calculate the expected running time. For other applications, it is possible to look at historical information and estimate the running time. Several sophisticated methods have been proposed, but most commercial systems rely on user estimates.

## Node Allocation

Scheduling, discussed in the previous section, is used to decide which job to run when. Once a job has been selected to run, node allocation decides on which nodes this job should run. In a multicomputer, the allocation granularity is usually a node, not individual processor. If a user requests 64 processors and there are four processors per node, then the allocator would assign 16 nodes to that job.

The network topology connecting the nodes arranges them in a geometric pattern such as 2-D or 3-D meshes, hypercubes, or trees. When allocating nodes, it is important to place processes that communicate frequently with each other on nearby nodes. If frequently communicating processes are placed far apart from each other in the network topology, then their communication costs increase. Furthermore, since the links in the network paths between the nodes are shared with

other communication flows, further slowdowns occur due to contention in the network.

While placing the processes of a parallel job in a contiguous region of the network topology is desirable, it is not always possible. The coming and going of differently sized jobs causes fragmentation that leads to the situation where enough nodes are available to run a particular job, but they are scattered throughout the system. Using the nodes anyway leads to poor communication performance, while waiting for more nodes to become available leads to poor system utilization and throughput.

Another complicating factor is that the allocator does not know which processes of an application will communicate with each other once the job starts executing. Each process of a message-passing application receives a rank number at the start of execution. The allocator assumes that processes with similar rank numbers will be the ones communicating with each other most frequently. This assumption is based on the observation that simulations of physical systems often map processes in a 2-D or 3-D layout. In many simulations, physical properties of one region are computed based on the conditions in the surrounding regions. That means information has to flow among processors assigned to neighboring regions.

Unfortunately, when application programmers write their code, they cannot know how their processes will be assigned to nodes, especially if their programs will run on different network topologies and machine sizes. Therefore, programmers often make the assumption that nearby ranks are also near each other in the network. If a program logically arranges its processes in a  $4 \times 4$  grid, the allocator only knows that this application requires 16 processors to run. It is tempting to assume that allocating nodes in a square is best, but that strategy fails for nonsquare numbers of nodes, cannot always be done given the available nodes, and may be a mismatch with the logical ordering the application uses.

In many systems, the scheduler and allocator are separate components. The scheduler receives information about the number of available nodes and uses that to dispatch jobs. The allocator must find nodes for that job, even if that leads to higher fragmentation and decreased application performance. It is beneficial to combine these two functions and allowing the allocator to delay a job until a more favorable configuration

of nodes becomes available. That requires a more complex scheduler, and ways to assure fairness and prevent starvation.

In large-scale, high-performance systems, migration of running processes is considered to be too complex and has too much overhead to be used to achieve better node allocations. Complicating allocation further is that in some systems the nodes are not homogeneous. Matching job requirements with nodes that have different memory sizes and CPUs further complicates node allocation. Often, same node types are grouped and served by separate queues in the scheduler.

## Space and Time-Sharing

Node allocation discussed so far implicitly assumed *space-sharing*, where each processor is allocated to a single job. There might be multiple jobs currently running on a machine, but each job has its own set of processors allocated to it. On nonparallel systems, *time-sharing*, where multiple processes run for short slices of time on a single processor, is more common.

Space-sharing improves the performance of parallel applications. When the individual processes of a parallel application communicate with each other, they often depend on a quick response time for data requests, so they can continue with their computations. If process A is waiting for information from process B, but B is not currently running, then A will be blocked and it may lose its time slice before the response from B arrives. When the data finally does arrive, A may not be running anymore and will not be able to process the data and respond right away. This in turn may delay B, while it is waiting for a response from A.

This problems gets worse the more processes depend on each other's responses. The solution in high-performance systems is space-sharing. Once allocated to their nodes, the processes of a parallel application will not be preempted by other parallel jobs. This improves responsiveness among the processes of a parallel job and lets it finish more quickly.

A method called *gang-scheduling* can help with synchronizing processes of the same job. The idea is to coordinate processors and make sure that context-switches from one job to another occur at exactly the same time on all processors. That way processors can be time-shared among several jobs, but all the processes of one job run during the same time slices. This prevents

blocking when processes of the same job communicate with each other.

Gang-scheduling is not very common because it requires hardware support or very coarse time slices. IBM's Blue Gene machines have a built-in synchronization network that makes gang-scheduling feasible. Gang-scheduling can reduce OS noise: All OS housekeeping tasks are done at the same time on all processors and no application process gets delayed waiting for another process whose OS is busy.

## Future Directions

Desktop computers and even laptops are now parallel systems. With the number of cores increasing for the foreseeable future, the parallelism available in these systems will soon rival many clusters in use today. While scheduling in a shared-memory system is easier, for example, process migration is less difficult, the amount of parallelism and the need for data locality will pose challenges for which traditional OSs have not been designed, but which have been explored in the parallel job scheduling community for many years.

One of the many new challenges in this area to be considered is the following. In the future, the cores inside a multicore CPU will be connected by a Network on Chip (NoC). Each core in use generates heat and raises the temperature in adjacent cores. It is therefore beneficial to distribute the cores that are currently active in a way such that the overall temperature of the chip is minimized. This adds another dimension to the already complex problem of allocation.

## Related Entries

- ▶ [Affinity Scheduling](#)
- ▶ [Checkpointing](#)
- ▶ [Scheduling Algorithms](#)

## Bibliographic Notes and Further Reading

The International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) has been held annually for more than 15 years and is a rich source of information on the topic of parallel scheduling and processor allocation [2]. Over the years the field has evolved. In the introduction to two of the JSSPP workshops, Feitelson et al. document some of the

changes that have occurred [3, 4], while [5] looks at some of the challenges ahead.

Knowing the expected running time of an application is important for job scheduling. Whether user estimates are accurate has been explored in [8], and using historical information to automate runtime prediction has been looked at in [13].

Some common batch scheduling systems in use are: Platform Computing's Load Sharing Facility (LSF) [16], the Portable Batch System (PBS) [6], and a Simple Linux Utility for Resource Management (SLURM) [7, 14].

Topology aware processor allocation and the effect of contiguous or noncontiguous allocation have been most recently studied in [12], while communication awareness is discussed in [1]. Innovative use of (one-dimensional) space-filling curves to allocate processors to improve communication efficiency is explored in [9] and [1]. Making the allocator temperature-aware has been suggested in [10]. The tradeoffs of space- and time-sharing as well as combining several of these methods to improve utilization are discussed in [15].

An excellent description of the architectural and software components of a large-scale parallel system, and how they interact, is contained in [11]. That paper provides a sense of what is required to integrate node scheduling and allocation into a complex parallel system.

## Bibliography

1. Bender MA, Bunde DP, Demaine ED, Fekete SP, Leung VJ, Meijer H, Phillips CA (2008) Communication-aware processor allocation for supercomputers: finding point sets of small average distance. *Algorithmica* 50(2):279–298
2. Feitelson D (Nov 2009) Workshops on job scheduling strategies for parallel processing. <http://www.cs.huji.ac.il/~feit/parsched/>
3. Feitelson DG, Rudolph L, Schwiegelshohn U, Sevcik KC, Wong P (1997) Theory and practice in parallel job scheduling. In: IPPS '97: Proceedings of the job scheduling strategies for parallel processing, Geneva. Springer, London, pp 1–34
4. Feitelson DG, Rudolph L, Schwiegelshohn U (2004) Parallel job scheduling – a status report. In: Feitelson DG, Rudolph L, Schwiegelshohn U (eds) Job scheduling strategies for parallel processing. Lecture Notes in Computer Science, vol 3277. Springer, Berlin, pp 1–16
5. Frachtenberg E, Schwiegelshohn U (2007) New challenges of parallel job scheduling. In: Frachtenberg E, Schwiegelshohn U (eds) Job scheduling strategies for parallel processing. Lecture Notes in Computer Science, vol 4942. Springer, Berlin, pp 1–23

6. Henderson RL (1995) Job scheduling under the portable batch system. In: IPPS '95: Proceedings of the workshop on job scheduling strategies for parallel processing, Santa Barbara. Springer, London, pp 279–294
7. Lawrence Livermore National Laboratory (Nov 2009) SLURM: a highly scalable resource manager. <https://computing.llnl.gov/linux/slurm/>
8. Lee CB, Schwartzman Y, Hardy J, Snavely A (2004) Are user run-time estimates inherently inaccurate? In: Feitelson DG, Rudolph L, Schwiegelshohn U (eds) Job scheduling strategies for parallel processing. Lecture Notes in Computer Science, vol 3277. Springer, Berlin, pp 253–263
9. Leung VJ, Arkin EM, Bender MA, Bunde D, Johnston J, Lal A, Mitchell JSB, Phillips C, Seiden SS (2002) Processor allocation on C plant: achieving general processor locality using one-dimensional allocation strategies. In: CLUSTER '02: proceedings of the IEEE international conference on cluster computing, Chicago. IEEE Computer Society, Washington, DC, pp 296
10. Liao X, Jigang W, Srikanthan T (2008) A temperature-aware virtual submesh allocation scheme for NoC-based manycore chips. In: SPAA '08: proceedings of the twentieth annual symposium on parallelism in algorithms and architectures, Munich. ACM, New York, pp 182–184
11. Moreira JE, Salapura V, Almasi G, Archer C, Bellofatto R, Bergner P, Bickford R, Blumrich M, Brunheroto JR, Bright AA, Brutman M, Castaños JG, Chen D, Coteus P, Crumley P, Ellis S, Engelsiepen T, Gara A, Giampa M, Gooding T, Hall S, Haring RA, Haskin R, Heidelberger P, Hoenicke D, Inglett T, Kopcsay GV, Lieber D, Limpert D, McCarthy P, Megerian M, Mundy M, Ohmacht M, Parker J, Rand RA, Reed D, Sahoo R, Sanomiya A, Shok R, Smith B, Stewart GG, Takken T, Vranas P, Wallenfelt B, Michael B, Ratterman J (2007) The Blue Gene/L supercomputer: a hardware and software story. Int J Parallel Program 35:181–206
12. Pascual JA, Navaridas J, Miguel-Alonso J (2009) Effects of topology aware allocation policies on scheduling performance. In: Frachtenberg E, Schwiegelshohn U (eds) Job scheduling strategies for parallel processing. Lecture Notes in Computer Science, vol 5798. Springer, Berlin, pp 138–156
13. Smith W, Foster I, Taylor V (1998) Predicting application run times using historical information. In: Feitelson DG, Rudolph L (eds) Job scheduling strategies for parallel processing. Lecture Notes in Computer Science, vol 1459. Springer, Berlin, pp 122–142
14. Yoo AB, Jette MA, Grondona M (2003) SLURM: simple Linux utility for resource management. In: Feitelson DG, Rudolph L, Schwiegelshohn U (eds) Job scheduling strategies for parallel processing. Lecture Notes in Computer Science, vol 2862. Springer, Berlin, pp 44–60
15. Zhang Y, Franke H, Moreira J, Sivasubramaniam A (2003) An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration. IEEE T Parall Distr 14(3):236–247
16. Zhou S, Zheng X, Wang J, Delisle P (1993) Utopia: a load sharing facility for large, heterogeneous distributed computer systems. Softw Pract Exp 23(12):1305–1336

# K

## k-ary n-cube

- [Hypercubes and Meshes](#)
- [Networks, Direct](#)

## k-ary n-fly

- [Networks, Multistage](#)

## k-ary n-tree

- [Networks, Multistage](#)

## Knowledge Discovery

- [Data Mining](#)

## KSR

- [Cache-Only Memory Architecture](#)

# L

## LANai

► Myrinet

## Languages

► Programming Languages

## LAPACK

JACK DONGARRA, PIOTR LUSZCZEK  
University of Tennessee, Knoxville, TN, USA

### Definition

LAPACK is a library of Fortran 77 subroutines for solving most commonly occurring problems in dense matrix computations. It has been designed to be efficient on a wide range of modern high-performance computers. The name LAPACK is an acronym for Linear Algebra PACKage. LAPACK can solve systems of linear equations, linear least squares problems, eigenvalue problems, and singular value problems. LAPACK can also handle many associated computations such as matrix factorizations or estimating condition numbers.

### Discussion

LAPACK contains *driver routines* for solving standard types of problems, *computational routines* to perform a distinct computational task, and *auxiliary routines* to perform a certain subtask or common low-level computation. Each driver routine typically calls a sequence of computational routines. Taken as a whole, the computational routines can perform a wider range of tasks than are covered by the driver routines. Many of the auxiliary routines may be of use to numerical analysts or software

developers, so the Fortran source for these routines has been documented with the same level of detail used for the LAPACK routines and driver routines.

LAPACK is designed to give high efficiency performance on vector processors, high-performance “super-scalar” workstations, and shared memory multiprocessors. It can also be used satisfactorily on all types of scalar machines (PCs, workstations, and mainframes). A distributed memory version of LAPACK, ScaLAPACK [1], has been developed for other types of parallel architectures (e.g., massively parallel SIMD machines, or distributed memory machines).

LAPACK has been designed to supersede LINPACK [2] and EISPACK [3, 4] principally by restructuring the software to achieve much greater efficiency, where possible, on modern high-performance computers, and also by adding extra functionality, by using some new or improved algorithms, and by integrating the two sets of algorithms into a unified package.

LAPACK routines are written so that as much as possible of the computation is performed by calls to the Basic Linear Algebra Subprograms (BLAS) [5–7]. Highly efficient machine-specific implementations of the BLAS are available for many modern high-performance computers. The BLAS enable LAPACK routines to achieve high performance with portable code.

The complete LAPACK package or individual routines from LAPACK are freely available on Netlib [8] and can be obtained via the World Wide Web or anonymous ftp. The LAPACK homepage can be accessed via the following URL address: <http://www.netlib.org/lapack/>.

LAPACK90 is a Fortran 90 interface to the Fortran 77 LAPACK library. LAPACK++ is an object-oriented C++ extension to the LAPACK library. The CLAPACK library was built using a Fortran to C conversion utility called f2c [9]. The J LAPACK project provides the LAPACK and BLAS numerical subroutines translated

from their Fortran 77 source into class files, executable by the Java Virtual Machine (JVM) and suitable for use by Java programmers. The ScaLAPACK (or Scalable LAPACK) [1] library includes a subset of LAPACK routines redesigned for distributed memory message-passing MIMD computers and networks of workstations supporting PVM and/or MPI.

Two types of driver routines are provided for solving systems of linear equations: simple and expert. Different driver routines are provided to take advantage of special properties or storage schemes of the system matrix. Supported matrix types and storage schemes include: general, general band, general tridiagonal, symmetric/Hermitian, positive definite, symmetric/Hermitian, positive definite (packed storage), positive definite band, positive definite tridiagonal, indefinite, complex symmetric, and indefinite (packed storage).

## Related Entries

- ▶ [LINPACK Benchmark](#)
- ▶ [ScaLAPACK](#)

## Bibliography

1. Blackford LS, Choi J, Cleary A, D'Azevedo E, Demmel J, Dhillon I, Dongarra J, Hammarling S, Henry G, Petitet A, Stanley K, Walker D, Whaley RC (1997) ScaLAPACK users' guide. SIAM
2. Dongarra JJ, Bunch JR, Moler CB, Stewart GW (1979) LINPACK users' guide Society for Industrial and Applied Mathematics, Philadelphia
3. Smith BT, Boyle JM, Dongarra JJ, Garbow BS, Ikebe Y, Klema VC, Moler CB (1976) Matrix eigensystem routines – EISPACK guide, vol 6, Lecture notes in computer science, Springer, Berlin
4. Garbow BS, Boyle JM, Dongarra JJ, Moler CB (1977) Matrix eigensystem routines – EISPACK guide extension, vol 51, Lecture notes in computer science, Springer, Berlin
5. Lawson CL, Hanson RJ, Kincaid D, Krogh FT (1979) Basic Linear Algebra Subprograms for Fortran usage. ACM Trans Math Soft 5:308–323
6. Dongarra JJ, Croz Du J, Hammarling S, Hanson RJ (1988) An extended set of FORTRAN Basic Linear Algebra Subroutines. ACM Trans Math Soft 14(1):1–17
7. Dongarra JJ, Croz Du J, Duff IS, Hammarling S (1990) A set of Level 3 Basic Linear Algebra Subprograms. ACM Trans Math Soft 16(1):1–17
8. Dongarra JJ, Grosse E (1987) Distribution of mathematical software via electronic mail. Commun ACM 30(5):403–407
9. Feldman SI, Gay DM, Maimone MW, Schryer NL (1990) A Fortran-to-C Converter. AT & T Bell Laboratories, Murray Hill. Computing Science Technical Report, 149

## Large-Scale Analytics

- ▶ [Massive-Scale Analytics](#)

## Latency Hiding

Latency hiding improves machine utilization by enabling the execution of useful operations by a device while it is waiting for a communication operation or memory access to complete. Prefetching, context switching, and instruction-level parallelism are mechanisms for latency hiding.

## Related Entries

- ▶ [Denelcor HEP](#)
- ▶ [Multi-Threaded Processors](#)
- ▶ [Superscalar Processors](#)

## Law of Diminishing Returns

- ▶ [Amdahl's Law](#)

## Laws

- ▶ [Amdahl's Law](#)
- ▶ [Gustafson's Law](#)
- ▶ [Little's Law](#)
- ▶ [Moore's Law](#)

## Layout, Array

PAUL FEAUTRIER  
Ecole Normale Supérieure de Lyon, Lyon, France

## Definitions

A high-performance architecture needs a fast processor, but a fast processor is useless if a memory subsystem does not provide data at the rate of several words per clock cycle. Run-of-the-mill memory chips in today technology have a latency of the order of ten to a hundred processor cycles, far more than the necessary performance. The usual method for increasing the memory

bandwidth as seen by the processor is to implement a cache, i.e., a small but fast memory which is geared to hold frequently used data. Caches work best when used by programs with almost random but nonuniform addressing patterns. However, high-performance applications, like linear algebra or signal processing, have a tendency to use very regular addressing patterns, which degrade cache performance. In linear algebra codes, and also in stream processing, one finds long sequences of accesses to regularly increasing addresses. In image processing, a template moves regularly across a pixel array.

To take advantage of these regularities, in fine-grain parallel architectures, like SIMD or vector processors, the memory is divided into several independent banks. Addresses are evenly scattered among the banks. If  $B$  is the number of banks, then word  $x$  is located in bank  $x \bmod B$  at displacement  $x \div B$ . The low-order bits of  $x$  (the byte displacement) do not participate in the computation. In this way, words in consecutive addresses are located in distinct banks, and can, with the proper interface hardware, be accessed in parallel. A problem arises when the requested words are not at consecutive addresses. *Array layouts* were invented to allow efficient access to various *templates*.

## Discussion

### Parallel Memory Access

There are many ways of taking advantage of such a memory architecture. One possibility is for the processors to access in one memory cycle the  $B$  words which are necessary for a piece of computation, and then to cooperate in computing the result. Consider, for instance, the problem of summing the elements of a vector. One reads  $B$  words, adds them to an accumulator, and proceeds to the next  $B$  words.

Another organization is possible if the program operations can be executed in parallel, and if the data for each operation are regularly located in memory. Consider, for instance, the problem of computing the weighted sum of three rows in a TV image (downsampling). One possibility is to read three words in the same column and do the computation. But since it is likely that  $B$  is much larger than three, the memory subsystem will be under-utilized. The other possibility is to read  $B$  words in the first row,  $B$  words in the second row, and so on, do the summation, and store  $B$  words of the

result. This method is more efficient, but places more constraints on the target application.

### Templates and Template Size

In both situations, the memory is addressed through *templates*, i.e., finite sets of cells which can move across an array. Consider for instance a vector:

```
float A[100].
```

The template  $(0, 1, 2, 3)$  represents four consecutive words. If located (or *anchored*) at position  $i$ , it covers elements  $A[i]$ ,  $A[i+1]$ ,  $A[i+2]$  and  $A[i+3]$  of vector  $A$ . But the most interesting case is that of a two-dimensional array, which may be a matrix or an image. The template  $((0,0), (0,1), (1,0), (1,1))$  represents a two-by-two square block. If anchored at position  $(i,j)$  in a matrix  $M$ , it covers elements  $M[i][j]$ ,  $M[i][j+1]$ ,  $M[i+1][j]$ , and  $M[i+1][j+1]$ . The first template is one dimensional, and the second one is two dimensional. One may consider templates of arbitrary dimensions, but these two are the most important cases.

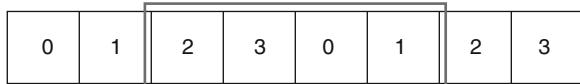
The basic problem is to distribute arrays among the available memory banks in such a way that access to all elements of the selected template(s) takes only one memory cycle. It follows that the size of a template is at most equal to the number of banks,  $B$ . As has been seen before, memory usage is optimized when templates have exactly  $B$  cells; this is the only choice that is discussed here.

Observe also that, as the previous example has shown, there may be many template selections for a given program. The problem of selecting the best one will not be discussed here.

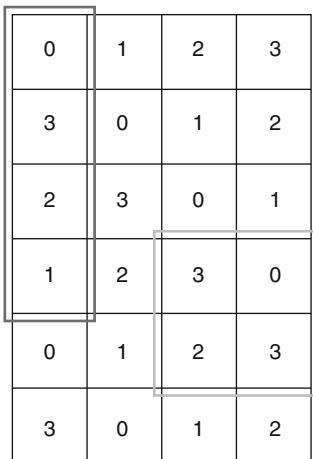
### Layouts

A layout is an assignment of bank numbers to array cells. A template can be viewed as a stencil that moves across an array and shows bank numbers through its openings. A layout is *valid* for a template position if all shown bank numbers are distinct. In Fig. 1, the layout is valid for all positions of the solid template. In Fig. 2, the layout is valid for all positions of the solid template, and valid for no position of the dashed template.

The reader may have noticed that the information given by array layouts such as Fig. 1 or 2 is not complete. One must also know at which address each cell is located in its bank. This information can always be retrieved



Layout, Array. Fig. 1 A vector layout



Layout, Array. Fig. 2 A matrix layout

from the layout. Simply order the array cells and allocate each cell to the first free word in its bank. For Fig. 2, the bank address is found equal to the row number. This scheme imposes two constraints on a layout:

- Array cells must be evenly distributed among banks.
- The correspondance between cells and bank addresses cannot be too complex, since each processor must know and use it.

Depending on the application, one may want that a layout be valid at some positions in the array, or at every possible position. For instance, in linear algebra, it is enough to move the template in such a way that each entry is covered once and only once. One says that the template tessellates or *tiles* the array domain. Once a tiling has been found one simply numbers each square of the template from 0 to  $B - 1$ , and reproduces this assignment (perhaps after a permutation) at all positions of the tile. This insures that the array cells are evenly distributed among the banks. In other applications, like image processing, the template must be moved at every position in the array. Imagine for instance that a smoothing operator must be applied at all pixels of an image. In fact, the two situations are

equivalent: If a template tiles an array and if block numbers are assigned as explained above, then the layout is valid for every position. For a proof, see [4].

## Skewing Schemes: Vector Processing

Consider the following example:

```
for(i=0; i< n; i=i+1)
 X[i] = Y[i] + Z[i],
```

to be run on a SIMD architecture with  $B$  processing elements. If the layout of  $Y$  is such that  $Y[i]$  is stored in bank  $i \bmod B$  at address  $i \div B$ ,  $B$  elements of  $Y$  can be fetched in one memory cycle. After another fetch of  $B$  words of  $Z$ , the SIMD processor can execute  $B$  addition in parallel. Another memory cycle is needed to store the  $B$  results. Note that this scheme has the added advantage that all groups of  $B$  words are at the same address in each bank, thus fitting nicely in the SIMD paradigm.

Suppose now that the above loop is altered to read:

```
for(i=0; i< n; i=i+1)
 X[d*i] = Y[d*i] + Z[d*i],
```

where  $d$  is a constant (the step). The block number  $d \cdot i \bmod B$  is equal to  $d \cdot i - k \cdot B$  for some  $k$ , and hence is a multiple of the greatest common divisor (gcd)  $g$  of  $d$  and  $B$ . It follows that only  $B/g$  words can be fetched in one cycle, and the performance is divided by  $g$ . Since  $B$  is usually a power of 2 in order to simplify the computation of  $x \bmod B$  and  $x \div B$ , a loss of performance will be incurred whenever  $d$  is even. One can always invent layouts that support non-unit step accesses. For instance, if  $d = 2$ , assign cell  $i$  to bank  $(i \div 2) \bmod B$ . However, in many algorithms that use non-unit steps, like the Fast Fourier Transform,  $d$  is a variable. It is not possible to accomodate all its possible values without costly remapping.

## Skewing Schemes: Matrix Processing

In C, the canonical method for storing a matrix is to concatenate its rows in the order of their subscripts. If the matrix is of size  $M \times N$ , the displacement of element  $(i, j)$  is  $N \cdot i + j$ . In Fortran, the convention is reversed, but the conclusions are the same, *mutatis mutandis*. One may assume that the row length  $N$  is a multiple of  $B$ , by padding if necessary; hence, all matrices may be assumed to have size  $M \times B$ . As a consequence, element  $(i, j)$  is allocated to bank  $j \bmod B$ .

|            |     |     |     |            |            |            |            |
|------------|-----|-----|-----|------------|------------|------------|------------|
| <b>0,0</b> | 0,1 | 0,2 | 0,3 | <b>0,0</b> | 0,1        | 0,2        | 0,3        |
| <b>1,0</b> | 1,1 | 1,2 | 1,3 | 1,3        | <b>1,0</b> | 1,1        | 1,2        |
| <b>2,0</b> | 2,1 | 2,2 | 2,3 | 2,2        | 2,3        | <b>2,0</b> | 2,1        |
| <b>3,0</b> | 3,1 | 3,2 | 3,3 | 3,1        | 3,2        | 3,3        | <b>3,0</b> |

$B = 4, S = 0$ : four cycles are needed to access column 0  
a

$B = 4, S = 1$ : column 0 can be accessed in one cycle  
b

Layout, Array. Fig. 3 The effect of skewing

If the algorithm accesses a matrix by rows, as in:

```
for(j=0; j<N; j++)
... X[i][j] ...
```

performance will be maximal. However, when accessing columns, consecutive items will be separated by  $B$  rows, and be located all in the same bank. All parallelism will be lost.

A better solution is *skewing* (Fig. 3). The cell  $(i, j)$  is now allocated to bank  $(S \cdot i + j) \bmod B$ .  $S$  is the skewing factor. Here, two consecutive cells in a row are still at a distance of 1; hence, a row can still be accessed in one cycle. But two consecutive cells in a column,  $(i, j)$  and  $(i + 1, j)$  are at a distance of  $S$ . If  $S$  is selected to be relatively prime to  $B$  (for instance,  $S = 1$  if  $B$  is even Fig. 3 (b)) access to a column will also take one cycle.

This raises the question of whether other patterns can be accessed in parallel. Consider the problem of accessing the main diagonal of a matrix. In the above skewing scheme, the distance from cell  $(i, i)$  to  $(i+1, i+1)$  is  $S+1$ , and  $S$  and  $S+1$  both cannot be odd. Hence, accessing both columns and diagonals (or both rows and diagonals) in parallel with the same skewing scheme is impossible when  $B$  is even.

### Skewing Schemes: More Templates

It is clear that a more general theory is needed when the subject algorithm needs to use several templates at the same time. As seen above, a template has a valid layout if and only if it tiles the two-dimensional plane. This result seems natural if the template is required to cover once and only once each cell of the underlying array. If bank numbers are assigned in the same order in each tile [4] or are regularly permuted from tile to tile [3] then the layout will obviously be valid for each tile. The striking fact is that the layout will also be valid when the template is offset from a tile position. Observe that the previous

results on matrix layouts corresponds to the many ways of tiling the plane with  $B$ -sized linear templates.

A tiling is defined by a set of vectors  $v_1, \dots, v_n$  (translations). If a template instance has been anchored at position  $(i, j)$ , then there are other instances at positions  $(i, j) + v_1, \dots, (i, j) + v_n$ . The translations  $v_1, \dots, v_n$  must be selected in such a way that the translates do not overlap.

If an algorithm needs several templates, the first condition is that each of them tiles the plane. But each tiling defines a layout, and these layouts must be compatible. One can show that the compatibility condition is that the translation vectors are the same for all templates. Since a template may tile the plane in several different ways, finding the right tiling may prove a difficult combinatorial problem.

### Pipelined Processors and GPU

Consider the loop:

```
for(i=0; i<n; i=i+1)
 s = s + A[i],
```

to be run on a pipelined processor which executes one addition per cycle. If the memory latency is equal to  $B$  processor cycles, the memory banks can be activated in succession, in such a way that, after a prelude of  $B$  cycles, the processor receives an element of  $A$  at each cycle. As for parallel processors, if the loop has a step of  $d$ , the performance will be reduced by a factor of  $\gcd(d, B)$ .

While today vector processors are confined to niche applications, the same problem is reemerging for Graphical Processing Units (GPU), which can access  $B$  words in parallel under the name “Global Memory Access Coalescing.”

### Reordering

With nonstandard layouts, the results of a parallel access may not be returned in the natural order. Consider, for instance, the problem of having parallel access both for rows and for the main diagonal of a  $4 \times 4$  matrix on a  $4$  banks memory. The skewing factor  $S$  must be such that  $S + 1$  is relatively prime to 4:  $S = 2$  is a valid choice. The corresponding layout is shown on Fig. 4. The integer in row  $i$  and column  $j$  is the number of the bank which holds element  $(i, j)$  of the matrix. One can see that bank 0 holds element  $(0, 0)$ , bank 1 holds element  $(3, 3)$ , and so on. If the banks are connected to processors in

|   |          |          |          |          |
|---|----------|----------|----------|----------|
|   | 0        | 1        | 2        | 3        |
| 0 | <b>0</b> | 1        | 2        | 3        |
| 1 | 2        | <b>3</b> | 0        | 1        |
| 2 | 0        | 1        | <b>2</b> | 3        |
| 3 | 2        | 3        | 0        | <b>1</b> |

Layout, Array. Fig. 4 The need for reordering

the natural order, the elements of the main diagonal are returned in the order  $(0,0), (3,3), (2,2), (1,1)$ . This might be unimportant in some cases, for instance, if the elements of the main diagonal are to be summed (the *trace* computation), since addition is associative and commutative.

In other cases, returned values must be reordered. This can be done by inserting a  $B$  words fast memory between the main memory subsystem and the processors. One can also use a *permutation network* which may route a datum from any bank to any processor.

## The Number of Banks

Remember that in vector processing, when the step  $d$  is not equal to one, performance is degraded by a factor  $\gcd(d, B)$ . It is tempting to use a prime number for  $B$ , since then the gcd will always be one except when  $d$  is a multiple of  $B$ . However, having  $B$  a power of two considerably simplifies the addressing hardware, since computing  $x \bmod B$  and  $x \div B$  is done just by routing wires. This has lead to the search for prime numbers with easy division, which are of one of the forms  $2^p \pm 1$ . See [2] for a discussion of this point.

## Bibliography

1. Budnik P, Kuck DJ (1971) The organisation and use of parallel memories. IEEE Trans Comput C-20:1566–1569
2. de Dinechin BD (1991) A ultra fast euclidean division algorithm for prime memory systems. In: Supercomputing'91, Albuquerque, pp 56–65
3. Jalby W, Frailong JM, Lenfant J (1984) Diamond schemes: an organization of parallel memories for efficient array processing. Technical Report RR-342, INRIA
4. Shapiro HD (1978) Theoretical limitations on the efficient use of parallel memories. IEEE Trans Comput C-27:421–428

## LBNL Climate Computer

► [Green Flash: Climate Machine \(LBNL\)](#)

## libflame

FIELD G. VAN ZEE<sup>1</sup>, ERNIE CHAN<sup>2</sup>

ROBERT A. VAN DE GEIJN<sup>1</sup>

<sup>1</sup>The University of Texas at Austin, Austin, TX, USA

<sup>2</sup>NVIDIA Corporation, Santa Clara, CA, USA

## Synonyms

FLAME

## Definition

libflame is a software library for dense matrix computations.

## Discussion

### Introduction

The libflame library is a modern, high-performance dense linear algebra library that is extensible, easy to use, and available under an open source license. It is both a framework for developing dense linear algebra solutions and a ready-made library that strives to be user-friendly while offering competitive (and in many cases superior) real-world performance when compared to the more traditional Basic Linear Algebra Subprograms (BLAS) and Linear Algebra PACKAGE (LAPACK) libraries. It was developed as part of the FLAME project.

### The FLAME Project

A solution based on fundamental computer science. The FLAME project advocates a more stylized notation for expressing loop-based linear algebra algorithms. This notation, illustrated in Fig. 1, closely resembles how matrix algorithms are illustrated with pictures. The notation facilitates formal derivation of algorithms so that the algorithms are developed hand-in-hand with their proof of correctness. For a given operation this yields a family of algorithms so that the best option for a given situation (e.g., problem size or architecture) can be chosen.

---

**Algorithm:**  $A \leftarrow \text{CHOL\_BLK}(A)$

---

$$\text{Partition } A \rightarrow \left( \begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right)$$

where  $A_{TL}$  is  $0 \times 0$

while  $m(A_{TL}) < m(A)$  do

Determine block size  $b$

Repartition

$$\left( \begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right) \rightarrow \left( \begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array} \right)$$

where  $A_{11}$  is  $b \times b$

---

$$\begin{aligned} A_{11} &:= A_{11} - A_{10}A_{11}^T && (\text{SYRK}) \\ A_{21} &:= A_{21} - A_{20}A_{10}^T && (\text{GEMM}) \\ A_{11} &:= \text{CHOL}(A_{11}) && (\text{CHOL}) \\ A_{21} &:= A_{21}A_{11}^{-T} && (\text{TRSM}) \end{aligned}$$


---

Continue with

$$\left( \begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right) \leftarrow \left( \begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array} \right)$$

endwhile

**libflame. Fig. 1** Blocked Cholesky factorization expressed with FLAME notation. Subproblems annotated as SYRK, GEMM, and TRSM correspond to level-3 BLAS operations while CHOL performs a Cholesky factorization of the submatrix  $A_{11}$

**Object-based abstractions and API.** The libflame library is built around opaque structures that hide implementation details of matrices, such as data layout, and exports object-based programming interfaces to operate upon these structures. FLAME algorithms are expressed (and coded) in terms of smaller operations on sub-partitions of the matrix operands. This abstraction facilitates programming without array or loop indices, which allows the user to avoid index-related programming errors altogether, as illustrated in Fig. 2.

**Educational value.** The clean abstractions afforded by the notation and API, coupled with the systematic methodology for deriving algorithms, make FLAME well suited for instruction of high-performance linear algebra courses at the undergraduate and graduate level.

## The libflame Library

The techniques developed as part of the FLAME project have been instantiated in the libflame library.

**A complete dense linear algebra framework.** The libflame library provides ready-made implementations of common linear algebra operations. However, libflame differs from traditional libraries in two important ways. First, it provides families of algorithms for each operation so that the best can be chosen for a given circumstance. Second, it provides a framework for building complete custom linear algebra codes. This makes it a flexible and useful environment for prototyping new linear algebra solutions.

**High performance.** The libflame library dispels the myth that user- and programmer-friendly linear algebra codes cannot yield high performance. Implementations of operations such as Cholesky factorization and triangular matrix inversion often outperform the corresponding implementations currently available in traditional libraries. For sequential codes, this is often simply the result of implementing the operations with algorithmic variants that cast the bulk of their computation in terms of higher-performing matrix operations such as general matrix rank-k updates.

**Support for nontraditional storage formats.** The libflame library supports both column-major and row-major order (“flat”) storage of matrices. Indeed, it allows for general row and column strides to be specified. In addition, the library supports storing matrices hierarchically, by blocks, which can yield performance gains through improved spatial locality. Previous efforts [2] at implementing storage-by-blocks focused on solutions that required intricate indexing. libflame simplifies the implementation by allowing the elements within matrix objects to be references to smaller submatrices. Furthermore, the library provides user-level APIs which allow application programmers to read and modify individual elements and submatrices while simultaneously abstracting away the details of the storage scheme.

**Supported operations.** As of this writing, the following operations have been implemented within libflame to operate on both “flat” and hierarchical matrix objects:

- **Factorizations.** Cholesky; LU with partial pivoting; QR and LQ.
- **Inversions.** Triangular; Symmetric/Hermitian Positive Definite.

```

void FLA_Chol_blk(FLA_Obj A, int nb_alg)
{
 FLA_Obj ATL, ATR, A00, A01, A02,
 ABL, ABR, A10, A11, A12,
 A20, A21, A22;
 int b;

 FLA_Part_2x2(A, &ATL, &ATR,
 &ABL, &ABR, 0, 0, FLA_TL);

 while (FLA_Obj_length(ATL) < FLA_Obj_length(A))
 {
 b = min(FLA_Obj_length(ABR), nb_alg);

 FLA_Repart_2x2_to_3x3(
 ATL, /**/ ATR, &A00, /**/ &A01, &A02,
 /* **** */ /* **** */ /* **** */
 &A10, /**/ &A11, &A12,
 ABL, /**/ ABR, &A20, /**/ &A21, &A22,
 b, b, FLA_BR);
 /* ----- */
 FLA_Syrk(FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
 FLA_MINUS_ONE, A10, FLA_ONE, A11);

 FLA_Gemm(FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
 FLA_MINUS_ONE, A20, A10, FLA_ONE, A21);

 FLA_Chol_unb(FLA_LOWER_TRIANGULAR, A11);

 FLA_Trsm(FLA_RIGHT, FLA_LOWER_TRIANGULAR,
 FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
 FLA_ONE, A11, A21);
 /* ----- */
 FLA_Cont_with_3x3_to_2x2(
 &ATL, /**/ &ATR, A00, A01, /**/ A02,
 A10, A11, /**/ A12,
 /* **** */ /* **** */ /* **** */
 &ABL, /**/ &ABR, A20, A21, /**/ A22,
 FLA_TL);
 }
}

```

**libflame.** Fig. 2 The algorithm shown in Fig. 1 implemented with the FLAME/C API

- **Solvers.** Linear systems via any of the factorizations above; Solution of the Triangular Sylvester equation.
- **Level-3 BLAS.** General, Triangular, Symmetric, and Hermitian matrix-matrix multiplies; Symmetric and Hermitian rank-k and rank-2k updates; Triangular solve with multiple right-hand sides.
- **Miscellaneous.** Triangular-transpose matrix multiply; Matrix-matrix AXPY and COPY; and several other supporting operations.

**Cross-platform support.** The libflame distribution includes build systems for both GNU/Linux and

Microsoft Windows. The GNU/Linux build system provides a configuration script via GNU autoconf and employs GNU make for compilation and installation, and thus adheres to the well-established “configure; make; make install” sequence of build commands.

**Compatibility with LAPACK.** A set of compatibility routines is provided that maps conventional LAPACK invocations to their corresponding implementations within libflame. This compatibility layer is optional and may be disabled at configure-time in order to allow the application programmer to continue to

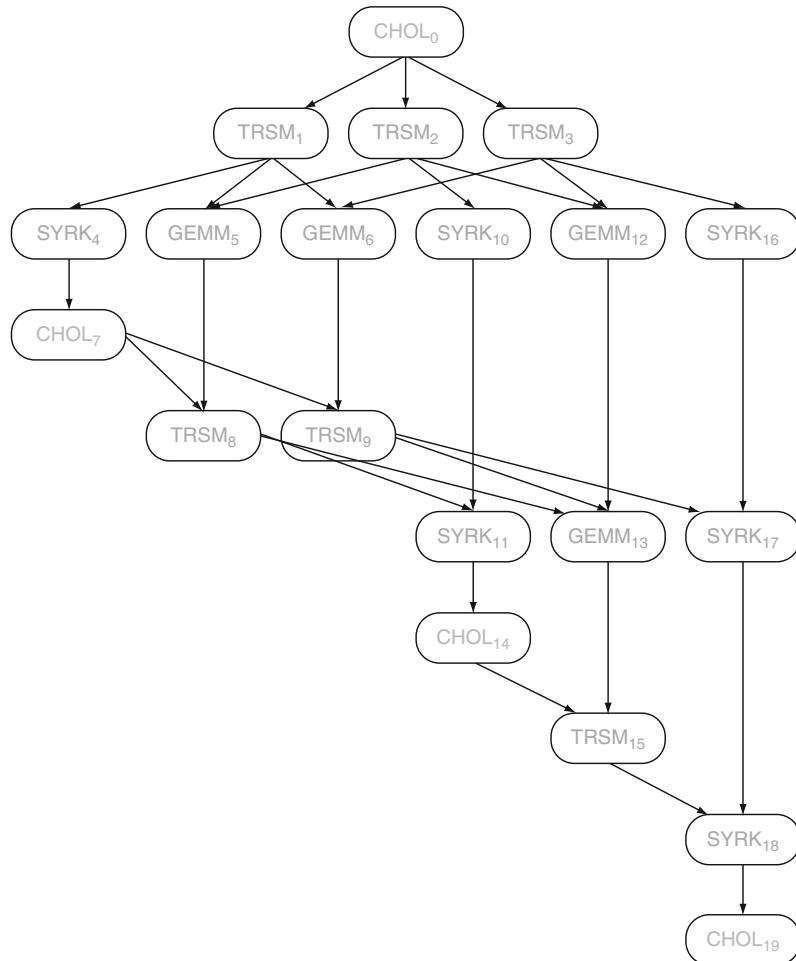
use legacy LAPACK implementations alongside native libflame functions.

## Support for Parallel Computation

**Multithreaded BLAS.** A simple technique for exploiting thread-level parallelism is to link the sequential algorithms for flat matrices in libflame to multithreaded BLAS libraries where each call to a BLAS routine is parallelized individually. Inherent synchronization points occur between successive calls to BLAS routines and thus restrict opportunities for parallelization between subproblems. Routines are available within libflame to adjust the block size of these sequential implementations in order to attain the best performance from the multithreaded BLAS libraries.

SuperMatrix. The fundamental bottleneck to exploiting parallelism directly within many linear algebra codes is the web of data dependencies that frequently exists between subproblems. As part of libflame, the SuperMatrix runtime system has been developed to detect and analyze dependencies found within algorithms-by-blocks (algorithms whose subproblems operate only on block operands). Once data dependencies are known, the system schedules sub-operations to independent threads of execution. This system is completely abstracted from the algorithm that is being parallelized and requires virtually no change to the algorithm code while exposing abundant high-level parallelism.

The directed acyclic graph for the Cholesky factorization, given a  $4 \times 4$  matrix of blocks (a hierarchical



libflame. Fig. 3 The directed acyclic graph for the Cholesky factorization on a  $4 \times 4$  matrix of blocks

matrix stored with one level of blocking), is shown in Fig. 3. The nodes of the graph represent tasks that operate over different submatrix blocks, and the edges represent data dependencies between tasks. The subscripts in the names of each task denote the order in which the tasks would typically be executed in a sequential algorithm-by-blocks. Many opportunities for parallelism exist within the directed acyclic graph such as  $\text{TRSM}_1$ ,  $\text{TRSM}_2$ , and  $\text{TRSM}_3$  being able to be executed in parallel.

**PLAPACK.** FLAME and `libflame` have their roots in the distributed-memory parallel PLAPACK library. Integrating PLAPACK into `libflame` is a current effort.

## Related Entries

- ▶ [BLAS \(Basic Linear Algebra Subprograms\)](#)
- ▶ [LAPACK](#)
- ▶ [PLAPACK](#)

## Bibliographic Notes and Further Reading

The FLAME notation and APIs have their root in the Parallel Linear Algebra Package (PLAPACK). Eventually the insights gained for implementing distributed-memory libraries found their way back to the sequential library `libflame`.

The first papers that outlined the vision that became the FLAME project were published in 2001 [3, 4]. A book has been published that describes in detail the formal derivation of dense matrix algorithms using the FLAME methodology [6]. How scheduling of tasks is supported was first published in [1] and discussed in more detail in [5]. An overview of `libflame` can be found in [7] and an extensive reference guide has been published [8].

## Bibliography

1. Chan E, Quintana-Ortí ES, Quintana-Ortí G, van de Geijn R (June 2007) SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In: SPAA '07: Proceedings of the nineteenth ACM symposium on parallelism in algorithms and architectures, San Diego, CA, USA, pp 116–125
2. Elmroth E, Gustavson F, Jonsson I, Kagstrom B (2004) Recursive blocked algorithms and hybrid data structures for dense matrix library software. SIAM Rev 46(1):3–45
3. Gunnels JA, Gustavson FG, Henry GM, van de Geijn RA (Dec 2001) FLAME: formal linear algebra methods environment. ACM Trans Math Softw 27(4):422–455
4. Gunnels JA, van de Geijn RA (2001) Formal methods for high-performance linear algebra libraries. In: Proceedings of the IFIP TC2/WG2.5 working conference on the architecture of scientific software, Ottawa, Canada, pp 193–210
5. Quintana-Ortí G, Quintana-Ortí ES, van de Geijn RA, Van Zee FG, Chan E (July 2009) Programming matrix algorithms-by-blocks for thread-level parallelism. ACM Trans Math Softw 36(3):14:1–14:26
6. van de Geijn RA, Quintana-Ortí ES (2008) The science of programming matrix computations. <http://www.lulu.com/content/1911788/>
7. Van Zee FG, Chan E, van de Geijn R, Quintana-Ortí ES, Quintana-Ortí G (2009) Introducing: The libflame library for dense matrix computations. IEEE Computing in Science and Engineering 11(6):56–62
8. Van Zee FG (2009) libflame: the complete reference. <http://www.lulu.com/content/5915632/>

## Libraries, Numerical

- ▶ [Numerical Libraries](#)

## Linda

ROBERT BJORNSEN  
Yale University, New Haven, CT, USA

### Definition

Linda [1] is a coordination language for parallel and distributed computing, invented by David Gelernter and first implemented by Nicholas Carriero. Work on Linda was initially done at SUNY Stony Brook and continued at Yale University. Linda uses tuple spaces as the mechanism for communication, coordination, and process control. The goal is to provide a tool that makes it simple to develop parallel programs that are easy to understand, portable and efficient.

### Discussion

#### Introduction

Linda is a coordination language concerned with issues related to parallelism and distributed computing, such as communication of data, coordination of activities, and creation and control of new threads/processes.

Linda is not concerned with other programming language issues such as control structures (testing, looping), file I/O, variable declaration, etc. Thus, it is orthogonal to issues of traditional programming languages, and is normally embedded in a sequential language such as C, Fortran, Java, etc., resulting in a Linda dialect of that language (C-Linda, Fortran-Linda, Java-Linda, etc). It is important to distinguish between Linda as a conceptual model for coordination and a concrete X-Linda implementation. The latter necessarily imposes restrictions and limitations on the former.

## Tuples and Tuple Spaces

Tuples (**too-pulls**) are the fundamental data objects in Linda. A tuple is an ordered, typed collection of any number of elements. These elements can be whatever values or objects are supported by the host language. Elements can also be typed place holders, or *formals*, which will act as wild cards. Here are some example tuples:

```
(“counter”, 3)
(“table”, 4, 6, 5.67)
(“table, i, j, ? a)
```

Tuples exist in tuple spaces. Processes communicate and synchronize with one another by creating and consuming tuples from a common tuple space. Tuple spaces exist, at least abstractly, apart from the processes using them. Tuple space is sometimes referred to as an associative memory. A tuple establishes associations between the values of its constituent elements. These associations are exploited by specifying some subset of the values to reference the tuple as a whole.

## Tuple Space Operations

There are two operations that create tuples: `out` and `eval`. `Out` simply evaluates the expressions that yield its elements in an unspecified order and then places the resulting tuple in tuple space:

```
out(“counter”, 5)
```

A tuple space is a multiset: if the above operation is repeated  $N$  times (and no other operations are done), then the tuple space will contain  $N$  identical (“counter”, 5) tuples.

`Eval` is identical to `out`, except that the tuple is placed into tuple space without completing the evaluation of

its elements. Instead, new execution entities are created to evaluate the elements of the tuple (depending on the implementation, these entities might be threads or processes). Once all elements have been evaluated, the tuple is indistinguishable from one created via `out`. `Eval` is typically used with tuples that contain function calls, and is an idiomatic and portable way to create parallelism in Linda.

For example, a process might create a new worker by doing:

```
eval(“new worker”, worker(i))
```

`Eval` will immediately return control to the calling program, but in the background a new execution entity will begin computing `worker(i)`. When that subroutine returns, its value will be the second element of a “new worker” tuple.

There are two operations that can extract tuples from tuple space: `in` and `rd` (Traditionally read was spelled `rd` to avoid collisions with `read` from the C library. The exact verbs used have varied somewhat, with `store` and `write` common variants of `out`, and `take` a common variant of `in`). These operations take a template and perform a match of its arguments against the tuples currently in tuple space. A template is similar to a tuple, consisting of some number of actual and formal fields.

If a match is found, the tuple is returned to the process performing the `in` or `rd`. In the case of `in`, the tuple is deleted from tuple space after matching, whereas `rd` leaves the tuple in tuple space. If multiple matches are available, one is chosen arbitrarily. If no matches are available, the requesting process blocks until a match is available. Tuples are always created and consumed atomically.

In order to match, the tuple and template must:

1. Agree on the number of fields
2. Agree on the types of corresponding fields
3. Agree in the values of corresponding actual fields
4. Have no corresponding formals (this is unusual)

‘?’ indicates that the argument is a place-holder, or formal argument. In this case, only the types must match, and as a side effect, the formal variable in the template will be assigned the value from the tuple. Formals normally appear only in templates, but in some limited circumstances can appear in tuples as well (it is

not uncommon for a particular implementation to omit support for formals in tuples). Some work was also proposed to allow a “ticking” type modifier to enable specification of elements still under evaluation. Absent this, the matching above implicitly enforces matches on tuples whose elements have all completed evaluation.

For example, the template:

`in("counter", 5)`

would match the above tuple, as would

`in("counter", ?i) also setting i to 5`

`in("counter", i) assuming i was bound to 5`

But these templates would not match:

`in("counter", 5, 5) rule 1`

`in("counter", 5.0) rule 2`

`in("counter", 6) rule 3`

In addition to `in` and `rd`, there are non-blocking variants `inp` and `rdp`, which return `FALSE` rather than blocking. These variants can be useful in certain circumstances, but their use has generally been discouraged since it introduces timing dependencies.

In the above examples, note that the first field is always a string that serves to name the tuple. This is not required, but is a common useful idiom. It also provides a useful hint to the Linda implementation that can improve performance, as discussed below.

## Linda Examples

Linda programmers are encouraged to think of coordination in terms of data structure manipulations rather than message passing, and a number of useful idioms have been developed along these lines. Indeed, Linda can be viewed as a means to generalize Wirth’s “Algorithms + Data Structures = Programs”: “Algorithms + Distributed Data Structures = Parallel Programs” [2]. That is, tuples can be used to create rich, complex *distributed* data structures in tuple space. Distributed data structures can be shared and manipulated by cooperating processes that transform them from an initial configuration to a final result. In this sense, the development of a parallel program using Linda remains conceptually close to the development of a sequential (ordinary) program. Two very simple, yet common, examples are a FIFO queue and a master/worker pattern.

## FIFO Queue

In the FIFO queue (Fig. 1), each queue is represented by a number of tuples: one for each element currently in the queue, plus two counter tuples that indicate the current position of the head and tail of the queue. Pushing a value onto the tail of the queue is accomplished by assigning it the current value of the tail counter, and then incrementing the tail, and similarly for popping. Important to note is that no additional synchronization is needed to allow multiple processes to push and pop from the queue “simultaneously.” The head and tail tuples act as semaphores, negotiating access to each position of the queue automatically.

## Master/Worker Pattern

In the master/worker example (Fig. 2), the master initially creates some number of worker processes via `eval`. It then creates task tuples, one per task, and waits for the results. Once all the results are received, a special “poison” task is created that will cause all the workers to shut down.

The workers simply sit in a loop, waiting to get a tuple that describes a task. When they do, if it is a normal task, they process it into a result tuple that will be consumed by the master. In the case of a poison task, they immediately use its elements to create a new task tuple to replace the one they consumed, poisoning another worker, and then finish.

```

init(qname) {
 out(qname, 'head', 0)
 out(qname, 'tail', 0)
}

push(qname, elem) {
 in(qname, 'tail', ?p)
 out(qname, 'elem', p, elem)
 out(qname, 'tail', p+1)
}

pop(qname) {
 in(qname, 'head', ?p)
 in(qname, 'elem', p, ?elem)
 out(qname, 'head', p+1)
 return elem
}

```

Linda. Fig. 1 A FIFO queue written in Linda

```

master() {
 for (i=0; i<NUMWORKERS; ++i) {
 eval(worker())
 }

 for (i=0; i<NUMTASKS; ++i) {
 out('task', i, task[i])
 }

 for (i=0; i<NUMTASKS; ++i) {
 in('result', ?taskid, ?result)
 /* process result */
 ...

 out('task', -1, POISON)
 }
}

worker() {
 while (1) {
 in('task', ?taskid, ?task)
 if (taskid===-1) {
 /* return poison and quit */
 out('task', taskid, task)
 break
 /* this is the actual work */
 result = process(task, taskid)
 out('result', taskid, result)
 }
}

```

**Linda. Fig. 2** The master/worker pattern written in Linda

This is the simplest sort of master/worker pattern one could imagine, yet it still presents a number of interesting features. Process startup and shutdown is represented. Load balancing occurs automatically, since slow workers or large tasks will simply take longer, and as long as there are significantly more tasks than workers the processing time will tend to balance.

The code as shown can be modified in a number of simple but interesting ways:

1. If there are a very large number of tasks, rather than having the master create all the task tuples at once (and thereby possibly filling memory) it can create an initial pool. Then, each time it receives a result it can create another task, thus maintaining a reasonable number of tasks at all times.
2. If the master needs to process the results in task order, rather than as arbitrarily received, the in statement that retrieves the result can be modified to make taskid an actual, i.e., in("result", i, ?result).
3. The program can be made fault-tolerant by wrapping the worker's inner loop in a transaction (for Linda implementations that support transactions) [3]. If a worker dies during the processing of a task,

the original task tuple will automatically be returned when the transaction fails, and another worker will pick up the task.

4. The program can be easily generalized to have new task tuples generated by the workers (in a sense, the propagation of the poison task is an example of this). This could, for example, be used to implement a dynamic divide-and-conquer decomposition.

## Implementations

Linda has been implemented on virtually every kind of parallel hardware ever created, including shared-memory-multiprocessors, distributed-memory-multi processors, non-uniform memory access (NUMA) multiprocessors, clusters, and custom hardware.

Linda is an extremely minimalist language design, and naive, inefficient implementations are deceptively simple to build. Most take the approach of building a runtime library that simply implements tuple space as a list of tuples and performs tuple matching by exhaustively searching the list. These sorts of implementations led to a suspicion that Linda is not suitable for serious computation. Many critics admired the language's elegance, but insisted that it could not be implemented efficiently enough to compete with other approaches, such as shared memory with locking, or native message passing. In response, considerable work was done at Yale and elsewhere during the 1980s and 1990s to create efficient implementations that could compete with lower-level approaches. The efforts focused on two main directions:

1. Compile-time optimizations. Carriero [4] recognized that tuple matching overhead could be greatly reduced by analyzing tuple usage at compile time and translating it to simpler operations. Tuple operations could be segregated into distinct, non-interacting partitions based on type information available at compile time. Each partition was then classified into one of a number of usage patterns and reduced to an efficient data structure such as a counter, hash table or queue, which could then be searched and updated very efficiently. This work demonstrated that Linda on shared-memory multiprocessors could be as efficient as handwritten codes using low level locks. An important part of the demonstration showed for a variety of

representative problems that on average fewer than two tuples would need to be searched before a match was found.

2. Run-time optimizations. For distributed memory implementations, optimization of tuple storage was key to reducing the overall number of messages required by tuple space operations. Each tuple had a designated node on which it would be stored, usually chosen via hashing. Naively, each out/in pair would require as many as three distinct messages, since out would require a single message sending the tuple to its storage location, and in would require two messages, a request to the storage location and a reply containing the matched tuple. Bjornson [5] built upon Carriero's compile-time optimizations by analyzing tuple traffic at runtime, recognizing common communication patterns such as broadcast (one-to-many), point-to-point, and many-to-one, and rearranging tuple space storage to minimize the number of messages required. This work demonstrated that most communication in Linda required only one or two sends, just as with message passing.

Overall, this work demonstrated that while in the abstract Linda programs could make unrestrained use of expensive operations such as content addressability and extensive search, in practice any given program was likely to confine itself to well-understood patterns that could be reduced through analysis to much simpler and more efficient low-level operations.

Implementations of Linda closely followed hardware developments in parallel computing. The first implementation was on a custom, bus-based architecture called the S/Net [4], followed by implementations on shared memory multiprocessors (Sequent Balance, Encore Multimax), distributed memory multiprocessors (Intel iPSC/1 and 2 hypercubes, Ncube, IBM SP1), collections of workstations, and true cluster architectures.

In terms of host languages, Linda has been embedded in many of the most commonly used computer languages. Initial imbedding was done in C, but important embeddings were built in Fortran, Lisp, and Prolog. In the 1990s, Java became an important host language for Linda, including the Jini, JavaSpaces [6], and Gigaspaces [7] implementations. More recently, Scientific Computing Associates (SCAI) created a variant of Linda

called NetWorkSpaces that is designed for interactive languages such as Perl, Python, R (see below).

## Open Linda

Initial compile-time Linda systems can be termed “closed,” in the sense that the Linda compiler needed to see all Linda operations that would ever occur in the program, in order to perform the analysis and optimization. In addition, only a single global tuple space was provided, which was created when the program started and destroyed when it completed. Linda programs tended to be collections of closely related processes that stemmed from the same code, and started, ran for a finite period, and stopped together.

During the 1990s, interest in distributed computing increased dramatically, particularly computing on collections of workstations. These applications consisted of heterogeneous collections of processes that derived from unrelated code, started and stopped independently, and might run indefinitely. The components could be written by different users and using different languages. In 1995, SCAI developed an Open Linda (Open referred to the fact that the universe of tuples remained flexible and open at runtime, rather than fixed and closed at compile time), which was later trademarked as “Paradise.” With Open Linda, tuple spaces became first-class objects. They could be created and destroyed as desired, or persist beyond the lifetime of their creator. Other Open Lindas that were created include JavaSpaces [6], TSpaces [8], and Gigaspaces [7].

In 2005 SCAI created a simplified Open Linda called NetWorkSpaces (NWS) [9], designed specifically for high productivity languages such as Perl, R, Python, Matlab, etc. In NetWorkSpaces, rather than communicating via tuples in tuple spaces, processes use shared variables stored in network spaces. Rather than use associative lookup via matching, in NetWorkSpaces shared data is represented as a name/value binding. Processes can create, consult, and remove bindings in shared network spaces.

[Figure 3](#) contains an excerpt from a NWS code for performing a maximization search, using the Python API. In NWS, store, find, and fetch correspond to Linda's out, rd, and in, respectively. In the code, a single shared variable “max” is used to hold the current best value found by any worker. The call NetWorkSpace (“my

```

def init():
 ws=NetWorkSpace('my space')
 ws.store('max', START)

def worker():
 ws=NetWorkSpace('my space')
 for x in MyCandidateList:
 currentMax = ws.find('max')
 y = f(x)
 if y > currentMax:
 currentMax = ws.fetch('max')
 if y > currentMax: currentMax = y
 ws.store('max', currentMax)

```

**Linda. Fig. 3** Excerpt of NetWorkSpaces code for performing a maximization search

space") connects to the space of that name, creating it if necessary, and returning a handle to it.

### Key Characteristics of Linda, and Comparison to Other Approaches

Linda presents a number of interesting characteristics that contrast with other approaches to parallel computing:

**Portability:** A Linda program can run on any parallel computer, regardless of the underlying parallel infrastructure and interconnection, whether it be distributed- or shared-memory or something in between. Most competing parallel approaches are strongly biased toward either shared- or distributed-memory.

**Generative Communication:** In Linda, when processes communicate, they create tuples, which exist apart from the process that created them. This means that the communication can exist long after the creating process has disappeared, and can be used by processes unaware of that process' existence. This aids debugging, since tuples are objects that can be inspected using tools like TupleScope [10].

**Distribution in space and time:** Processes collaborating via tuple spaces can be widely separated over both these dimensions. Because Linda forms a virtual shared memory, processes need not share any physical memory. Because tuples exist apart from processes, those processes can communicate with one another even if they don't exist simultaneously.

**Content Addressability:** Processes can select information based on its content, rather than just a tag or

the identity of the process that sent it. For example, one can build a distributed hash table using tuples.

**Anonymity:** Processes can communicate with one another without knowing one another's location or identity. This allows for a very flexible style of parallel program. For example, in a system called Piranha [11], idle compute cycles are harvested by creating and destroying workers on the fly during a single parallel run. The set of processes participating at any given time changes constantly. Linda allows these dynamic workers to communicate with the master, and even with one another, because processes need not know details about their communication partners, so long as they agree on the tuples they use to communicate.

**Simplicity:** Linda consists of only four operations, in, out, rd, eval, and two variants, inp and rdp. No additional synchronization functions are needed, nor any administrative functions.

When Linda was invented in the early 1980s, there were two main techniques used for parallel programming: shared memory with low-level locking, and message passing. Each parallel machine vendor chose one of these two styles and built their own, idiosyncratic API. For example, the first distributed-memory parallel computers (Intel iPSC, nCube, and IBM SP1) each had their own proprietary message passing library, which was incompatible with all others. Similarly, the early shared-memory parallel computers (Sequent Balance, Encore Multimax) each had their own way of allocating shared pages and creating synchronization locks.

Linda presented a higher level approach to parallelism that combined the strengths of both. Like shared memory, Linda permits direct access to shared data (no need to map such data onto an "owner" process, as is the case for message passing). Like message passing, Linda self-synchronizes (no need to worry about accessing uninitialized shared data, as is the case for hardware shared memory). Once the corresponding Linda implementation had been ported to a given parallel computer, any Linda program could be simply recompiled and run correctly on that computer.

At the time of this writing, it is fair to say that message passing, as embodied by Message Passing Interface (MPI), has become the dominant paradigm for writing parallel programs. MPI addressed the portability

problem in message passing by creating a uniform library interface for sending and receiving messages. Because of the importance of MPI to parallel programming, it is worth contrasting Linda and MPI.

*Compiler Support:* Linda's compiler support performs both type and syntax checking. Since the Linda compiler understands the types of the data being passed through tuple space, marshalling can be done automatically. The compiler can also detect outs that have no matching in, or vice versa, and issue a warning. In contrast, MPI is a runtime library, and the burdens of type checking and data marshalling fall largely to the programmer.

*Decoupled/anonymous communication:* In Linda, communicating processes are largely decoupled. In MPI, processes must know one another's location and exist simultaneously. In addition, receivers must anticipate sends. If, on the hand, the coordination needed by an application lends itself to a message passing style solution, this is still easy to express in Linda using associative matching. In this case, the compile and runtime optimization techniques discussed will often result in wire traffic similar, if not identical, to that obtained if message passing was used.

*Simplicity:* MPI is a large, complex API. Debugging usually requires the use of a sophisticated parallel debugger. Even simple commands such as `send()` exist in a number of subtly differently variants. In the spirit of Alan Perlis's observation "If you have a procedure with 10 parameters, you probably missed some," the MPI standard grew substantially from MPI 1 to MPI 2. Acknowledging the lack of a simple way to share data, MPI 2 introduced one-sided communication. This was not entirely successful [12]. In contrast, Linda consists of four simple commands.

## Future

In 2010, parallel computing stands at an interesting crossroads. The largest supercomputer in the world contains 224,000 cpus [13]. Meanwhile, multicore systems are ubiquitous; modern high performance computers typically contain 8–32 cpu cores, with hundreds envisioned in the near future. These two use cases represent two very different extrema on the spectrum of parallel programming. The largest supercomputers require

extremely efficient parallel codes in order to scale to tens of thousands of cpus, implying expert assistance by highly skilled programmers, careful coding, optimization and verification of programs, and are well suited to large-scale development efforts in languages such as C and tools such as MPI.

In contrast, nearly everyone with a compute-bound program will have access to a non-trivial parallel computer containing tens or hundreds of cpus, and will need a way to make effective use of those cpus.

An interesting characteristic of many of these users is that, rather than using traditional high performance languages such as C, Fortran, or even Java, they use scripting languages such as Perl, Python or Ruby, or high-productivity environments such as Matlab or R. These languages dramatically improve their productivity in terms of writing and debugging code, and therefore in terms of their overall goals of exploring data or testing theories. That is, they have explicitly opted for simplicity and expressivity in their tool of choice for sequential programming. However, these environments pose major challenges in terms of performance and memory usage. Parallelism on a relatively modest scale can permit these users to compute effectively while remaining comfortably within an environment they find highly productive. But, to be consistent with the choice made when selecting these sequential computing environments, the parallelism tool must also emphasize simplicity and expressivity. High level coordination languages like Linda can provide them with simple tools for accessing that parallelism.

## Influences

Some of the work influencing Linda includes: Sussman and Steele's language, Scheme, particularly continuations [14], Dijkstra's semaphores [15], Halstead's futures [16], C.A.R. Hoare's CSP [17], and Shapiro's Concurrent Prolog [18], particularly the concept of unification.

## History

Linda was initially conceived by David Gelernter while a graduate student at SUNY Stony Brook in the early 1980s. After taking a faculty position at Yale University in 1982, Gelernter, along with a number of graduate students, most notably Carriero, built a number of Linda implementations as well as high level applications that used Linda.

In the early 1990s Scientific Computing Associates (SCAI), founded by Yale professors Stan Eisenstat and Martin Schultz, began commercial development of Linda. Linda systems created by SCAI were sold in a variety of industries, most notably finance and energy. Large-scale systems, running on hundreds of computers, were built and used in production environments for years at a time, performing tasks such as portfolio pricing, risk analysis, and seismic image processing.

Around 2000, interest grew in more flexible languages and approaches. Java became the rage, particularly in financial organizations, and a number of commercial Java Linda systems were built. Although they varied in detail, all shared many common features: an object oriented design, open tuple spaces, and support for transactions.

## Related Entries

- [CSP \(Communicating Sequential Processes\)](#)
- [Logic Languages](#)
- [Multilisp](#)
- [Top500](#)

## Bibliographic Notes and Further Reading

The following papers are good general introductions to Linda: [19, 20]. Carriero and Gelernter used Linda in their introductory parallel programming book [21]. There have been conferences organized around the ideas embodied in Linda [22, 23].

## Bibliography

1. Gelernter D (1985) Generative communication in Linda. *ACM Trans Program Lang Syst* 7(1):80–112
2. Wirth N (1976) Algorithms [plus] data structures. Prentice-Hall, Englewood Cliffs
3. Jeong K, Shasha D (1994) PLinda 2.0: A transactional/checkpointing approach to fault tolerant Linda. In: 13th symposium on reliable distributed systems, Dana Point, pp 96–105
4. Carriero N (1987) Implementation of tuple space machines. Technical Report. Department of Computer Science, Yale University
5. Bjornson RD (1993) Linda on distributed memory multiprocessors. PhD dissertation, Department of Computer Science, Yale University
6. Freeman E, Arnold K, Hupfer S (1999) JavaSpaces principles, patterns, and practice. Addison-Wesley Longman, Reading
7. Gigaspaces Technologies Ltd. [cited; Available from: [www.gigaspaces.com](http://www.gigaspaces.com)]
8. TSpaces. [cited; Available from: <http://www.almaden.ibm.com/cs/TSpaces>]

9. Bjornson R et al (2009) NetWorkSpace: a coordination system for high-productivity environments. *Int J Parallel Program* 37(1):106–125
10. Bercovitz P, Carriero N (1990) TupleScope: a graphical monitor and debugger for Linda-based parallel programs. Technical Report. Department of Computer Science, Yale University
11. Carriero N et al (1995) Adaptive parallelism and Piranha. *IEEE Comput* 28(1):40–49
12. Bonachea D, Duell J (2004) Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *Int J High Perform Comput Network* 1(1/2/3):91–99
13. Top 500 Supercomputing sites. [cited; Available from: [www.top500.org](http://www.top500.org)]
14. Steele GL, Sussman GJ (1975) Scheme: An interpreter for extended lambda calculus. Massachusetts Institute of Technology, Artificial Intelligence Laboratory. <http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-349.pdf>
15. Dijkstra EW (2002) Cooperating sequential processes. In: Per Brinch Hansen (ed) *The origin of concurrent programming: from semaphores to remote procedure calls*. Springer, New York, pp 65–138
16. Halstead Jr RH (1985) Multilisp: A language for concurrent symbolic computation. *TOPLAS* 7(4):501–538
17. Hoare CAR (1978) Communicating sequential processes. *Commun ACM* 21(8):666–677
18. Shapiro E (1988) Concurrent prolog: a progress report. In: *Concurrent prolog*. MIT Press, Cambridge, pp 157–187
19. Gelernter D, Carriero N (1992) Coordination languages and their significance. *Commun ACM* 35(2):97–107
20. Bjornson R, Carriero N, Gelernter D (1997) From weaving threads to untangling the web: a view of coordination from Linda's perspective. In: *Proceedings of the second international conference on coordination languages and models*, Springer, New York
21. Carriero N, Gelernter DH (1990) *How to write parallel programs: a first course*. MIT Press, Cambridge, Massachusetts; London, 232 pp
22. Garlan D, Le Metayer D (1997) *Coordination: languages and models*. Springer, Berlin
23. Ciancarini P, Hankin C (1996) *Coordination languages and models*. Springer, Berlin

## Linear Algebra Software

JACK DONGARRA, PIOTR LUSZCZEK  
University of Tennessee, Knoxville, TN, USA

### Definition

Linear algebra software is software that performs numerical calculations aimed at solving a system of linear equations or related problems such as eigenvalue,

singular value, or condition number computations. Linear algebra software operates within the confines of finite precision floating-point arithmetic and is characterized by its computational complexity with respect to the sizes of matrix and vectors involved. Equally important metric is that of numerical robustness: The linear algebra methods aim to deliver a solution that is as close as possible (in a numerical sense by taking into account the accuracy of the input data) to the true solution (obtained in sufficiently extended floating-point precision) if such a solution exists.

## Discussion

The increasing availability of high-performance computers has a significant effect on all spheres of scientific computation, including algorithm research and software development in numerical linear algebra. Linear algebra – in particular, the solution of linear systems of equations – lies at the heart of most calculations in scientific computing. There are two broad classes of algorithms: those for dense, and those for sparse matrices. A matrix is called sparse if it has a substantial number of zero elements, making specialized storage and algorithms necessary.

Much of the work in developing linear algebra software for high-performance computers is motivated by the need to solve large problems on the fastest computers available. For the past 15 years or so, there has been a great deal of activity in the area of algorithms and software for solving linear algebra problems. The goal of achieving high performance on codes that are portable across platforms has largely been realized by the identification of linear algebra kernels, the Basic Linear Algebra Subprograms (BLAS). See discussion of the EISPACK, LINPACK, LAPACK, and ScaLAPACK libraries, which are expressed in successive levels of the BLAS.

The key insight to designing linear algebra algorithms for high-performance computers is that the frequency with which data are moved between different levels of the memory hierarchy must be minimized in order to attain high performance. Thus, the main algorithmic approach for exploiting both vectorization and parallelism in various implementations is the use of block-partitioned algorithms, particularly in conjunction with highly tuned kernels for performing

matrix–vector and matrix–matrix operations (the Level 2 and 3 BLAS).

Algorithms for sparse matrices differ drastically because they have to take full advantage of sparsity and its particular structure. Optimization techniques differ accordingly due to changes in the memory access patterns and data structures. Commonly, then, the sparse methods are discussed separately.

## Dense Linear Algebra Algorithms

### Origins of Dense Linear Systems

A major source of large dense linear systems is problems involving the solution of boundary integral equations [1]. These are integral equations defined on the boundary of a region of interest. All examples of practical interest compute some intermediate quantity on a two-dimensional boundary and then use this information to compute the final desired quantity in three-dimensional space. The price one pays for replacing three dimensions with two is that what started as a sparse problem in  $O(n^3)$  variables is replaced by a dense problem in  $O(n^2)$ .

Dense systems of linear equations are found in numerous applications, including:

- Airplane wing design
- Radar cross-section studies
- Flow around ships and other off-shore constructions
- Diffusion of solid bodies in a liquid
- Noise reduction
- Diffusion of light through small particles

The electromagnetics community is a major user of dense linear systems solvers. Of particular interest to this community is the solution of the so-called radar cross-section problem. In this problem, a signal of fixed frequency bounces off an object; the goal is to determine the intensity of the reflected signal in all possible directions. The underlying differential equation may vary, depending on the specific problem. In the design of stealth aircraft, the principal equation is the Helmholtz equation. To solve this equation, researchers use the *method of moments* [2, 3]. In the case of fluid flow, the problem often involves solving the Laplace or Poisson equation. Here, the boundary integral solution is known as the *panel methods* [4, 5], so named from the quadrilaterals that discretize and approximate a



structure such as an airplane. Generally, these methods are called *boundary element methods*.

Use of these methods produces a dense linear system of size  $O(N)$  by  $O(N)$ , where  $N$  is the number of boundary points (or panels) being used. It is not unusual to see size  $3N$  by  $3N$ , because of three physical quantities of interest at every boundary element.

A typical approach to solving such systems is to use LU factorization. Each entry of the matrix is computed as an interaction of two boundary elements. Often, many integrals must be computed. In many instances, the time required to compute the matrix is considerably larger than the time for solution.

The builders of stealth technology who are interested in radar cross sections are using direct Gaussian elimination methods for solving dense linear systems. These systems are always symmetric and complex, but not Hermitian.

## Basic Elements in Dense Linear Algebra Methods

Common operations involving dense matrices are the solution of linear systems  $Ax = b$ , the least squares solution of over- or underdetermined systems  $\min_x \|Ax - b\|_2$ , and the computation of eigenvalues and vectors  $Ax = \lambda x$ . Although these problems are formulated as matrix–vector equations, their solution involves a definite matrix–matrix component. For instance, in order to solve a linear system, the coefficient matrix is first factored as  $A = LU$  (or  $A = U^T U$  in the case of symmetry) where  $L$  and  $U$  are lower and upper triangular matrices, respectively. It is a common feature of these matrix–matrix operations that they take, on a matrix of size  $n$  by  $n$ , a number of operations proportional to  $n^3$ , a factor  $n$  more than the number of data elements involved.

It is possible to identify three levels of linear algebra operations:

- Level 1: vector–vector operations such as the update  $y \leftarrow y + \alpha x \times y$  and the inner product  $\alpha = x^T y$ . These operations involve (for vectors of length  $n$ )  $O(n)$  data and  $O(n)$  operations.
- Level 2: matrix–vector operations such as the matrix–vector product. These involve  $O(n^2)$  operations on  $O(n^2)$  data.

- Level 3: matrix–matrix operations such as the matrix–matrix product  $C = AB$ . These involve  $O(n^3)$  operations on  $O(n^2)$  data.

These three levels of operations have been realized in a software standard known as the Basic Linear Algebra Subprograms (BLAS) [6–8]. Although BLAS routines are freely available on the net, many computer vendors supply a tuned, often assembly-coded, BLAS library optimized for their particular architecture. The relation between the number of operations and the amount of data is crucial for the performance of the algorithm.

## Optimized Implementations

Due to its established position, BLAS form the foundation for optimized dense linear algebra operation on homogeneous as well as heterogeneous multicore architectures. Similarly, LAPACK API serves as the defacto standard for accessing various decompositional techniques. Currently active optimized implementations include ACML from AMD, cuBLAS from NVIDIA, LibSci from Cray, MKL from Intel, and open source ATLAS. In the distributed memory regime, Basic Linear Algebra Communication Subroutines (BLACS), PBLAS, and ScaLAPACK serve the corresponding role of defining an API and are accompanied by vendor-optimized implementations.

## Sparse Linear Algebra Methods

### Origin of Sparse Linear Systems

The most common source of sparse linear systems is the numerical solution of partial differential equations. Many physical problems, such as fluid flow or elasticity, can be described by partial differential equations (PDE). These are implicit descriptions of a physical model, describing some internal relation such as stress forces. In order to arrive at an explicit description of the shape of the object or the temperature distribution, the PDE needs to be solved, and for this, numerical methods are required.

### Discretized Partial Differential Equations

Several methods for the numerical solution of PDEs exist, the most common ones being the methods of finite elements, finite differences, and finite volumes. A common feature of these is that they identify discrete



points in the physical object, and give a set of equations relating these points.

Typically, only points that are physically close together are related to each other in this way. This gives a matrix structure with very few nonzero elements per row, and the nonzeros are often confined to a “band” in the matrix.

### Sparse Matrix Structure

Matrices from discretized partial differential equations contain so many zero elements that it pays to find a storage structure that avoids storing these zeros. The resulting memory savings, however, are offset by an increase in programming complexity, and by decreased efficiency of even simple operations such as the matrix–vector product.

More complicated operations, such as solving a linear system, with such a sparse matrix present a next level of complication, as both the inverse and the *LU* factorization of a sparse matrix are not as sparse, thus needing considerably more storage. Specifically, the inverse of a band sparse matrix is a full matrix, and factoring such a sparse matrix fills in the band completely.

### Basic Elements in Sparse Linear Algebra

#### Methods

Methods for sparse systems use, like those for dense systems, vector–vector, matrix–vector, and matrix–matrix operations. However, there are some important differences.

For iterative methods, there are almost no matrix–matrix operations. Since most modern architectures prefer these Level 3 operations, the performance of iterative methods will be limited from the outset.

An even more serious objection is that the sparsity of the matrix implies that indirect addressing is used for retrieving elements. For example, in the popular row-compressed matrix storage format, the matrix–vector multiplication looks like

```
for i=1 ... n
 p←pointer to row i
 for j=1 ... ni
 yi ← yi + a(p+j)* (c(p+j))
```

where  $n_i$  is the number of nonzeros in row  $i$ , and  $p(.)$  is an array of column indices. A number of such algorithms for several sparse data formats are given in [9].

Direct methods can have a BLAS 3 component if they are a type of dissection method. However, in a given sparse problem, the denser the matrices are, the smaller they are on average. They are also not general full matrices, but only banded. Thus, very high performance should not be expected from such methods either.

### Direct Sparse Solution Methods

For the solution of a linear system, one needs to factor the coefficient matrix. Any direct method is a variant of Gaussian elimination. As remarked above, for a sparse matrix, this fills in the band in which the nonzero elements are contained. In order to minimize the storage needed for the factorization, research has focused on finding suitable orderings of the matrix. Re-ordering the equations by a symmetric permutation of the matrix does not change the numerical properties of the system in many cases, and it can potentially give large savings in storage. In general, direct methods do not make use of the numerical properties of the linear system, and thus, their execution time is affected in a major way by the structural properties of the input matrix.

### Iterative Solution Methods

Direct methods, as sketched above, have some pleasant properties. Foremost is the fact that their time to solution is predictable, either *a priori*, or after determining the matrix ordering. This is due to the fact that the method does not rely on numerical properties of the coefficient matrix, but only on its structure. On the other hand, the amount of fill can be substantial, and with it the execution time. For large-scale applications, the storage requirements for a realistic size problem can simply be prohibitive.

Iterative methods have far lower storage demands. Typically, the storage, and the cost per iteration with it, is of the order of the matrix storage. However, the number of iterations strongly depends on properties of the linear system, and is at best known up to an order estimate; for difficult problems, the methods may not even converge due to accumulated round-off errors.

### Basic Iteration Procedure

In its most informal sense, an iterative method in each iteration locates an approximation to the solution of the problem, measures the error between the approximation and the true solution, and based on the



error measurement improves on the approximation by constructing a next iterate. This process repeats until the error measurement is deemed small enough.

### Stationary Iterative Methods

The simplest iterative methods are the “stationary iterative methods.” They are based on finding a matrix  $M$  that is, in some sense, “close” to the coefficient matrix  $A$ . Instead of solving  $Ax = b$ , which is deemed computationally infeasible,  $Mx_1 = b$  is solved. The true measure of how well  $x_1$  approximates  $x$  is the error  $e_1 = x_1 - x$ , but, since the true solution  $x$  is not known, this quantity is not computable. Instead, the “residual”:  $r_1 = Ae_1 = Ax_1 - b$  is monitored since it is a computable quantity. It is easy to see that the true solution satisfies  $x = A^{-1}b = x_1 - A^{-1}r_1$ , so, replacing  $A^{-1}$  with  $M^{-1}$  in this relation,  $x_2 = x_1 - M^{-1}r_1$  can be defined.

Stationary methods are easily analyzed: It can be proven that  $r_i \rightarrow 0$  if all eigenvalues  $\lambda = \lambda(I - AM^{-1})$  satisfy  $|\lambda| < 1$ . For certain classes of  $A$  and  $M$ , this inequality is automatically satisfied [10, 11].

### Krylov Space Methods

The most popular class of iterative methods nowadays is that of “Krylov space methods.” The basic idea there is to construct the residuals such that the  $n$ -th residual  $r_n$  is obtained from the first by multiplication by some polynomial in the coefficient matrix  $A$ , that is,  $r_n = P^{n-1}(A)r_1$ . The properties of the method then follow from the properties of the actual polynomial [12–14].

Most often, these iteration polynomials are chosen such that the residuals are orthogonal under some inner product. From this, one usually obtains some minimization property, though not necessarily a minimization of the error.

Since the iteration polynomials are of increasing degree, it is easy to see that the main operation in each iteration is one matrix–vector multiplication. Additionally, some vector operations, including inner products in the orthogonalization step, are needed.

### Optimized Implementations

Sparse BLAS used to be available from some of the vendors. The more common approach is to have the dedicated sparse packages available publicly and have the vendors tune them for their software offerings. Examples of direct sparse solvers include MUMPS, SuperLU, UMFPACK, and Parafac. The iterative solvers in wide

use are PETSc and Trilinos. Also, autotuning efforts such as OSKI aim to provide basic building blocks for use with more complete iterative method frameworks.

### Available Software

You will find a comprehensive list of open source software that is freely available for solving problems in numerical linear algebra, specifically dense, sparse direct and iterative systems and sparse iterative eigenvalue problems at: <http://www.netlib.org/utk/people/JackDongarra/la-sw.html>

### Related Entries

- ▶ [LAPACK](#)
- ▶ [LINPACK Benchmark](#)
- ▶ [Linear Algebra, Numerical](#)
- ▶ [ScaLAPACK](#)

### Bibliography

1. Edelman A (1993) Large dense numerical linear algebra in 1993: the parallel computing influence. *Int J Supercomput Appl* 7:113–128
2. Harrington R (1990) Origin and development of the method of moments for field computation. *IEEE Antennas and Propagation Magazine*, June 1990
3. Wang JJH (1991) Generalized moment methods in electromagnetics. Wiley, New York
4. Hess JL (1990) Panel methods in computational fluid dynamics. *Annu Rev Fluid Mech* 22:255–274
5. Hess L, Smith MO (1967) Calculation of potential flows about arbitrary bodies. In: Kuchemann D (ed) *Progress in aeronautical sciences*, vol 8. Pergamon Press
6. Lawson CL, Hanson RJ, Kincaid D, Krogh FT (1979) Basic linear algebra subprograms for fortran usage. *ACM Trans Math Soft* 5:308–323
7. Dongarra JJ, Du Croz J, Hammarling S, Hanson RJ (1988) An extended set of FORTRAN basic linear algebra subroutines. *ACM Trans Math Soft* 14(1):1–17
8. Dongarra JJ, Du Croz J, Duff IS, Hammarling S (1990) A set of level 3 basic linear algebra subprograms. *ACM Trans Math Soft* 16(1):1–17
9. Barrett R, Berry M, Chan TF, Demmel J, Donato J, Dongarra J, Eijkhout V, Pozo R, Romine C, Vorst H (1994) Templates for the solution of linear systems: building blocks for iterative methods. SIAM, Philadelphia PA, <http://www.netlib.org/templates/templates.ps>.
10. Hageman LA, Young DM (1981) Applied iterative methods. Academic, New York
11. Young DM, Jea KC (1980) Generalized conjugate-gradient acceleration of nonsymmetrizable iterative methods. *Lin Alg Appl* 34:159–194



12. Axelsson O, Barker AV (1984) Finite element solution of boundary value problems. Theory and computation. Academic, Orlando, FL
13. Birkho G, Lynch RE (1984) Numerical solution of elliptic problems. SIAM, Philadelphia
14. Chan T, van der Vorst H (1997) Linear system solvers: Sparse iterative methods. In Keyes D et al. (eds) Parallel numerical algorithms, Proc. of the ICASW/LaRC Workshop on Parallel Numerical Algorithms, May 23–25, 1994, pp 91–118. Kluwer, Dordrecht, The Netherlands, 1997

A sparse matrix is one that is populated mainly with zeros, and hence the need for storing only the nonzero elements especially for large problems. In the context of sparse matrix computations, one usually refers to sparse matrices with irregular sparsity structure. Designing algorithms that realize high performance for structured sparse matrix computations is much easier than the case of irregular sparsity. Irregular sparsity leads to lack of data locality and expensive memory references across the memory hierarchies of modern parallel architectures. While the creation of a standard set of *Sparse Basic Linear Algebra Subroutines* (SBLAS) was not as successful as the classical dense BLAS the most important kernels in sparse matrix computations are: (i) sparse matrix–vector multiplication and (ii) sparse matrix–tall dense matrix multiplication. Next to generating and applying preconditioners, these two kernels have a substantial impact on the realizable performance of Krylov subspace iterative schemes on these architectures.

One should also state that in *Finite Element Analysis* it has been the tradition to resort to *matrix-free* computations where the sparse matrices involved are not stored explicitly but algorithms are provided to compute their action, or the action of functions of such matrices, on a vector or a block of vectors. Such *matrix-free* approach makes it rather difficult to use effective preconditioners for solving large sparse linear systems with the needed iterative methods. Thus, matrix-free formulations may not be the optimal choice when using the emerging petascale parallel architectures with their abundant memories.

## Dense Matrix Computations

Unlike sparse matrix computations, dense matrix algorithms benefit from the regular storage of the dense matrices involved to achieve high data locality and hence high performance through the fine-tuned three-level dense BLAS on parallel architectures. The third level of this set BLAS-3 includes matrix–matrix multiplications, which are the procedures that reach the highest rate of computation on such architectures. Even when dealing with sparse matrix computations, the goal is often to transform algorithms that are originally vector-oriented into their block versions so as to involve dense matrix multiplications. Moreover, the very popular Krylov methods for solving sparse linear systems or eigenvalue problems consist of projecting the

## Linear Algebra, Numerical

BERNARD PHILIPPE<sup>1</sup>, AHMED SAMEH<sup>2</sup>

<sup>1</sup>Campus de Beaulieu, Rennes, France

<sup>2</sup>Purdue University, West Lafayette, IN, USA

### Synonyms

Matrix computations

### Definition

Numerical linear algebra (or matrix computations) is the design and analysis of matrix operations and algorithms. Matrix computations plays a vital role in enabling the majority of computational science and engineering applications. Examples of such applications include: computational fluid dynamics, structural mechanics, fluid-structure interaction, computational electromagnetics, image processing, web search (PageRank), and information retrieval, just to list a few. Matrix computations can be divided into two classes: (a) dense matrix computations, and (b) sparse matrix computations, with the former being rich in data locality and hence can readily achieve high performance on modern architectures. The basic standard problems in numerical linear algebra are: (i) solving linear systems of equations, (ii) solving linear least squares problems with or without constraints, (iii) solving standard and generalized eigenvalue problems, and (iv) obtaining singular values or singular triplets of a matrix.

### Discussion

#### Sparse Matrix Computations

The overwhelming majority of science and engineering applications give rise to sparse matrix computations.

initial sparse problem onto some subspace of reduced dimension, leading to a similar dense matrix problem.

## Bibliographic Notes and Further Reading

For examples of parallel dense matrix computations, see [1, 3, 4], and for examples of parallel sparse matrix computations, see [2].

## Bibliography

1. Blackford L S, Choi J, Cleary A, D'Azevedo E, Demmel J, Dhillon I, Dongarra J, Hammarling S, Henry G, Petitet A, Stanley K, Walker D, Whaley RC (1996) *ScalAPACK Users' Guide*. Society for Industrial and Applied Mathematics, available on line at <http://www.netlib.org/lapack/lug/>
2. Dongarra JJ, Duff IS, Sorensen DC, Van der Vorst H (1998) Numerical Linear Algebra for High Performance Computers. SIAM, Philadelphia
3. Gallivan KA, Plemmons RJ, Sameh AH (1990) Parallel algorithms for dense linear algebra computations. In: Gallivan KA, Heath MT, Ng E, Ortega JM, Peyton BW, Plemmons RJ, Romine CH, Sameh AH, Voigt RG (ed) Parallel algorithms computations. SIAM, Philadelphia, PA
4. Golub GH, Van Loan CF (1996) Matrix computations, 3rd edn. John Hopkins, New York

## Definition

A **linear least squares problem** is given by

obtain  $x \in \mathbb{R}^n$ , such that the two-norm  $\|b - Ax\|$  is minimized, (1)

where  $A \in \mathbb{R}^{m \times n}$  and  $b \in \mathbb{R}^m$ . The vector  $x$  is a solution if and only if the residual  $r = b - Ax$  is orthogonal to the range of  $A$ :  $A^T r = 0$ . When  $A$  is of full column rank  $n$ , there exists a unique solution. When  $\text{rank}(A) < n$ , the solution of choice is that one with the smallest two-norm.

## Discussion

**Note:** The number of the floating-point operations (additions, multiplications, divisions, square roots) of the algorithms are simply called operations in what follows.

## Full Rank Standard Problem

Consider first the case in which  $n \leq m$  and  $A$  is of full column rank  $n$ .

**General scheme.** Since the two-norm is invariant under orthogonal transformations, the most common approach for solving this least squares problem consists of computing the orthogonal factorization of  $A$ . The procedure amounts to premultiplying  $A$  and  $b$  by a finite number – say  $p$  – of orthogonal transformations  $Q_1, \dots, Q_p$ , such that

$$Q_p \cdots Q_2 Q_1 A = \begin{pmatrix} R \\ 0 \end{pmatrix}, \quad (2)$$

where  $R \in \mathbb{R}^{n \times n}$  is a nonsingular upper triangular or upper trapezoidal matrix. Therefore,

$$\|b - Ax\| = \|Q_p \cdots Q_1(b - Ax)\| = \left\| \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} - \begin{pmatrix} R \\ 0 \end{pmatrix} x \right\|$$

where  $c_1 \in \mathbb{R}^n$ , and the unique solution of (1) is given by solving  $Rx = c_1$ . In what follows, the orthogonal reduction to the triangular form will only be considered. For solving the resulting triangular system, please refer to [BLAS 12.c.ii.1.a, linear solvers 12.c.ii.2.a].

## Linear Equations Solvers

- Dense Linear System Solvers
- Multifrontal Method
- Numerical Libraries
- Preconditioners for Sparse Iterative Methods
- Sparse Direct Methods

## Linear Least Squares and Orthogonal Factorization

BERNARD PHILIPPE<sup>1</sup>, AHMED SAMEH<sup>2</sup>

<sup>1</sup>Campus de Beaulieu, Rennes, France

<sup>2</sup>Purdue University, West Lafayette, IN, USA

## Synonyms

Linear regression; Overdetermined systems; Underdetermined systems

## Householder Reductions

**The sequential algorithm.** Given a vector  $a \in \mathbb{R}^s$ , a vector  $v \in \mathbb{R}^s$  can be determined such that the transformation  $H = H(v) = I_s - \beta vv^T$ , where  $\beta = (2/v^T v)$ , yields  $Ha = \pm \|a\|e_1$  where  $e_1$  is the first canonical vector of  $\mathbb{R}^s$  (there are two such transformations from which one is selected for numerical stability [16]).  $H$  is an elementary reflector, i.e.,  $H$  is orthogonal and  $H^2 = I_s$ . Application of  $H$  to a given vector  $x \in \mathbb{R}^s$  is given by  $H(v)x = x - \beta(v^T x)v$  and computed using two Level 1 BLAS routines [19], namely, DDOT and DAXPY, involving only  $4s$  operations and not  $2s^2$  as the case in matrix-vector multiplication.

The triangularization of  $A$  is obtained by successively applying such orthogonal transformations to  $A$ :  $A_0 = A$  and for  $k = 1 : \min(n, m-1)$ ,  $A_{k+1} = H_k A_k$  where

$$H_k = \begin{pmatrix} I_{k-1} & 0 \\ 0 & H(v_k) \end{pmatrix} \text{ with } v_k \text{ chosen such that all but}$$

the first entry of  $H(v_k)A_k(k : m, k)$  are zero. At step  $k$ , the multiplication  $H_k A_k$  is obtained from  $H(v_k)A_k(k : m, k : n)$  which involves  $4(n-k)(m-k) + O(m-k)$  operations. It can be implemented by two Level 2 BLAS routines [9], namely, the matrix-vector multiplication DGEMV and the rank-one update DGER. The procedure stores the matrix  $A_{k+1}$  in place of  $A_k$ .

The total procedure involves  $2n^2(m-n/3) + O(mn)$  operations for obtaining  $R$ . To apply simultaneously the transformations to the right-hand-side  $b$  of problem (1), one may append that vector as the  $n+1$ st column of  $A$ . When necessary for subsequent calculations, the sequence of the vectors  $(v_k)_{1,n}$  can be stored for instance in the lower part of the transformed matrix  $A_k$ . This is the case when  $n \ll m$  and an orthogonal basis  $\tilde{Q} = [q_1, \dots, q_n] \in \mathbb{R}^{m \times n}$  of the range of  $A$  is sought: the

basis is obtained by premultiplying the matrix  $\begin{pmatrix} I_n \\ 0 \end{pmatrix}$

successively by  $H_n, H_{n-1}, \dots, H_1$ . This process involves  $4(m^2n - mn^2 + n^3/3) + O(mn)$  operations.

Block versions of the procedure have been introduced by Bischof and Van Loan [4] (the  $WY$  form) and by Schreiber and Parlett [25] (the  $GG^T$  form). The generation and application of the  $WY$  form are implemented in the routines DLARFT and DLARFB of LAPACK [1]. The latter involves Level 3 BLAS primitives [9]. The  $WY$  form consists of considering a narrow window in which the single-vector algorithm is used

before applying the corresponding transformations to the remaining part of  $A$ : if  $s$  is the window width,  $s$  steps are accumulated in the form  $H_{k+s} \cdots H_{k+1} = I + WY^T$  where  $W, Y \in \mathbb{R}^{m \times s}$ . This expression allows the use of Level 3 BLAS in updating the remaining part of  $A$ . In LAPACK, the driver routine which implements the resolution of the least square problem (1) is DORGQR.

**The parallel algorithm.** Theoretically, by using  $O(mn)$  processors, the computation may be performed in  $O(n \log m)$  time steps. This is a direct extension to rectangular matrices of Sameh's result [22]: in each of the  $n$  update phases, each scalar product can be performed in  $O(\log m)$  time steps on  $m$  processors and there are at most  $n$  columns to be processed in parallel.

Since, except for very special computing architectures (e.g., systolic arrays), independent tasks cannot be reduced to one operation; a plausible approach is to wrap the columns of  $A$  across a ring of processors as shown in [14]. The block version of the algorithm above is the preferred parallel scheme which is implemented in ScaLAPACK [7] via the routines PDGEQRF and PDGELS where  $A$  and  $b$  are distributed on a two-dimensional grid of processes according to the block cyclic scheme [7] (see [4. Parallel programming / i. Data distribution]). The block size is chosen large enough to allow Level 3 BLAS routines to achieve maximum performances on each involved uniprocessor (at least 90% as recommended by the authors of ScaLAPACK [8]). While increasing the block size improves the granularity of the computation, it may negatively affect concurrency. An optimal tradeoff therefore must be found depending on the architecture of the computing platform.

## Givens Rotations

**The sequential algorithm.** In the standard Givens' process, the orthogonal matrices in (2) consist of  $p = mn - n(n+1)/2$  plane rotations extended to the entire space  $\mathbb{R}^m$ . The triangularization of  $A$  is obtained by successively applying plane rotations  $R_k$ , so that  $A_{k+1} = R_k A_k$  for  $k = 1 : mn - n(n+1)/2$  with  $A_0 = A$ . Each of these rotations will be chosen to introduce a single zero below the diagonal of  $A$ : at step  $k$ , eliminating the entry  $a_{i\ell}^{(k)}$  of  $A_k$  where  $1 \leq \ell \leq n$  and  $1 < i \leq m$  can be done by the rotation

$$R_{ij}^{(k)} = \begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & & \ddots & & \\ & & & & 1 & \\ & & -s & & & c \\ & & & & & 1 \\ & & & & & & 1 \end{pmatrix} \quad (3)$$

where  $1 \leq j \leq n$ , and  $c^2 + s^2 = 1$  are such that

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} a_{j\ell}^{(k)} \\ a_{i\ell}^{(k)} \end{pmatrix} = \begin{pmatrix} a_{j\ell}^{(k+1)} \\ 0 \end{pmatrix}.$$

Two such rotations exist from which one is selected for numerical stability [16]. The construction of a rotation and its application to a couple of rows are implemented in the Level 1 BLAS [19] via the procedures DROTG and DROT, respectively.

The order of annihilation is quite flexible as long as applying a new rotation does not destroy a previously introduced zero. The simplest way is to organize columnwise eliminations. The total number of operations is  $3mn^2 - n^3 + O(mn)$  which is roughly 1.5 times that required by the Householder's reduction.

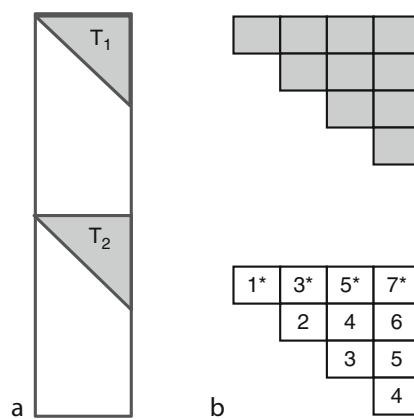
**Parallel algorithms.** One parallel algorithm consists of organizing the  $mn - n/(n + 1)/2$  rotations into sweeps which are sets of at most  $\lfloor m/2 \rfloor$  parallel rotations. For square matrices, Sameh and Kuck [24] and later, Cosnard and Robert [12], introduced such  $O(n)$  procedures using  $O(n^2)$  processors. In the general situation of rectangular matrices, Cosnard et al. in [11] proved the theoretical optimality of the greedy algorithm introduced by Modi and Clarke in [21] over any other algorithm. More details on the chronological appearance of the Givens algorithms and

their comparison with the Householder reductions are given in [14].

## Hybrid Algorithms ( $m \gg n$ )

**Dense matrix.** This is a situation that arises in some computations like in the adjustment of geodetic networks [15], or in the parallel construction of a Krylov basis where often  $m$  is larger than a million and  $n$  is smaller than 100 [26].

The algorithm was introduced by Sameh [23]. It is depicted in Fig. 1 in the case where two processors are used. It consists of partitioning  $A$  into contiguous blocks of  $r$  rows where  $r \geq n$ , with each block assigned to one processor. In the first step (Fig. 1a), each block is triangularized by Householder reduction. This step consumes the major part of the overall computation and is void of any communications. The second step is devoted to annihilating all the resulting triangular factors except the first one. There are several possibilities for achieving this goal. In [15], for annihilating the entries of the triangular matrix  $T_2$ , the rotations are chosen as depicted in Fig. 1b. The entry (1,1) of the first row of  $T_2$  is annihilated by rotating the first rows of  $T_1$  and  $T_2$ . Then all the other entries of the first diagonal of  $T_2$  can easily be annihilated by rotating rows of  $T_2$ . As soon as the entry (2,2) of  $T_2$  is annihilated, the entry (1,2) of  $T_2$  can be annihilated before the annihilation of the first diagonal of the matrix is completed. Rotations indicated with stars are the only ones which require synchronization.



## Linear Least Squares and Orthogonal Factorization. Fig. 1

By following that strategy, Sidje [26] developed the software RODDEC which performs the QR factorization on a ring of processors.

**Block angular matrix.** In some applications, a domain decomposition provides a matrix such that under some appropriate numbering of the domains and corresponding interfaces, the matrix  $A$  is of the form:

$$A = \begin{pmatrix} B_1 & C_1 \\ B_2 & C_2 \\ \ddots & \vdots \\ B_r & C_r \end{pmatrix} \quad (4)$$

where  $A$  as well as each block  $B_i$ , for  $i = 1, \dots, r$  is of full column rank. The special structure implies that the QR factorization of  $A$  yields an upper triangular matrix with the same block structure; this can be seen as shown by Golub et al. in [15] from the Cholesky factorization of  $A^T A$  which is a block arrowhead matrix. Therefore, similar to the first step of the algorithm previously described, for  $i = 1, \dots, r$ , processor  $i$  first performs the QR factorization of the block  $B_i$ . The corresponding blocks of  $[B_i, C_i]$  are now transformed into  $\begin{pmatrix} R_i \\ 0 \end{pmatrix}, \begin{pmatrix} D_{i1} \\ D_{i2} \end{pmatrix}$ . By virtually reordering the block

rows, the resulting matrix can be expressed by :

$$A_1 = \begin{pmatrix} R_1 & D_{11} \\ R_2 & D_{21} \\ \ddots & \vdots \\ R_r & D_{r1} \\ D_{12} & \\ D_{22} & \\ \vdots & \\ D_{r2} & \end{pmatrix}. \quad (5)$$

To complete the QR factorization of  $A$ , it is therefore necessary to obtain the QR factorization of the submatrix defined by the sub-blocks  $D_{i2}$  for  $i = 1, \dots, r$ . This is exactly the situation where the procedure RODDEC

can be used. Other approaches can also be considered as mentioned in [6]. For instance Golub et al. in [15] introduce a version of the algorithm adapted to the hierarchical definition of the subdomains. The algorithm takes advantage of the locality of the frontiers to minimize communications.

**Gram-Schmidt procedures.** The goal of these procedures is to obtain the orthogonal factorization  $A = QR$  of a full rank matrix  $A = [a_1, \dots, a_n] \in \mathbb{R}^{m \times n}$  where  $Q = [q_1, \dots, q_n] \in \mathbb{R}^{m \times n}$  has orthonormal columns, and  $R \in \mathbb{R}^{n \times n}$  is a nonsingular upper-triangular matrix with positive diagonal elements thus insuring the uniqueness of the factorization. This factorization is often referred to as the *skinny* QR factorization. Since orthogonality of the vectors  $q_j$  deteriorates rather quickly, reorthogonalization is often necessary. The general structure of the scheme is given by

ALGORITHM : Gram-Schmidt

```
do k = 0 : n-1,
 w = P_k(a_{k+1});
 q_{k+1} = w / \|w\|;
end
```

where  $P_k$  is the orthogonal projector onto the orthogonal complement of the subspace spanned by  $\{a_1, \dots, a_k\}$  ( $P_0$  is the identity).  $R = Q^T A$  can be built step by step during the process.

Two different versions of the algorithm are obtained by expressing  $P_k$  in distinct ways:

**Classical\_GS:** for the Classical Gram-Schmidt  $P_k = I - Q_k Q_k^T$ , where  $Q_k = [q_1, \dots, q_k]$ . Unfortunately, this version has been proved by Bjorck [5] to be numerically unreliable except when it is applied two times which makes it numerically equivalent to **Modified\_GS**.

**Modified\_GS:** for the Modified Gram-Schmidt  $P_k = (I - q_k q_k^T) \cdots (I - q_1 q_1^T)$ . Numerically, the columns of  $Q$  are orthogonal up to a tolerance determined by the machine precision multiplied by the condition number of  $A$  [5]. By applying the algorithm a second time (i.e., with complete reorthogonalization), the columns of  $Q$  become orthogonal up to the tolerance determined by the machine precision parameter similar to the Householder or Givens reductions.

The basic procedures involve  $mn^2 + O(mn)$  operations. `Classical_GS` is based on Level 2 BLAS procedures but it must be applied two times while `Modified_GS` is based on Level 1 BLAS routines. By inverting the two loops of `Modified_GS`, the procedure can proceed with a Level 2 BLAS routine. This is only possible when  $A$  is available explicitly (for instance, this is not possible with the Arnoldi procedure [Krylov methods 12.c.ii.3.f]).

In order to proceed with Level 3 BLAS routines, a block version `Block_GS` is obtained by considering blocks of vectors instead of single vectors in `Modified_GS` and by replacing the normalizing step by an application of `Modified_GS` on the individual blocks. Jalby and Philippe [17] proved that it is necessary to apply the normalizing step twice to reach the numerical accuracy of `Modified_GS`. However, with  $q$  blocks of  $r$  vectors ( $n = qr$ ), the `Block2_GS` procedure only involves  $(1/q)(mn^2) + O(mn)$  additional operations which makes `Block2_GS` faster than `Modified_GS` as soon as  $q$  is not too small.

Parallel implementations of the different forms of Gram-Schmidt orthogonalization are similar to those of Gauss elimination schemes in which there are two nested loops of vector operations. To orthogonalize the columns of  $A$  on a cluster, the procedure `Modified_GS` can be expressed by:

```
ALGORITHM : Parallel Modified_GS
do j=1 : n,
 aj = aj / ||aj|| ;
 doall k = j+1 : n,
 ak = ak - (aj^T ak)aj ;
 end
end
```

By inserting a synchronizing mechanism to insure that normalizing a vector happens only after it is completely updated, it is even possible to transform the sequential outer loop into a `doacross` loop.

On distributed memory platforms, this algorithm may also be used, but it might be more efficient to use a block partitioning similar to that of ScaLAPACK for either `Modified_GS` or `Block2_GS`.

## Other Least Squares Problems

**Rank-deficient problems.** When  $A$  is rank  $q < n$ , there is an infinite set of solutions for (1): for any solution  $x$ , the

set of all solutions is expressed by  $\mathcal{S} = x + \mathcal{N}(A)$  where  $\mathcal{N}(A)$  is the  $(n - q)$ -dimensional null space of  $A$ . The Singular Value Decomposition (SVD) of  $A$  is given by

$$A = U \begin{pmatrix} \Sigma & 0_{q,n-q} \\ 0_{m-q,q} & 0_{m-q,n-q} \end{pmatrix} V^T, \quad (6)$$

where the diagonal entries of the matrix  $\Sigma = \text{diag}(\sigma_1, \dots, \sigma_q)$  are positive,  $U \in \mathbb{R}^{m \times m}$ ,  $V \in \mathbb{R}^{n \times n}$  are

orthogonal matrices. Let  $c = U^T b = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}$  where  $c_1 \in \mathbb{R}^q$ . Therefore,  $x \in \mathcal{S}$  if and only if there exists  $y \in \mathbb{R}^{n-q}$  such that  $x = V \begin{pmatrix} \Sigma^{-1} c_1 \\ y \end{pmatrix}$  and the corresponding minimum residual norm is equal to  $\|c_2\|$  which is null when  $m = q$ . In  $\mathcal{S}$ , the classically selected solution is the one with the smallest norm, i.e., when  $y = 0$ .

**Householder QR with column pivoting.** In order to avoid computing the singular-value decomposition which is quite expensive, the factorization (6) is often replaced by a similar factorization where  $\Sigma$  is not assumed to be diagonal. Such factorizations are called *complete orthogonal factorizations*. One way to compute such factorization relies on the procedure introduced by Businger and Golub [10]. It computes the QR factorization by Householder reductions with column pivoting. Factorization (2) is now replaced by

$$Q_q \cdots Q_2 Q_1 A P_1 P_2 \cdots P_q = \begin{pmatrix} T \\ 0 \end{pmatrix}, \quad (7)$$

where matrices  $P_1, \dots, P_q$  are permutations and  $T \in \mathbb{R}^{q \times n}$  is upper-trapezoidal. The permutations are determined so as to insure that the absolute values of the diagonal entries of  $T$  are nonincreasing. To obtain a complete orthogonal factorization of  $A$  the process performs first the factorization (7) and then computes an LQ factorization of  $T$  (equivalent to a QR factorization of  $T^T$ ). The process ends up with a factorization similar to (6) with  $\Sigma$  being a nonsingular lower-triangular matrix.

In LAPACK, Householder QR with column pivoting is implemented in the routine DGEQPF which only involves Level 1 and Level 2 BLAS routines. The new

routine DGEQP3 is proposed with Level 3 BLAS calculations. The rank-deficient linear least squares problems are solved by the routines DGELSX which uses DGEQPF or DGELSY which, in turn, uses DGEQP3.

**Parallel implementation.** In ScaLAPACK, Householder QR with column pivoting and complete orthogonal factorization are both implemented via the routines PDGEQPF and PDTZRZF, respectively.

Bischof introduced a method that avoids global pivoting to achieve a better parallel efficiency [2]. It consists of a local pivoting strategy limited to each block allocated to a single processor. This strategy is numerically acceptable as long as the condition number of a block remains below some threshold. In order to control it, the author uses an Incremental Condition Estimator (ICE) studied in [3]. Achieving adequate local pivoting strategies is a topic that is yet to be completely settled.

**Generalized least squares and total least squares problems.** In many applications, the right-hand-side  $b$  of problem (1) is known only with some uncertainty: its error is known to follow a statistical law given by its covariance matrix  $\Omega$ . In general,  $\Omega$  is positive definite and its Cholesky factorization  $\Omega = LL^T$  does exist. The Generalized Least Squares problem consists of computing  $x \in \mathbb{R}^n$  such that  $\|b - Ax\|_{\Omega^{-1}}$  is minimized. It can therefore be recast into the regular least squares problem: compute  $x \in \mathbb{R}^n$  such that  $\|L^{-1}(b - Ax)\|$  is minimized. This approach is acceptable when (1)  $\Omega$  is not ill-conditioned and (2) the factor  $L$  can be computed.

When the matrix  $A$  suffers from uncertainty as well, the problem becomes one of Total Least Squares. The basic problem is then given by

$$\text{minimize } \| [A; b] - [\hat{A}; \hat{b}] \|_F, \quad \text{subject to } \hat{b} \in R(\hat{A}),$$

where  $\|\cdot\|_F$  denotes the Frobenius norm and where  $R(\hat{A})$  is the range of  $\hat{A}$ . Any  $x \in \mathbb{R}^n$  for which  $\hat{A}x = \hat{b}$  is a Total Least Squares solution (TLS). By denoting, respectively,  $\sigma_n$  and  $\tilde{\sigma}_{n+1}$  the smallest singular values of  $A$  and  $[A; b]$ , assuming that  $\sigma_n > \tilde{\sigma}_{n+1}$ , the only TLS solution is  $\hat{x} = (A^T A - \tilde{\sigma}_{n+1}^2 I)^{-1} A^T b$ . Total Least Squares theory, problems, as well as solution schemes are given by Van Huffel and Vandewalle in [27].

Parallel algorithms for the Generalized or Total Least Squares problems are based on Linear Least

Squares, Cholesky factorization, and Singular Value Decomposition.

Kontoghiorghes considers the Seemingly Unrelated Regression Equations Models (SURE) models as Generalized Least Squares problems [18]. In subsequent work with his co-authors, he considers updating (adding equations to the model), down-dating (deleting equations) as well as parallel resolution of the models. Gang Lou and Shih-Ping Han introduced in [20] a general class of parallel methods for solving other Least Squares problems with constraints.

## Related Entries

- ▶ [BLAS \(Basic Linear Algebra Subprograms\)](#)
- ▶ [Dense Linear System Solvers](#)
- ▶ [LAPACK](#)
- ▶ [Linear Algebra, Numerical](#)
- ▶ [PLAPACK](#)
- ▶ [ScaLAPACK](#)

## Bibliographic Notes and Further Reading

Numerical methods for least squares problems: [6] Parallel Algorithms for dense computations: [13, 14] Total least squares : [27]

## Bibliography

1. Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D (1999) LAPACK users' guide, 3rd edn. SIAM, Philadelphia, Library available at <http://www.netlib.org/lapack/>
2. Bischof CH (1988) A parallel QR factorization algorithm using local pivoting. In: Proceedings of the ACM/IEEE Conference on Supercomputing, Orlando, pp 400–499
3. Bischof CH (1990) Incremental condition estimation. SIAM J Matrix Anal Appl 11:2:312–322
4. Bischof CH, Van Loan C (1987) The WY representation for products of householder matrices. SIAM J Sci Stat Comp 8:s2–s13
5. Bjorck A (1967) Solving linear least squares problems by Gram-Schmidt orthogonalization. BIT 7:1–21
6. Bjorck A (1996) Numerical methods for least squares problems. SIAM, Philadelphia
7. Blackford LS, Choi J, Cleary A, D'Azevedo E, Demmel J, Dhillon I, Dongarra J, Hammarling S, Henry G, Petitet A, Stanley K, Walker D, Whaley RC (1986) ScaLAPACK users' guide. SIAM, Philadelphia, Available on line at <http://www.netlib.org/lapack/lug/>

8. Blackford LS, Choi J, Cleary A, D'Azevedo E, Demmel J, Dhillon I, Dongarra J, Hammarling S, Henry G, Petitet A, Stanley K, Walker D, Whaley RC (1997) ScaLAPACK: a linear Algebra Library for Message-Passing Computers. Information Bridge: DOE Scientific and Technical Information, #468499. Available at <http://www.osti.gov/bridge/servlets/purl/468499-dBCNwX/webviewable/468499.pdf>
9. Blackford LS, Demmel J, Dongarra J, Duff I, Hammarling S, Henry G, Heroux M, Kaufman L, Lumsdaine A, Petitet A, Pozo R, Remington K, Whaley RC (2002) An updated set of basic linear algebra subprograms (BLAS). ACM Trans Math Soft 28–2:135–151
10. Businger PA, Golub GH (1965) Linear least squares solutions by householder transformations. Numer Math 7:269–276
11. Cosnard M, Muller J-M, Robert Y (1986) Parallel QR decomposition of a rectangular matrix. Numer Math 48:239–249
12. Cosnard M, Robert Y (1986) Complexity of parallel QR factorization. J ACM 33(4):712–723
13. Dongarra JJ, Duff IS, Sorensen DC, Van der Vorst H (1998) Numerical linear algebra for high performance computers. SIAM, Philadelphia
14. Gallivan KA, Plemmons RJ, Sameh AH (1990) Parallel algorithms for dense linear algebra computations. In: Gallivan KA, Heath MT, Ng E, Ortega JM, Peyton BW, Plemmons RJ, Romine CH, Sameh AH, Voigt RG (eds) Parallel algorithms computations. SIAM, Philadelpdia
15. Golub GH, Plemmons RJ, Sameh AH (1988) Parallel block schemes for large-scale least-squares computations. In: Wilkinson RB (ed) High-speed computing, scientific applications and algorithm design. University of Illinois Press, Illinois, pp 171–179
16. Golub GH, Van Loan CF (1996) Matrix computations, 3rd edn. John Hopkins University Press, Baltimore
17. Jalby W, Philippe B (1991) Stability analysis and improvement of the Block Gram-Schmidt algorithm. SIAM J Sci Stat Comput 12(5):1058–1073
18. Kontoghiorghes EJ (2000) Parallel algorithms for linear models: numerical methods and estimation problems. Advances in computational economics, vol 15, Springer, Boston, Dordrecht, London, Kluwer Academic Publishers
19. Lawson CL, Hanson RJ, Kincaid DR, Krogh FT (1979) Basic linear algebra subprograms for FORTRAN usage. ACM Trans Math Soft 5–3:308–323
20. Lou G, Han S-P (1988) A parallel projection method for solving generalized linear least-squares problems. Numer Math 53(3):255–264
21. Modi JJ, Clarke MRB (1984) An alternative Givens ordering. Numer Math 43:83–90
22. Sameh AH (1977) Numerical parallel algorithms – a survey. In: Kuck D, Lawrie D, Sameh A (eds) High speed computer and algorithm organization. Academic, New York, pp 207–228
23. Sameh AH (1985) Solving the linear leastsquares problem on a linear array of processors. In: Snyder L, Jamieson LH, Gannon DB, Siegel HJ (eds) Algorithmically specialized parallel computers. Academic, New York, pp 191–200
24. Sameh AH, Kuck D (1978) On stable parallel linear system solvers. J ACM 25(1):81–91
25. Schreiber R, Parlett BN (1987) Block reflectors: theory and computation. SIAM J Numer Anal 25:189–205
26. Sidje RB (1997) Alternatives for parallel Krylov subspace basis computation. Numer Linear Algebra Appl 4–4:305–331
27. Van Huffel S, Vandewalle J (1991) The total least squares problems: computational aspects and analysis. SIAM, Philadelphia

## Linear Regression

### ► Linear Least Squares and Orthogonal Factorization

## LINPACK Benchmark

JACK DONGARRA, PIOTR LUSZCZEK  
University of Tennessee, Knoxville, TN, USA

### Definition

The LINPACK benchmark is a computer benchmark that reports the performance for solving a system of linear equations with a general dense matrix of size 100, 1,000, and also of arbitrary size. The matrices, the calculations, and the solution must use 64-bit floating point arithmetic, and partial pivoting needs to be used for numerical stability. The allowed parallelism modes include automatic parallelization done by the compiler as well as manual parallelization that uses hardware-assisted shared memory or explicit message passing on a distributed memory machine.

### Discussion

The original LINPACK Benchmark is, in some sense, an accident. It was originally designed to assist users of the LINPACK package [1] by providing information on execution times required to solve a system of linear equations. The first “LINPACK Benchmark” report appeared as an appendix in the LINPACK Users’ Guide in 1979. The appendix comprised data for one commonly used path in the LINPACK software package. Results were provided for a matrix problem of size 100, on a collection of widely used computers (23 computers in all). This was done so users could estimate the time required to solve their matrix problem by extrapolation.

Over the years, additional performance data was added, more as a hobby than anything else, and today the collection includes over 1,300 different computer systems. In addition to the increasing number of computers, the scope of the benchmark has also expanded. The benchmark report describes the performance for solving a general dense matrix problem  $Ax = b$  at three levels of problem size and optimization opportunity: 100 by 100 problem (inner loop optimization), 1,000 by 1,000 problem (three loop optimization – the whole program), and a scalable parallel problem.

The LINPACK benchmark features two routines: DGEFA and DGESL (these are the double precision versions, usually 64-bit floating point arithmetic, SGEFA and SGESL are the single-precision counter parts, usually 32 bit floating point arithmetic); DGEFA performs the decomposition with partial pivoting and DGESL uses that decomposition to solve the given system of linear equations. Most of the execution time –  $O(n^3)$  floating-point operations – is spent in DGEFA. Once the matrix has been decomposed, DGESL is used to find the solution; this requires  $O(n^2)$  floating-point operations.

DGEFA and DGESL in turn call three BLAS routines: DAXPY, IDAMAX, and DSCAL. By far the major portion of time – over 90% at order 100 – is spent in DAXPY. DAXPY is used to multiply a scalar,  $ff$ , times a vector,  $x$ , and add the results to another vector,  $y$ . It is called approximately  $n^2/2$  times by DGEFA and  $2n$  times by DGESL with vectors of varying length. The statement  $y_i \leftarrow y_i + ff x_i$ , which forms an element of the DAXPY operation, is executed approximately  $n^3/3 + n^2$  times, which gives rise to roughly  $2/3n^3$  floating-point operations in the solution. Thus, the  $n = 100$  benchmark requires roughly 2/3 million floating-point operations.

The statement  $y_i \leftarrow y_i + ff x_i$ , besides the floating-point addition and floating-point multiplication, involves a few one-dimensional index operations and storage references. While the LINPACK routines DGEFA and DGESL involve two-dimensional array references, the BLAS refer to one-dimensional arrays. The LINPACK routines in general have been organized to access two-dimensional arrays by column. In DGEFA, the call to DAXPY passes an address into the two-dimensional array  $A$ , which is then treated as a one-dimensional reference within DAXPY. Since the indexing is down

a column of the two-dimensional array, the references to the one-dimensional array are sequential with unit stride. This is a performance enhancement over, say, addressing across the column of a two-dimensional array. Since Fortran dictates that two-dimensional arrays be stored by column in memory, accesses to consecutive elements of a column lead to simple index calculations. References to consecutive elements differ by one word instead of by the leading dimension of the two-dimensional array.

### Detailed Operation Counts

The results reflect only one problem area: solving dense systems of equations using the LINPACK programs in a Fortran environment. Since most of the time is spent in DAXPY, the benchmark is really measuring the performance of DAXPY. The average vector length for the algorithm used to compute LU decomposition with partial pivoting is  $2/3n$ . Thus in the benchmark with  $n = 100$ , the average vector length is 66.

In order to solve this matrix problem, it is necessary to perform almost 700,000 floating point operations. The time required to solve the problem is divided into this number to determine the megaflops rate.

The routines DGEFA calls IDAMAX, DSCAL and DAXPY. Routine IDAMAX, which computes the index of a vector with largest modulus, is called 99 times, with vector lengths running from 2 to 100. Each call to IDAMAX gives rise to  $n$  double precision absolute value computation and  $n - 1$  double precision comparisons. The total number of operations is 5,364 double precision absolute values and 4,950 double precision comparisons.

DSCAL is called 99 times, with vector lengths running from 1 to 99. Each call to DSCAL performs  $n$  double precision multiplies, for a total of 4,950 multiplications.

DAXPY does the bulk of the work. It is called  $n$  times, where  $n$  varies from 1 to 99. Each call to DAXPY gives rise to one double precision comparison with zero,  $n$  double precision additions, and  $n$  double precision multiplications. This leads to 4,950 comparisons against 0, 328,350 additions and 328,350 multiplications. In addition, DGEFA itself does 99 double precision comparisons against 0, and 99 double precision reciprocals. The total operation count for DGEFA is given in [Table 1](#).

**LINPACK Benchmark. Table 1** Double precision operations counts for LINPACK 100's DGEFA routine

| Operation type | Operation count |
|----------------|-----------------|
| Add            | 328,350         |
| Multiply       | 333,300         |
| Reciprocal     | 99              |
| Absolute value | 5,364           |
| $\leq$         | 4,950           |
| $\neq 0$       | 5,247           |

**LINPACK Benchmark. Table 2** Double precision operations counts for LINPACK 100's DGESL routine

| Operation type | Operation count |
|----------------|-----------------|
| Add            | 9,900           |
| Multiply       | 9,900           |
| Divide         | 100             |
| Negate         | 100             |
| $\neq 0$       | 199             |

Routine DGESL, which is used to solve a system of equations based on the factorization from DGEFA, does much more modest amount of floating point operations. In DGESL, DAXPY is called in two places, once with vector lengths running from 1 to 99 and once with vector lengths running from 0 to 99. This leads to a total of 9,900 double precision additions, the same number of double precision multiplications, and 199 compares against 0. DGESL does 100 divisions and 100 negations as well. The total operation count for DGESL is given in [Table 2](#).

This leads to a total operation count for the LINPACK benchmark given in [Table 3](#) or a grand total of 697,509 floating point operations. (The LINPACK uses approximately  $2/3n^3 + 2n^2$  operations, which for  $n = 100$  has the value of 686,667.)

It is instructive to look just at the contribution due to DAXPY. Of these floating point operations, the calls to DAXPY from DGEFA and DGESL account for a total of 338,160 additions, 338,160 multiplications, and 5,147 comparisons with zero. This gives a total of 681,467 operations, or over 97% of all the floating point operations that are executed. The total time is taken up with more than arithmetic operations. In particular, there is quite a lot of time spent loading and storing the

**LINPACK Benchmark. Table 3** Double precision operations counts for LINPACK 100 benchmark

| Operation type | Operation count |
|----------------|-----------------|
| Add            | 338,250         |
| Multiply       | 343,200         |
| Reciprocal     | 99              |
| Divide         | 100             |
| Negate         | 100             |
| Absolute value | 5,364           |
| $\leq$         | 4,950           |
| $\neq 0$       | 5,446           |
| Total          | 697,509         |

operands of the floating point operations. We can estimate the number of loads and stores by assuming that all operands must be loaded into registers, but also assuming that the compiler will do a reasonable job of promoting loop-invariant quantities out of loops, so that they need not be loaded each time within the loop. Then DAXPY accounts for 681,468 double precision loads and 338,160 double precision stores. IDAMAX accounts for 4,950 loads, DSCAL for 5,049 loads and 4,950 stores, DGEFA outside of the loads and stores in the BLAS does 9,990 loads and 9,694 stores, and DGESL for 492 loads and 193 stores. Thus, the total number of loads is 701,949 and stores is 352,997. Here again DAXPY dominates the statistics. The other overhead that must be accounted for is the load indexing, address arithmetic, and the overhead of argument passing and calls.

## Precision

In discussions of scientific computing, one normally assumes that floating-point computations will be carried out to full precision or 64-bit floating point arithmetic. Note that this is not an issue of single or double precision as some systems have 64-bit floating point arithmetic as single precision. It is a function of the arithmetic used.

## Related Entries

- [Benchmarks](#)
- [HPC Challenge Benchmark](#)
- [LINPACK Benchmark](#)
- [Livermore Loops](#)
- [TOP500](#)

## Bibliography

1. Dongarra JJ, Bunch J, Moler C, Stewart GW (1979) LINPACK user's guide, SIAM, Philadelphia

## Linux Clusters

► [Clusters](#)

## \*Lisp

GUY L. STEELE JR.  
Oracle Labs, Burlington, MA, USA

### Definition

\*Lisp (pronounced “star-lisp”) is a data-parallel dialect of Lisp developed around 1984 for Connection Machine supercomputers manufactured by Thinking Machines Corporation. It consists of Common Lisp (or, in its earliest implementations, Symbolics Zetalisp) augmented with a package of relatively low-level data-parallel operations. The language design draws a sharp distinction between data residing in the front-end Common Lisp processor and data residing in the massively parallel Connection Machine coprocessor. While pointers to any Lisp data could reside in the Connection Machine processors, parallel processing was most effective on arrays of numeric and character data.

### Discussion

Of the four programming languages (\*Lisp, C\*, CM Fortran, and CM-Lisp) provided by Thinking Machines Corporation for Connection Machine Systems, \*Lisp was the first to be implemented and the one most closely aligned to details of the Connection Machine architecture; essentially anything that could be done with the Connection Machine hardware could be expressed in \*Lisp, but the programmer was freed from much drudgery because expression evaluation made automatic use of a stack for parallel data structures and provided automatic conversions among data types and sizes as necessary.

To quote from the *Connection Machine Model CM-2 Technical Summary*, 1987:

- The parallel primitives of \*Lisp support a model of the Connection Machine in which each processor executes a subset of Common Lisp, with a single thread of control residing on the front-end computer. For most Common Lisp functions, \*Lisp provides a corresponding parallel function that can operate on all processors, or some selected subsets, simultaneously. In addition, the language provides Lisp-level operators for communicating between processors, both through pointers and in regular patterns. Sequential Common Lisp code, running on the front end, can be freely intermixed with the parallel code executed on the Connection Machine.

Most \*Lisp functionality corresponds directly to underlying Paris [Connection Machine PARallel Instruction Set] instructions.... As a result ... direct calls to Paris instructions and special purpose microcode blend in naturally with \*Lisp code [1, p. 46], [2, p. 223].

By 1989 the phrase “most Common Lisp functions” had been amended to “many Common Lisp functions” [3, p. 27].

A \*Lisp *pvar* (parallel variable) was, in effect, an array indexed by processor number, so that each (virtual) Connection Machine processor contained one array element. A unary parallel operation would perform the same scalar operation on each element simultaneously; a binary parallel operation would operate on corresponding elements of two *pvars*. While, in general, a *pvar* could hold pointers to any Common Lisp objects, not all operations on Common Lisp objects were supported in parallel form. In particular, classic list operations such as CAR and CDR and ASSOC had no parallel versions (“\*Lisp does not implement list *pvars*” [4, p. 155]). On the other hand, most atomic Common Lisp types were eventually supported, including integers and floating-point numbers of various fixed sizes, complex numbers, and characters; and many of the Common Lisp sequence functions were provided in parallel form for operating on many arrays simultaneously. “The eight basic *pvar* types are boolean, integer, floating-point, complex, character, structure, and front-end value” [5, p. 80]. The operation of indexing into a *pvar* to fetch or store a specified element was supported in both scalar and parallel forms; in this way either the front-end computer or the entire set of Connection Machine processors could freely fetch or store any element of a *pvar*.

\*Lisp was useful because it provided all the program-building tools of Common Lisp [6, 7],

including user-defined macros, and because it provided convenient access to *all* the facilities of the Connection Machine hardware, including (on the model CM-2) the multidimensional NEWS network, “backwards routing” in the hypercube network, the thousands of red LEDs that were a distinctive feature of the Connection Machine cabinetry, and operations on integers and floating-point numbers that could be of any specified length, such as 23-bit integers or floating-point numbers with 9-bit exponents and 43-bit significands (a flexibility of the model CM-1 that few users exploited in practice once the hardware floating-point accelerators came into use on the model CM-2).

By 1990, \*Lisp had grown to include over 400 functions and macros, listed in a *\*Lisp Dictionary* of over 1,100 pages [8]. Compared with other dialects and extensions of Lisp, \*Lisp had a distinct visual style because of its consistent use of either a leading “\*” or a trailing “!!” in names:

- ▶ The names of functions and macros that return pvars as their values end in !!. This suffix, pronounced *bang-bang*, is meant to look like two parallel lines.... All \*Lisp functions that perform parallel computation and do not end in !! begin with \* (pronounced *star*); hence, the name \*Lisp [9, p. 5].

A later document further explains:

- ▶ The characters “!!” are meant to resemble the mathematical symbol ||, which means *parallel*. [5, p. 82]

(It should be noted that the lexical syntax of Common Lisp [6, 7] would have made it awkward to use two vertical-bar symbols “||” rather than two exclamation points.)

**Figure 1** shows an early (1987) example of a \*Lisp program that identifies prime numbers less than 100,000 by the method of the Sieve of Eratosthenes, taken (with minor corrections) from [1]. A set of virtual processors of size 100,000 is assumed to have been established before the function `find-primes` is called. The special form `*all` enables all virtual processors for activity. The special form `*let` is analogous to Common Lisp `let` in that it binds local variables, but `*let` binds local *parallel* variables, allocating them on a stack within the Connection Machine (actually, one stack per virtual processor). In this case, the local parallel variable `prime` is initially `nil` (false) within each processor, and the local parallel variable `candidate` is initially `t` (true) within each processor. The conditional form `*when` is analogous to Common Lisp `when`, but operates by (a) evaluating a parallel test form; (b) temporarily disabling for execution any processors that computed `nil` for the test form in step (a); (c) executing the form(s) in the body of the `*when` construct; and finally (d) re-enabling any processors disabled in step (b). The function `self-address!!` returns the virtual processor address, which is a different integer in each virtual processor (starting from zero); the function `!!` broadcasts a scalar value from the front-end processor to all Connection Machine processors, so that `(!! 2)` is a parallel quantity with value 2 in every virtual processor; and function `<!!` compares two parallel quantities, producing a parallel Boolean (`t` or `nil`) result. The functions `*or` and `*min` are reduction operations, producing the logical OR or the arithmetic minimum of one value taken from every *active* virtual processor; thus `(*or candidate)` will be `nil` only if every virtual

```
(*defun find-primes ()
 (*all
 (*let ((prime (!! nil) (candidate (!! t))))
 (*when (<!! (self-address!!) (!! 2))
 (*set candidate (!! nil)))
 (do () ((*or candidate))
 (*when candidate
 (let ((next-prime (*min (self-address!!))))
 (setf (pref prime next-prime) t)
 (*when (zerop!! (mod!! (self-address!!) (!! next-prime)))
 (*set candidate (!! nil))))))
 prime)))
 prime)))
```

\*Lisp. Fig. 1 Example \*Lisp program for identifying prime numbers

processor has the value nil for candidate, and (\*min (self-address!!)), because it occurs within (\*when candidate ...), will return the smallest integer for which candidate is true. The function pref is analogous to Common Lisp aref; it may be used by the front end to access a single element of a parallel value by supplying an integer index. (The parallel equivalent, not used in this example, is pref !!, which allows every Connection Machine processor to fetch a value by indexing into a parallel value; this is one way of performing interprocessor communication in \*Lisp.) The functions zerop !! and mod are parallel arithmetic operations that are analogous to Common Lisp zerop and mod.

## Related Entries

- C\*
- Connection Machine
- Connection Machine Fortran
- Connection Machine Lisp

## Bibliography

1. Thinking Machines Corporation. Connection Machine model CM-2 technical summary, technical report HA87-4. Cambridge, MA, April 1987
2. Michael Hord R (1990) Parallel supercomputing in SIMD architectures. CRC Press, Boca Raton
3. Thinking Machines Corporation. Connection Machine technical summary, version 5.1. Cambridge, MA, May 1989
4. Thinking Machines Corporation. Supplement to the \*Lisp reference manual, version 5.0. Cambridge, MA, September 1988
5. Thinking Machines Corporation. Connection Machine CM-5 technical summary, 3rd edn. Cambridge, MA, November 1993
6. Steele GL Jr, Fahlman SE, Gabriel RP, Moon DA, Weinreb (1984) Common Lisp: The language. Digital Press, Burlington
7. Steele GL Jr., Fahlman SE, Gabriel RP, Moon DA, Weinreb DL, Bobrow DG, DeMichiel LG, Keene SE, Kiczales G, Perdue C, Pitman KM, Waters RC, White JL (1990) Common Lisp: The language, 2nd edn. Digital Press, Bedford
8. Thinking Machines Corporation. \*Lisp dictionary, version 5.2. Cambridge, MA, February 1990
9. Thinking Machines Corporation. \*Lisp reference manual, version 5.0. Cambridge, MA, September 1988

## Lisp, Connection Machine

- Connection Machine Lisp

## Little's Law

JOHN L. GUSTAFSON

Intel Labs, Intel Corporation, Santa Clara, CA, USA

## Synonyms

Little's lemma; Little's principle; Little's result; Little's theorem

## Definition

Little's Law says that in the long-term, steady state of a production system, the average number of items  $L$  in the system is the product of the average arrival rate  $\lambda$  and the average time  $W$  that an item spends in the system, that is,

$$L = \lambda W.$$

## Discussion

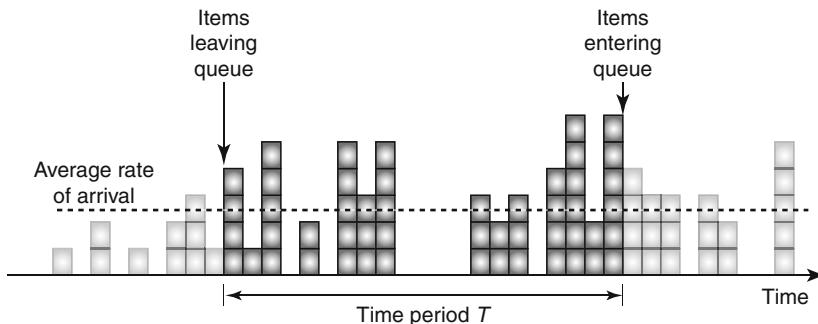
At first glance, Little's Law looks like common sense. If items arrive faster than the system can process them, the system will overflow. Perhaps the earliest mention of the relation is by A. Cobham in 1954, and he states it as a fact without proof [2]. However, the law is insightful in two ways. First, it does not depend on the probability distributions of any of the variables. Second, since arrival rates are generally less than maximum processing rates, it says what the capacity of the system must be to handle the queue in a system design.

Figure 1 shows a visualization of a queue in steady state that explains Little's Law:

Proofs of Little's Law use the Law of Large Numbers, which is implicit in the phrase “steady state” of the system. The number of items in any time period  $T$  is the integral of the arrival rate over the period, so if there is a steady state average rate  $\lambda$ , then  $L = \lambda W$  is the area of the rectangle under the dotted line for time period  $W$ . A more formal proof appears in [3] or [4].

Notice that the units in the expression are like that of a speed equation: work equals rate times time. Here, “work” is items that complete processing, “rate” is the arrival rate, and “time” is the time spent in the system.

The result is from queuing theory, and the meaning of  $L$ ,  $\lambda$ , and  $W$  varies by application. In manufacturing,  $L$  might mean inventory,  $\lambda$  the rate of arrival of materials, and  $W$  the amount of time the materials spend



**Little's Law. Fig. 1** Visualization of Little's law

in inventory. In retail applications,  $L$  might mean the average number of customers in a store,  $\lambda$  the rate they arrive at the store, and  $W$  the average time a customer spends in the store.

### Example

Suppose a fast food restaurant has an average business of one customer every two minutes, and the average customer takes 40 minutes to eat. For purposes of planning seating to accommodate the customers, Little's Law says how many seats in the restaurant will be in use, on average. Here,  $\lambda = 0.5$  customers per minute, and  $W = 40$  minutes; hence,  $L = 0.5 \times 40 = 20$  occupied seats.

If the business doubles to  $\lambda = 1.0$  customers per minute and the restaurant does not have enough seating area, then Little's Law says there are two solutions: double the seating area, or halve the average amount of time customers spend eating to 20 minutes, say, by making the seats less comfortable.

### Compound Queues

It is often useful to apply Little's Law in compound ways. For example, a store has a rate at which customers arrive and are in the process of browsing, and after selecting an item, they enter the checkout queue within the store to pay for the item. Thus, the law predicts the length of the checkout line and how crowded the store will be. Alternatively, it can predict how many checkout tellers suffice to prevent the checkout lines from averaging more than a certain goal in length.

### Example

Suppose the same fast food business assesses the trade-off between the cost of personnel and the effect waiting in line has on customer satisfaction. It then sets a goal

that the average amount of time a customer wishing to order food should have to wait in line is 6 minutes. Suppose also that there are two queues for ordering. How long will the line be to order food, on the average?

Since the two queues must accommodate the total arriving customer rate of  $\lambda = 1.0$  customers per minute, the rate for each food-ordering line is 0.5 customers per minute. Each line will then have an average length of

$$L = \lambda W = 0.5 \times 6 = 3 \text{ customers.}$$

### Implications for Computer Design

Little's Law is of obvious value for computer design, especially for *decoupled architectures*. Computer architect Burton Smith first noted the following form of Little's Law in 1995:

$$\text{Concurrency} = \text{Latency} \times \text{Bandwidth}.$$

Bandwidth is a rate of arrival of data into a processing queue. Latency is the time spent in the queue, and concurrency (or degree of parallelism) is the number of simultaneous data items to process. Some refer to this as *Little's Law of High Performance Computing* [1].

Processing elements require holding areas for data (buffers), and Little's Law provides guidance for how to match the processing rate, storage size of a buffer, and time a datum spends in the buffer. Just as more checkout tellers can reduce the average size of the checkout queue, parallel computing elements can reduce the amount of buffer space and the time spent in the buffer.

### Example

Suppose the processing units in a parallel computer can collectively process 1 trillion ( $10^{12}$ ) items of data in main memory per second, and the latency between the

main memory and the processors is 50 ns. Then Little's Law says that the number of parallel elements for processing memory must be

$$\text{Concurrency} = 10^{12} \times (50 \times 10^{-9}) = 50,000 \text{ processors.}$$

The parallel elements could be in the form of pipeline stages, data parallel units, or any other way that permits concurrent activity at the level of 50,000 items of data processed at once. Thus, if the system consists of 10,000 servers organized as a networked cluster, then each server must have an internal parallelism of a factor of five to accommodate the demands of the queue.

Even within a single processor, Little's Law is quite useful for answering architectural questions. For example, if the latency to the main memory is 60 ns for one core of a multicore processor, and the core operates at 3.0 GHz, then the memory system must be able to accommodate

$$(3.0 \times 10^9) \times (60 \times 10^{-9}) = 180 \text{ outstanding memory references,}$$

if each core is to make full use of its capability (that is, if the average arrival rate is the same as the maximum possible arrival rate).

Another important computer application of Little's Law is in assuring the fairness of benchmarks, especially those related to transaction processing. When measuring a system, if the benchmarking engineer asks it to process so many transactions (arrival rate  $\lambda$ ) that the storage (number of items  $L$  in the system) is larger than the main memory and thus spills into mass storage, the performance will drop catastrophically.

## Perspective

Operations Research (OR) originated in the 1950s, and courses on the subject rely heavily on queuing theory. Little was teaching formal queuing theory at Case Western Reserve University and noticed that the  $L = \lambda W$  behavior was independent of the probability distribution of the arrival times. At the urging of one of his students (Sid Hess), Little published the proof of the generality of the law. Little's 1961 theorem was one of the simplest yet most useful results from that early era. While formal queuing theory requires an understanding of concepts like Poisson distributions and a somewhat advanced level of statistics and mathematics, the simple formula  $L = \lambda W$  is one that requires no advanced

training to understand and apply. There was no thought in those early years of OR about answering questions of computer design. The traditional textbook examples are for inventory, manufacturing, order entry, and so on. Computer designs of that era tended to be tightly coupled, and so deterministic that the only statistical queuing behavior that would enter into their use would be that of users submitting jobs to the batch queue and waiting for computed results.

Burton Smith (Tera Computer) and David Bailey (NASA Ames Research) deserve credit for showing, in the late 1990s, the applicability of Little's Law to many issues of computing design. It seems obvious in retrospect that a law originally intended for use in the arena of business operations with human time scales could apply to computing design. Here the use is a subtle variation on the "average" behavior, in that the law assesses the *optimum* behavior. If the processing speed is to be some idealized maximum, and the latency is as small as possible, then the law predicts a lower bound on the capacity or parallelism required for near-peak performance.

Little's Law for High Performance Computing provides perhaps the simplest way to explain the necessity for the parallel computing approach. Latency tends to be difficult to reduce because of the laws of physics; concurrency is the product of latency and bandwidth (processing rate), so increasing bandwidth forces the need for more concurrency.

## Related Entries

- ▶ [Bandwidth-Latency Models \(BSP, LogP\)](#)
- ▶ [Dependences](#)
- ▶ [Dependence Abstractions](#)
- ▶ [Network Obliviousness](#)
- ▶ [NUMA Caches](#)
- ▶ [Pipelining](#)

## Bibliographic Notes and Further Reading

Little's original 1961 paper is of historic interest, as is a less general precedent text (1958) by OR pioneer Philip Morse: *Queues, Inventories and Maintenance* [6]. A very readable account with many examples drawn from everyday situations is in [5]. The latter work also gives perspective on the impact of the law over four

decades, and provides examples from managing email, traffic created by toll booths, and the housing market.

Bailey's brief 1997 paper [1] promulgated the use of Little's Law in High Performance Computing, and credits Burton Smith for the first restatement of the law in computer architecture terms.

## Bibliography

1. Bailey DH (1997) Little's law and high performance computing. RNR Technical Report, NAS applications and tools group, NASA Ames Research Center. At <http://crd.lbl.gov/~dhbailey/dhbpapers/little.pdf>
2. Cobham A (1954) Priority assignment in waiting line problems Oper Res 2(1):70–76
3. Jewell WS (1967) A simple proof of  $L = \lambda W$ . Oper Res 15(6): 1109–1116
4. Little JDC (1961) A proof of the queueing formula  $L = \lambda W$ . Oper Res 9:383–387
5. Little JDC, Graves SC (2008) Little's law. In: Chhajed D, Lowe TJ (eds) Building intuition: insights from basic operations management models and principles. MIT Press, Cambridge. Available at <http://web.mit.edu/sgraves/www/papers/Little's%20Law-Published.pdf>
6. Morse PM (1958) Queues, inventories and maintenance: the analysis of operational systems. Original edition by Wiley, New York. ISBN 0-86-3914-. Reprinted by Dover Phoenix Editions, 2004

## Livermore Loops

JACK DONGARRA, PIOTR LUSZCZEK  
University of Tennessee, Knoxville, TN, USA

## Definition

Livermore Loops are a set of 24 Fortran DO-loops (The Livermore Fortran Kernels, LFK) extracted from operational codes used at the Lawrence Livermore National Laboratory [1, 2]. They have been used since the early 1970s to assess the arithmetic performance of computers and their compilers. They are a mixture of vectorizable and nonvectorizable loops and test rather fully the computational capabilities of the hardware as well as the skill of the software in compiling and vectorization of efficient code. The main value of the benchmark is the range of performance that it demonstrates, and in this respect it complements the limited range of loops tested in the LINPACK benchmark.

## Discussion

As a benchmark, the Livermore Loops provide the individual performance of each loop, together with various averages (arithmetic, geometric, harmonic) and the quartiles of the distribution. However, it is difficult to give a clear meaning to these averages, and the value of the benchmark is more in the distribution itself. In particular, the maximum and minimum give the range of likely performance in full applications. The ratio of maximum to minimum performance has been called the instability or the specialty [3], and is a measure of how difficult it is to obtain good performance from the computer, and therefore how specialized it is. The minimum or worst performance obtained on these loops is of special value, because there is much truth in the saying that “the best computer to choose is that with the best worst-performance.”

The LFK set began in the early 1970s with only 12 kernels and was timed on the mainframes of the era with CPU-times of microsecond accuracy. By the early 1980s the number of kernels in the set grew to 24 and, during the decade, statistical features of the results reporting code were enhanced and adaptations were made to accommodate crude UNIX timers of only millisecond resolution. A port of LFK to C happened in the middle

## Little's Lemma

► Little's Law

## Little's Principle

► Little's Law

## Little's Result

► Little's Law

## Little's Theorem

► Little's Law

of 1980s. By the beginning of the 1990s, the compilers were able to perform code-hoisting which necessitated changes to the code like introducing functions to embed the critical kernels inside it and hide it from the optimization. The 1990s brought new hardware from HP, SGI, and SUN which exhibited vast performance improvements but still suffered with low timer resolution which required increase in the repetition count to assure 1% accuracy of the reported timings.

The origins or functions of each of the LFK kernels are the following:

1. Fragment of hydrodynamics code
2. ICCG (Incomplete Cholesky Conjugate Gradient) excerpt
3. Inner product calculation
4. Banded linear equations
5. Tri-diagonal elimination below the diagonal
6. General linear recurrence equations
7. Fragment of equation of state code
8. ADI (Alternating Direction Implicit) integration scheme for differential equations
9. Integrate predictor kernel
10. Difference predictor kernel
11. Summation with storage of partial sums
12. First difference scheme
13. 2D PIC (Particle In Cell) code
14. 1D PIC
15. A challenging (for the compiler) ordering of scalar operations in FORTRAN
16. Monte Carlo search loop
17. Implicit conditional computation
18. 2D explicit hydrodynamics fragment
19. General linear recurrence equations
20. Discrete ordinates transport: recurrence
21. Matrix-matrix multiplication
22. Planckian probability distribution
23. 2D implicit hydrodynamics fragment
24. A loop that finds location of first minimum in an array

In the LFK collection, scalar tests are numbered 5, 11, 17, 19, and 20 [1], page 14]. In the late 1980s, the scalar features of the contemporary machines were important for supercomputing at Los Alamos (LANL) – as much as 30% of the code executed at the laboratory might have been scalar [4]. The LFK scalar codes are small, so their indicated performance is generally higher than

that measured with larger benchmarks. Nonetheless, a good correlation exists among the LFK scalar tests and others [5].

Vectorization at the compiler level was turned on and off in the LFK kernels through special comments that were recognized as compiler directives. Similarly, code directives were used to render selected kernels scalar and force their execution onto the scalar units of the processor. The scalar execution was used as the definition of necessary computation and the appropriated number of the performed floating point operations. The actual number of floating point operations for vectorized execution could have been larger. For the purposes of the calculation of the performance results, each type of floating operation was weighted. The weight represented the execution time relative to floating point addition. For example, the weight of floating point division was 4 which meant that it takes four times as many cycles to complete a division than an addition of two floating point numbers.

The authors of LFK suggested to use the results by weighing their performance in proportion to the actual usage of that category of computation in the total workload. For example, for a hydrodynamics code, kernels 1, 18, and 23 would be the most relevant and consequently they should be given the greatest weight.

Another important aspect of performance benchmarking stressed by the LFK report was the treatment of the results' data as statistical quantity. A single number quoted as the performance rate was considered insufficient. Statistical sampling was encouraged and each reported number was to be accompanied by minimum and maximum values obtained as well as the equi-weighted harmonic, geometric, and arithmetic means. In addition, the first, second, third, and fourth quartiles were used to perform sensitivity analysis of harmonic mean rate with respect to various weight distributions.

At the time of release of the LFK set, the available hardware included machines from Cray, DEC, IBM, and NEC. The clock frequencies were of order 100 MHz and the achieved performance varied between single digit Mflop/s to man hundreds of Mflop/s depending on the kernel and the type of processor (vector or RISC).

## Related Entries

- [Benchmarks](#)
- [HPC Challenge Benchmark](#)

- [LINPACK Benchmark](#)
- [TOP500](#)

## Bibliography

1. McMahon FH (1986) The Livermore FORTRAN kernels: a computer test of the numerical performance range. Lawrence Livermore Laboratory technical report LLNL UCRL-53724. Lawrence Livermore National Laboratory, Livermore
2. McMahon FH (1988) The Livermore Fortran kernels test of the numerical performance range. In: Martin JL (ed) Performance evaluation of supercomputers. Elsevier Science, North-Holland, Amsterdam, pp 143–186
3. Hockney RW (1991) Performance parameters and benchmarking of supercomputers. Parallel Comput 17:1111–1130
4. Barley D, Brooks E, Dongarra J, Hayes A, Heath M, Lyon G (1988) Benchmarks to supplant export “FPDR” calculations. Technical report RNR-88-007, National Bureau of Standards, Gaithersburg
5. Lubeck OM (1988) Supercomputer performance: The theory, practice and results. Los Alamos National Laboratory Report LA-11204-MS, 58 pp

## Load Balancing

- [Load Balancing, Distributed Memory](#)
- [METIS and ParMETIS](#)

## Load Balancing, Distributed Memory

AARON BECKER, GENGBIN ZHENG, LAXMIKANT V. KALÉ  
University of Illinois at Urbana-Champaign, Urbana, IL, USA

### Definition

Load balancing in distributed memory systems is the process of redistributing work between hardware resources to improve performance, typically by moving work from overloaded resources to underloaded resources.

### Discussion

In a parallel application, when work is distributed unevenly so that underloaded processors are forced to wait for overloaded processors, the application is suffering from *load imbalance*. Load imbalance is one of

the key impediments in achieving high performance on large parallel machines, especially when solving highly dynamic and irregular problems. *Load balancing* is a technique that performs the task of distributing computation and communication load across the hardware resources of a parallel machine so that no single processor is overloaded. It can reduce processor idle time across the machine while also reducing communication costs by colocating related work on the same processor.

Balancing an application’s load involves making decisions about where to place newly created computational tasks on processors, or where to migrate existing work among processors. Orthogonally, some applications only require static load balancing: They have consistent behaviors over their lifetimes, and once they are balanced they remain balanced. Other applications exhibit dynamic changes in behavior and require periodic rebalancing. Typically it is far too expensive to compute an optimal distribution of work in any realistic application, but a wide variety of heuristic algorithms have been developed that are very effective in practice across a wide variety of application domains.

The load associated with an application can be conceptualized as a graph, where the nodes are discrete units of work and an edge exists between two nodes if there is a communication between them. To solve the load balancing problem one must split this graph into parts, with each part representing the work to be associated with a particular hardware resource. This partitioning process is at the heart of load balancing.

Given the heuristic nature of load balancing, which makes perfect load balance an unrealistic goal, it is important to remember that the goal of load balancing is not so much to equalize the amount of work on each processor as it is to minimize the load of the most heavily loaded processor in the system. The most overloaded processor is the bottleneck that determines time to completion. A heuristic that leaves some processors idle is preferable to one that reduces the variance in load, as long it reduces the load on the most loaded processor. Load balancing heuristics can also take factors beyond time to completion into account, for example by trying to minimize power usage over an application’s lifetime.

Modern HPC systems include clusters of multi-core nodes. The general load balancing strategies described

in this entry are still applicable at the level of nodes (i.e., work is assigned to nodes). Finer grained load balancing within a node (work assigned to cores) can be performed independently. In the simplest case, it can be done by applying the same load balancing strategies that are used for node-level load balancing, except that the cost of communication among cores within a node is much smaller than the internode communication, and the number of cores involved within a node is relatively small.

### Periodic Load Balancing

In a periodic load balancing scheme, computational tasks are persistent; load is balanced only as needed, and the balancing consists of migrating existing tasks and their associated data. With periodic load balancing, expensive load balancing decision making and data migration are not continuous processes. They occur only distinct point in the application when a decision to do balancing has been made. Periodic load balancing schemes are suitable for iterative scientific applications such as molecular dynamics, adaptive finite element simulation, and climate simulation, where the computation typically consists of a large number of time steps, iterations (as in iterative linear system solvers), or a combination of both. A computational task in these applications is executed for a long period of time, and tends to be persistent. During execution, at periodic intervals, partially executed tasks may be moved to different processors to achieve global load balance.

### The Load Balancing Process

In an application which rebalances its load periodically, there are four distinct steps in the load balancing process. The first step is load estimation, which tries to gauge what the load on each processor will be in the near future if no rebalancing is done. This may involve the use of a model that predicts future performance based on current conditions, or it may be based directly on past measured load, with the assumption that the near future will closely resemble the near past. The second step is making a decision of *when* to execute load balancing. This is a trade-off between the cost of load balancing itself (i.e., the cost of determining a new mapping of application tasks to hardware and the cost of migration) and the savings one can expect from a

better load distribution. The nature of the application being balanced and the cost of the load balancing method to be used factor heavily into this decision. The third step is determining how the load will be rebalanced. There are many algorithms devoted to computing a good mapping of work onto processors, each with its own strengths and weaknesses. This section describes some of the most widely used methods. The general problem of creating a mapping of work onto processors which minimizes time to completion is NP-hard, so all strategies we discuss are heuristic. The final step in the load balancing process is migration, the process by which work is actually moved. This may simply be a matter of relocating application data, but it can also encompass thread and process migration.

### Initial Balancing

For some classes of application, there is very little change in load balance over time once the application reaches a steady state. Thus, once good load balance is achieved, no further balancing need be done. However, load balancing is often still an important part of such applications. For example, consider the case of molecular dynamics, where one must distribute simulated particles onto processors. These applications do not experience significant dynamic load imbalance, but determining a good initial mapping may be difficult because of the difficulty of estimating the load of multiple computational tasks accurately *a priori*.

In cases like this, one can simply start the application with an unoptimized distribution of work, run the application long enough to accurately measure the load, rebalance based on those measurements, and then continue without the need for any further balancing.

### Classifying Load Balancers

There are many families of load balancing approaches that can be classified according to how they answer the fundamental questions of load balancing: How do we estimate the load, at what granularity should one balance the load, and where should the load balancing decisions be made.

Load estimation underlies all load balancing algorithms. Load balancing is fundamentally a forward-looking task which aims to improve the future performance of the application. To do this effectively, the load balancer must have some model of what the

future performance of the application will be in different scenarios. There are two common approaches to estimating future load. The first is to measure the current load associated with each piece of work in the application and to assume that this load will remain the same after load balancing. This assumption is based on the *principle of persistence*, which posits that, for certain classes of scientific and engineering applications, computational loads and communication patterns tend to persist over time, even in dynamically evolving computations. This approach has several advantages: It can be applied to any application, it accurately and automatically accounts for the particular characteristics of the machine on which the application is running, and it removes some of the burden of load estimation from the application developer.

The alternative is to build some model of application performance which will provide a performance estimate for any given distribution of work. These models can be sophisticated enough to take into account dynamic changes in application performance that a measurement-based scheme will not account for, but they must be created specifically to match the actual characteristics of the application they are used for. If the model does not match reality then poor load balancing may result.

Load balancing may take place at any of several levels of *granularity*. The greatest flexibility in mapping work onto hardware resources can be achieved by exposing the smallest possible meaningful units of data to the load balancer. For example, these may be nodes in a finite element application or individual particles in a molecular simulation. Alternatively, one may group this data into larger chunks and only expose those chunks to the load balancing algorithm. This makes the load balancing problem smaller while guaranteeing respect for locality within the chunks. For example, in a molecular simulation the load balancer might only try to balance contiguous regions which may contain a large number of particles without being directly aware of the particles. It is also possible to balance load by migrating entire processes, avoiding the need for application developers to write code dedicated to migrating their data during the load balancing process.

Load balancers may be further categorized according to *where* decisions are made: locally, globally, or

according to some hierarchical scheme. Global load balancers collect global information about load across the entire machine and can use this information to make decisions that take the entire state of the application into account. The advantage of these schemes is that load balancing decisions can take a global view of the application, and all decisions about which objects should migrate to which processors can be made without further coordination during the load balancing process. However, these schemes inherently lack scalability. As problem sizes increase, the object communication graph may not even fit in memory on a single node, making global algorithms infeasible. Even if the load balancing process is not constrained by memory, the time required to compute an assignment for very large problems may preclude the use of a global strategy. However, with coarse granularity, it is possible to use global strategies up to several thousand processors, especially if load balancing is relatively infrequent.

Parallelized versions of global load balancing schemes are also possible. These use the same principles for partitioning as a serial global scheme, but to improve scalability the task graph is spread across many processors. This introduces some overhead in the partitioning process, but allows much larger problems to be solved than a purely serial scheme. One example of a parallel global solver is ParMETIS, the parallel version of the METIS graph partitioner.

Distributed load balancing schemes lie at the opposite end of the scale from global schemes. Distributed schemes use purely local information in their decision-making process, typically looking to offload work from overloaded processors to their less-loaded immediate neighbors in some virtualized topology. This leads to a sort of diffusion of work through the system as objects gradually move away from overloaded regions into underloaded regions. These schemes are very cheap computationally, and do not require global synchronization. However, they have the disadvantage of redistributing work slowly compared to global schemes, often requiring many load balancing phases before achieving system-wide load balance, and they typically perform more migrations before load balance is achieved.

Hybrid load balancing schemes represent a compromise between global and distributed schemes. In a hybrid load balancer, the problem domain is split

into a hierarchy of sub-domains. At the bottom of the hierarchy, global load balancing algorithms are applied to the small sub-domains, redistributing load within each sub-domain. At higher levels of the hierarchy, the load balancing strategies operate on the sub-domains as indivisible units, redistributing these larger units of work across the machine. This keeps the size of the partitioning problem to be solved low without giving up some global decision-making capabilities.

## Algorithms

An application's load balancing needs can vary widely depending on its computational characteristics. Accordingly, a wide variety of load balancing algorithms have been developed. Some do careful analysis of an application's communication patterns in an effort to reduce internode communication; others only try to minimize the maximum load on a node. Some try to do a reasonably good job as quickly as possible; others take longer to provide a higher quality solution. Some are even tailored to meet the needs of particular application domains. Despite the wide variety of algorithms available, there are a few in wide use that demonstrate the range of available techniques.

## Parallel Prefix

In some cases, the pattern of communication between objects is unimportant, and we care only about keeping an equal amount of work on each processor. In this scenario, load may be effectively distributed using a simple parallel prefix strategy (also known as scan or prefix sum).

The input to the parallel prefix algorithm is simply a local array of work unit weights on each processor reflecting the amount of work that each unit represents. The classic parallel prefix algorithm computes the sum of the first  $i$  units for each of the  $n$  unit in the list. Once the prefix sum is computed, all work can be equally distributed across processors by sending each unit to the processor number given by its prefix sum value divided by the total work divided by the number of processors.

The primary advantage of the parallel prefix scheme is its fast, inexpensive nature. The computation can be shared across  $p$  processors using only  $\log p$  communications, and efficient prefix algorithms are widely available in the form of library functions like MPI\_Scan. An assignment of work units to processors based on parallel

prefix can be computed virtually instantaneously even for extremely large processor counts, regardless of variations in work unit weights.

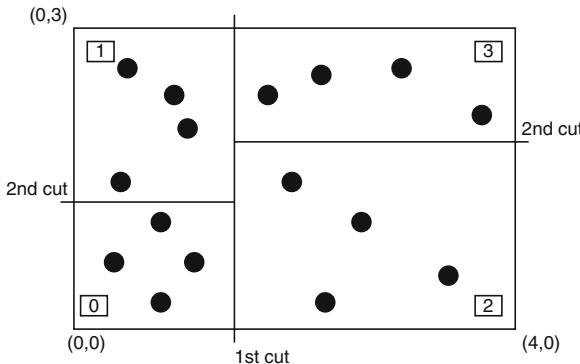
The corresponding disadvantage of using the parallel prefix algorithm for load balancing is its simplicity. It does not account for the structure of communication between processors and will not attempt to minimize the communication volume of the resulting partition. It also does not attempt to minimize the amount of migration needed.

## Recursive Bisection

Recursive bisection is a divide-and-conquer technique that reduces the partitioning problem to a series of bisection operations. The strategy of recursive bisection is to split the object graph in two approximately equal parts while minimizing communication between the parts, then proceeding to subdivide each half recursively until the required number of partitions is achieved. This recursive process introduces parallelism into the partitioning process itself. After the top-level split is completed, the two child splits are independent and can be computed in parallel, and after  $n$  levels of bisection there are  $2^n$  independent splitting operations to perform. In addition, geometric bisection operations can themselves be parallelized by constructing a histogram of the nodes to be split.

There are many variations of the recursive bisection algorithm based on the algorithm used to split the graph. Orthogonal recursive bisection (ORB), shown in Fig. 1, is a geometric approach in which each object is associated with coordinates in some spatial domain. The bisection process splits the domain in two using an axis-aligned cutting plane. This can be a fast way of creating a good bisection if most communication between objects is local. However, this method does not guarantee that it will produce geometrically connected sub-domains, and it may also produce sub-domains with high aspect ratios. Further variations exist which allow cutting planes that are not axis-aligned.

An alternate approach is to use spectral methods to do the bisection. This involves constructing a sparse matrix from the communication graph, finding an eigenvector of that matrix, and using it as a separator field to do the bisection. This method can produce superior results compared with ORB, but at a substantially higher computational cost.



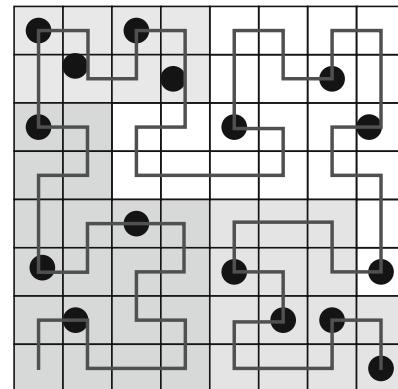
**Load Balancing, Distributed Memory.** Fig. 1 Orthogonal recursive bisection (ORB) partitions by finding a cutting plane that splits the work into two approximately equal pieces and recursing until the resulting pieces are small enough [8]

For many problems, recursive bisection is a fast way of computing good partitions, with the added benefit that the recursive procedure naturally exposes parallelism in the partitioning process itself. The advantages and disadvantages of recursive bisection depend greatly on the nature of the bisection algorithm used, which can greatly affect partition quality and computational cost.

### Space-Filling Curve

A space-filling curve is a mathematical function that maps a line onto the entire unit square (in two dimensions, or the entire unit  $N$ -cube in  $N$  dimensions). There are many such curves, and they are typically defined as the limit of sequence of simple curves, so that closer and closer approximations to the space-filling curve can be constructed using an iterative process. The value of space-filling curves for load balancing is that they can be used to convert  $n$ -dimensional spatial data into one-dimensional data. Once the higher-dimensional data has been linearized, it can be easily partitioned by splitting the line into pieces with equal numbers of elements.

Consider the case where each object in the application has a two-dimensional coordinate associated with it. Starting with a coarse approximation to the space-filling curve, a recursive refinement process can be used to create closer and closer approximations until each object is associated with its own segment of the approximated curve. This creates a linear ordering of the objects, as depicted in Fig. 2, which can then be split into even partitions.



**Load Balancing, Distributed Memory.** Fig. 2 To partition data using a space-filling curve, a very coarse approximation of the curve is replaced with finer and finer approximations until each unit of work is in its own segment. The curve is then split into pieces, with each piece containing an equal number of work units. This figure shows a Hilbert curve [8]

A good choice of space-filling curve will tend to keep objects which are close together in the higher-dimensional space. This property is necessary so that the partitions that result respect the locality of the data. Many different curves have been used for load balancing purposes, including the Peano curve, the Morton curve, and the Hilbert curve. The Hilbert curve is a common choice of space-filling curve for partitioning applications because it provides good locality.

### Graph Partitioning

The pattern of communication in any parallel program can be represented as a graph, with nodes representing discrete units of work, and weighted edges representing communication. Graph partitioning is a general-purpose technique for splitting such a graph into pieces, typically with the goal of creating one piece for each processor. There are two conflicting goals in graph partitioning: achieving equal-size partitions, and minimizing the total amount of communication across partition boundaries, known as the edge cut. Finding an optimal solution to the graph partitioning problem is NP-Hard, so heuristic algorithms are required.

Because it is computationally infeasible to attempt to partition the whole input graph at once, the graph partitioning algorithm proceeds by constructing a series of

coarser representations of the input graph by combining distinct nodes in the input graph into a single node in the coarser graph. By retaining information about which nodes in the original graph have been combined into each node of the coarse graph, the algorithm maintains basic information about the graph's structure while reducing its size enough that computing a partitioning is very simple.

Once the coarsest graph has been partitioned, the coarsening process is reversed, breaking combined nodes apart. At each step of this uncoarsening process, a refinement algorithm such as the Kernighan–Lin algorithm can be used to improve the quality of the partition by finding advantageous swaps of nodes across partition boundaries. Once the uncoarsening process is complete, the user is left with a partitioning for the original input mesh.

This technique can be further generalized to operate on hypergraphs. Whereas in a graph an edge connects two nodes, in a hypergraph a hyperedge can connect any number of nodes. This allows for the explicit representation of relationships that link several nodes together as one edge rather than the standard graph representation of pairwise edges between each of the nodes. For example, in VLSI simulations hypergraphs can be used to more accurately reflect circuit structure, leading to higher quality partitions. Hypergraphs can be partitioned using a variation of the basic graph partitioning algorithm which coarsens the initial hypergraph until it can be easily partitioned, then refining the partitioned coarse hypergraph to obtain a partitioning for the full hypergraph.

The primary advantages of load balancing schemes based on graph partitioning are their flexibility and generality. An object communication graph can be constructed for any problem without depending on any particular features of the problem domain. The resulting partition can be made to account for varying amounts of work per object by weighting the nodes of the graph and to account for varying amounts of communication between objects by weighting the edges. This flexibility makes graph partitioners a powerful tool for load balancing.

The primary disadvantage of these schemes is their cost. Compared to the other methods discussed here, graph partitioning may be computationally intensive, particularly when a very large number of partitions

is needed. The construction of an explicit communication graph may also be unnecessary for problems where communication may be inferred from domain-specific information such as geometric data associated with each object.

## Rebalancing with Refinement Versus Total Reassignment

There are two ways to approach the problem of rebalancing the computational load of an application. The first approach is to take an existing data distribution as a baseline and attempt to incrementally improve load balance by migrating small amounts of data between partitions while leaving the majority of data in place. The second approach is to perform a total repartitioning, without taking the existing data distribution into account. Incremental load balancing approaches are desirable when the application is already nearly balanced, because they impose smaller costs in terms of data motion and communication. Incremental approaches are more amenable to asynchronous implementations that operate using information from only a small set of processors.

## Software Frameworks

Many parallel dynamic load balancing libraries and frameworks have been developed before for specialized domains. These libraries are often particularly useful because they allow application developers to specify the structure of their application once and then easily try a number of different load balancing strategies to see which is most effective in practice without needing to reformulate the load measurement and task graph construction process for each new algorithm.

The Zoltan toolkit [1] provides a suite of dynamic load balancing and parallel repartitioning algorithms, including geometric, hypergraph, and graph methods. It provides a simple interface to switch between algorithms, allowing straightforward comparisons of algorithms in applications. The application developers provide an explicit cost function and communication graph for the Zoltan algorithms to use.

Charm++ [2] adopts a migratable object-based load balancing model. As such, Charm programs have a natural grain size determined by their objects, and can be load balanced by redistributing their objects among processors. Charm provides facilities for automatically

measuring load, and provides a variety of associated measurement-based load balancing strategies that use the recent past as a guideline for the near future. This avoids the need for developers to explicitly specify any cost functions or information about the application's communication structure. Charm++ includes a suite of global, distributed, and hierarchical load balancing schemes.

DRAMA [3] is a library for parallel dynamic load balancing of finite element applications. The application must provide the current distributed mesh, including information about its computation and communication requirements. DRAMA then provides it with all necessary information to reallocate the application data. The library computes a new partitioning, either via direct mesh migration or via parallel graph re-partitioning, by interfacing to the ParMetis or Jostle graph partitioning libraries. This project is no longer under active development.

The Chombo [4] package was developed by Lawrence Berkeley National Lab. It provides a set of tools including load balancing for implementing finite difference methods for the solution of partial differential equations on block-structured adaptively refined rectangular grids. It requires users to provide input indicating the computational workload for each box, or mesh partition.

## Task Scheduling Methods

Some applications are characterized by the continuous production of tasks rather than by iterative computations on a collection of work units. These tasks, which are continually being created and completed as the application runs, form the basic unit of work for load balancing. For these applications, load balancing is essentially a task scheduling or task allocation problem. This task pool abstraction captures the execution style of many applications such as master/worker and state-space search computations. Such applications are typically non-iterative.

Load balancing strategies in this category can be classified as centralized, fully distributed, or hierarchical. Hierarchical strategies are hybrids that aim to combine the benefits of centralized and distributed methods. In centralized strategies, a dedicated "central" processor gathers global information about the state of the entire machine and uses it to make global

load balancing decisions. On the other hand, in a fully distributed strategy, each processor exchanges state information only with other processors in its neighborhood.

Fully distributed load balancing received significant attention from the early days in parallel computing. One way to categorize them is based on which processor initiates movement of tasks.

Sender-initiated schemes assign newly created tasks to some processor, chosen randomly or from one of the neighbors in a physical or virtual topology. The decision to assign it to another processor, instead of retaining the task locally may also be taken randomly, or based on a load metric such as a queue size. Random assignment has some good statistical properties but suffers from high communication costs, by requiring that most tasks be sent to remote processors. With neighborhood strategies, global balancing is achieved as tasks are moved from heavily loaded neighborhood diffuse onto lightly loaded processors. In the adaptive contraction within neighborhood (ACWN) scheme, tasks always travel to topologically adjacent neighbors with the least load, but only if the difference in loads is more than a predefined threshold. In addition, ACWN does saturation control by classifying the system as being either lightly, moderately, or heavily loaded.

In receiver-initiated schemes, the underloaded processors request load from heavily loaded processors. You may chose the victim to request work from randomly, or via a round-robin policy, or from among "neighboring" processors. Randomized work stealing is yet another distributed dynamic load balancing technique, which is used in some runtime systems such as Cilk.

The distinction between sender or receiver initiation is blurred when processors exchange load information, typically with neighbors in a virtual topology. Although the decision to send work is taken by the sender, it is taken in response to load information from the receiver. For example, in neighborhood averaging schemes, after periodically exchanging load information with neighbors in a virtual (and typically, low-diameter) topology, each processor that is overloaded compared with its neighbors, sends equalizing work to its lower-loaded neighbors. Such policies tend to be proactive compared with work stealing, trading better load balance for extra communication.

Another strategy in this category, and one of the oldest ones, is the gradient model. Here each processor participates in a continuous fixed-point computation with its neighbors to identify the neighbor that is closest to an idle processor. Overloaded processors then send work toward the idle processors via that neighbor.

The gradient model is a demand-driven approach. In gradient schemes, underloaded processors inform other processors of their state, and overloaded processors respond by sending a portion of their load to the nearest lightly loaded processor. The resulting effect is a form of a gradient map that guides the migration of tasks from overloaded to underloaded processors.

The dimensional exchange method is a distributed strategy in which balancing is performed in an iterative fashion by “folding” an  $P$  processor system into  $\log P$  dimensions and balancing one dimension at a time. that is, in phase  $i$ , each processor exchanges load with its neighbor in the  $i$ th dimension so as to equalize load among the two. After  $\log P$  phases, the load is balanced across all the processors. This scheme is conceptually designed for a hypercube system but may be applied to other topologies with some modification.

Several hierarchical schemes have been proposed that avoid the bottleneck of a centralized strategies, while retaining some of their ability to achieve global balance quickly. Typically, processors are organized in a two-level hierarchy. Managers at the lower level behave like masters in centralized strategies for their domain of processors, and interact with other managers as processors in a distributed strategies. Alternatively, load balancing is initiated at the lowest levels in the hierarchy, and global balancing is achieved by ascending the tree and balancing the load between adjacent domains at each level in the hierarchy.

Often, priorities are associated with tasks, especially for applications such as branch-and-bound or searching for one solution in a state-space search. Task balancing for these scenarios is complicated because of the need to balance load while ensuring that high priority work does not get delayed by low priority work. Sender-initiated random assignment as well as some hierarchical strategies have shown good performance in this context.

## Related Entries

- ▶ [Chaco](#)
- ▶ [Graph Partitioning](#)
- ▶ [Hypergraph Partitioning](#)
- ▶ [Reduce and Scan](#)
- ▶ [Scan for Distributed Memory, Message-Passing Systems](#)
- ▶ [Space-filling Curves](#)
- ▶ [Task Graph Scheduling](#)
- ▶ [Topology Aware Task Mapping](#)

## Bibliographic Notes and Further Reading

There is a rich history of load balancing literature that spans decades. This entry is only able to cite a very small amount of this literature, and attempts to include modern papers with broad scope and great impact. Kumar et al. [5] describe the scalability and performance characteristics of several task scheduling schemes, and Devine et al. [6] gives a good overview of the range of techniques used for periodic balancing in modern scientific applications and the load balancing challenges faced by these applications. Xu and Lau [7] give an in-depth treatment of distributed load balancing articles, covering both mathematical theory and actual implementations.

For practical information on integrating load balancing frameworks into real applications, literature describing principles and practical use of such systems as DRAMA [3], Charm++ [2, 9], Chombo [4], and Zoltan [1] is a crucial resource.

More information is available on the implementation of task scheduling methods, whether they are centralized [10, 11], distributed [12–14], or hierarchical [14]. Work stealing is described in detail by Kumar et al. [5], while associated work on Cilk is described in Frigo et al. [15]. Dinan et al. [16] extended work stealing to run on thousands of processors using ARMCI.

## Bibliography

1. Devine K, Hendrickson B, Boman E, St. John M, Vaughan C (2000) Design of dynamic load-balancing tools for parallel applications. In: Proceedings of the international conference on supercomputing, Santa Fe, New Mexico, May 2000. ACM Press, New York

2. Kale LV, Zheng G (2009) Charm++ and AMPI: adaptive runtime strategies via migratable objects. In: Parashar M (ed) Advanced computational infrastructures for parallel and distributed applications. Wiley-Interscience, Hoboken, pp 265–282
3. Basermann A, Clinckemaillie J, Coupez T, Fingberg J, Digonnet H, Ducloux R, Gratien JM, Hartmann U, Lonsdale G, Maerten B, Roose D, Walshaw C (2000) Dynamic load balancing of finite element applications with the DRAMA library. *Appl Math Model* 25:83–98
4. Chombo Software Package for AMR Applications. (October 2010) <http://seesar.lbl.gov/anag/chombo/>. Accessed October 2010
5. Kumar V, Grama AY, Vempaty NR (1994) Scalable load balancing techniques for parallel computers. *J Parallel Distrib Comput* 22(1):60–79
6. Devine KD, Boman EG, Heaphy RT, Hendrickson BA, Teresco JD, Faik J, Flaherty JE, Gervasio LG (2005) New challenges in dynamic load balancing. *Appl Numer Math* 52(2–3): 133–152
7. Xu C, Lau FCM (1997) Load balancing in parallel computers theory and practice. Kluwer Academic Publishers, Boston
8. Zheng G (2005) Achieving high performance on extremely large parallel machines: performance prediction and load balancing
9. Zheng G, Meneses E, Bhatele A, Kale LV (2010) Hierarchical load balancing for Charm++ applications on large supercomputers. In: Proceedings of the third international workshop on parallel programming models and systems software for high-end computing (P2S2), San Diego
10. Chow YC, Kohler WH (1982) Models for dynamic load balancing in homogeneous multiple processor systems. *IEEE Trans Comput c-36:667–679*
11. Ni LM, Hwang K (1985) Optimal load balancing in a multiple processor system with many job classes. *IEEE Trans Software Eng SE-11:491–496*
12. Corradi A, Leonardi L, Zambonelli F (1999) Diffusive load balancing policies for dynamic applications. *IEEE Concurrency* 7(1):22–31
13. Willebeek-LeMair MC, Reeves AP (1993) Strategies for dynamic load balancing on highly parallel computers. *IEEE Trans Parallel Distrib Syst* 4(9):979–993
14. Sinha A, Kalé LV (1993) A load balancing strategy for prioritized execution of tasks. In: International parallel processing symposium, New Port Beach, CA, April 1993. IEEE Computer Society, Washington, DC, 230–237
15. Frigo M, Leiserson CE, Randall KH (1998) The implementation of the Cilk-5 multithreaded language. In: ACM SIGPLAN '98 conference on programming language design and implementation (PLDI), Montreal, vol 33 of ACM SIGPLAN notices, ACM, New York, pp 212–223
16. Dinan J, Larkins DB, Sadayappan P, Krishnamoorthy S, Nieplocha J (2009) Scalable work stealing. In: SC '09: proceedings of the conference on high performance computing networking, storage and analysis, Portland, OR. ACM, New York, pp 1–11

## Locality of Reference and Parallel Processing

KESHAV PINGALI

The University of Texas at Austin, Austin, TX, USA

### Definition

The term *locality of reference* refers to the fact that there is usually some predictability in the sequence of memory addresses accessed by a program during its execution. Programs are said to exhibit *temporal locality* if they access the same memory location several times within a small window of time. Programs that access nearby memory locations within a small window of time are said to exhibit *spatial locality*. Many parallel processors are nonuniform memory access (NUMA) machines in which the time required by a given processor to access a memory location may depend on the address of that location. Parallel programs that exploit this nonuniformity are said to exhibit *network locality*. Exploiting all types of locality is critical in parallel processing because the performance of most programs is limited by the latency of memory accesses. Hardware mechanisms for exploiting locality include caches and pre-fetching. Software mechanisms include program and data transformations, software pre-fetching, and topology-aware mappings of computations and data. Multithreading can reduce the impact of memory latency on programs that have a lot of parallelism but little locality of reference.

### Discussion

#### Introduction

The sequence of memory addresses accessed by a program or a thread during its execution is called its memory trace. Memory traces are not random number sequences since there is some predictability to the addresses in most traces, which arises from fundamental features of the von Neumann model of computation. For example, instructions between branches are executed sequentially, so if an instruction at address  $i$  is executed and it is not a branch

instruction, the instruction at address  $i + 1$  must be executed next. This kind of predictability in address traces is called locality of reference, and exploiting it is key to obtaining good performance on sequential and parallel machines.

There are three kinds of locality of reference: *temporal*, *spatial*, and *network*.

### Temporal Locality

A *program execution* exhibits temporal locality if the occurrences of a given memory address in its memory trace occur close to each other. A *program* is said to exhibit temporal locality if its executions for most inputs of interest exhibit temporal locality.

Temporal locality in instruction addresses arises mainly from the execution of loops (in particular, innermost loops in loop nests). For example, when multiplying two  $N \times N$  matrices using the standard algorithm shown below, the body of the innermost loop is executed  $N^3$  times back to back, resulting in temporal locality in instruction addresses. Temporal locality in instruction addresses may also arise from the execution of “hot” methods that are called repeatedly during program execution.

```
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
 for (k = 0; k < N; k++)
 C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

Temporal locality in data addresses may arise from accesses to constants and memory locations accessed repeatedly within loops. For example, in the matrix multiplication code shown above, every iteration of the innermost loop for given outer loop indices  $(i, j)$  accesses the same array location  $C[i][j]$ , giving rise to temporal locality. Some compilers can exploit this temporal locality by loading  $C[i][j]$  into a register just before the innermost loop begins execution, and performing the additions to this register rather than to the memory location  $C[i][j]$  (the value in the register is written back to memory when loop finishes execution). In contrast, there is little temporal locality in the references to the  $B$  array since successive accesses to a given element of  $B$  are separated by  $O(N^2)$  other addresses in the trace.

### Spatial Locality

A program execution exhibits spatial locality if the occurrences in its trace of nearby memory addresses

occur close together in that trace. A program is said to exhibit spatial locality if its executions for most inputs of interest exhibit spatial locality.

Spatial locality in instruction accesses arises because instructions between branches are executed sequentially, so if an instruction at address  $i$  is executed and it is not a branch, the instruction at address  $i + 1$  must be executed next. Instruction accesses within innermost loops therefore exhibit both temporal and spatial locality. Programs that perform operations on vectors exhibit spatial locality if successive vector elements are accessed during the computation, such as the following program which adds the elements of two vectors:

```
for (i = 0; i < N; i++)
 C[i] = A[i]+B[i];
```

Notice that in this program, there are no fewer than four sources of spatial locality: the accesses to the three vectors, and the instructions in the loop body. Programs that access vector elements with some small stride also exhibit spatial locality.

Programs that access arrays of two or more dimensions will exhibit spatial locality if array elements are accessed successively (or with some small non-unit stride) in the order in which they are stored in memory. For example, in the C programming language, arrays are stored in row-major order, so the elements of the matrix are stored row by row in memory. If a program accesses such arrays row by row, these accesses will exhibit spatial locality. In the matrix multiplication code shown above, the accesses to arrays  $A$  and  $C$  have spatial locality of reference, since these arrays are accessed row by row, but the accesses to array  $B$  do not since this array is accessed by columns.

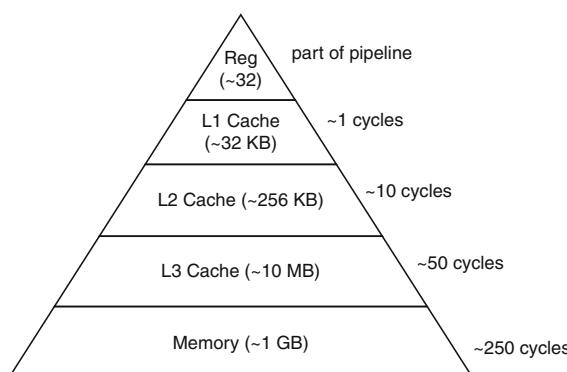
### Network Locality

In a parallel computer in which memory addresses are distributed across the processors, a processor may have faster access to memory locations mapped locally than to memory locations mapped to other processors. There may also be differences in the access times to memory locations mapped to different processors because most communication networks for parallel computers are multi-stage networks in which communication packets from a given processor may go through different numbers of stages to get to different processors, and each

stage adds some latency to the communication. Parallel computers that exhibit these kinds of nonuniform memory access times are called Non-Uniform Memory Access (NUMA) parallel computers. A parallel program that attempts to exploit features of NUMA parallel computers is said to exhibit *network locality*.

### Exploiting Locality: Caches

Cache memories exploit temporal and spatial locality of reference to provide programmers with the performance characteristics of a large and fast memory system but at reasonable cost. The key is a hierarchical organization of the storage, known usually as the memory hierarchy and shown pictorially below. On modern processors, there may be many memory hierarchy levels such as the registers, Level 1 (L1) cache, Level 2 (L2) cache, Level 3 (L3) cache, and main memory. The virtual memory system can be considered to be another level in the memory hierarchy. L1 cache memory is small, fast, and usually implemented in SRAM, while main memory is large, relatively slow, and implemented in DRAM; the other levels of cache fall somewhere in between these two extremes. For example, on the Intel Nehalem multicore processor, each core has 32 KB L1 instruction and data caches, and a 256 KB L2 cache; all the cores share a 8 MB L3 cache. The size of the main memory may be many GB. On most processors, accesses to the L1 cache typically take around 1–3 cycles, and the latency of memory accesses increases by roughly an order of magnitude in going from one level to the next (Fig. 1).



**Locality of Reference and Parallel Processing. Fig. 1**

Memory hierarchy of a typical processor

Given such a memory hierarchy, the effective latency of memory accesses as observed by the processor will be considerably less than DRAM access time provided most of the memory accesses are satisfied by the faster cache levels. One way to accomplish this is to store the addresses and contents of the most recently accessed memory locations in the cache memories. When a processor issues a memory request, the fastest cache level in which that address is cached responds to the request, so the long-latency trip to memory is required only if the address is not cached anywhere. Programs that have good temporal locality obviously benefit from this caching strategy. When a memory request goes all the way to main memory, most memory systems will fetch not just the contents of the desired address but the contents of a block of addresses, one of which is the desired address (this block of addresses is called the *cache block* or *cache line* containing that address). Because the latency of fetching a cache block from memory is almost the same as the latency of fetching the contents of a single address, programs with spatial locality benefit from this caching policy.

The design of caches becomes more complex in parallel machines because of the *cache-coherence* problem. A major advantage of caches is that they are transparent to the programmer in the sense that although caches may confer performance benefits, they do not change the output of the program. Ensuring this transparency in a shared-memory multiprocessor is complicated by the fact that a memory location may be cached in the local caches of several cores/processors, so it is necessary to ensure that updates to that address made by different processors are coordinated in some way to preserve the transparency of caching to the programmer. This is referred to as the cache-coherence problem, and many solutions to this problem have been explored in the literature.

### Program Transformations for Exploiting Caches

While caches are transparent to the programmer, obtaining the performance benefits of caching may require the programmer to be aware of the cache hierarchy and to transform programs to become more “cache-friendly” by enhancing temporal and spatial locality. For the most part, these transformations can be divided

into two classes: (1) rescheduling of operations and (2) reordering of data structures.

The goal of rescheduling operations is to ensure that operations that access the same memory address (or more generally, the same cache line) are executed more or less contemporaneously, thereby improving locality. In dense matrix computations, loop permutation and loop tiling are the most important transformations for enhancing locality. For example, consider the matrix multiplication program discussed above. As is well known, all six permutations of the three loops produce the same output; however, their locality characteristics are very different. If the *j* loop is permuted into the innermost position in the loop nest, the references to arrays C and B enjoy good spatial locality since the inner loop accesses these arrays row by row, and the reference to array A enjoys good temporal locality since all iterations of the inner loop for given outer loop indices access the same element of A. Conversely, permuting the *i* loop into the innermost position is bad for exploiting spatial locality since none of the matrices will be accessed by row in the innermost loop.

While loop permutation is useful, the most important transformation for enhancing locality is loop tiling. A tiled version of matrix multiplication is shown below. The effect of this transformation is to produce block matrix programs, in which the inner loops can be viewed as operating on blocks of the original matrices. In the tiled code shown below, BLOCK\_SIZE is a parameter that determines the size of the blocks multiplied in the inner three loops. The value of this parameter must be chosen carefully so that the working set of the three innermost loops fits in cache. If there are multiple cache levels, it may be necessary to tile for some or all these levels.

Choosing an optimal tile size is a difficult problem in general, and it may be necessary to execute the program for different tile sizes and determine the best one by measurement. This approach is called empirical optimization and it is used by the ATLAS system, which generates high-performance Basic Linear Algebra Subroutines (BLAS). It is often possible to use simple machine models to reduce the search space.

```
for (ii = 0; ii < N; ii += BLOCK_SIZE)
 for (kk = 0; kk < N; kk += BLOCK_SIZE)
 for (jj = 0; jj < N; jj += BLOCK_SIZE)
 for (i = ii; i < ii +
```

```
BLOCK_SIZE && i < N; i++)
 for (k = kk; k < kk +
 BLOCK_SIZE && k < N; k++)
 for (j = jj; j < jj +
 BLOCK_SIZE && j < N; j++)
 C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

Spatial locality can also be improved by changing the layout of data structures in memory. For example, in the *(i, j, k)* order of the three nested loop version of matrix multiplication, changing the layout of B to column-major order will improve spatial locality for the references to array B. For data transformations to be useful, the overhead of transforming the data layout must be amortized by the benefits of improved spatial locality in computing with that data. For example, when performing the tiled matrix multiplication shown above, high performance implementations will usually copy blocks of the matrices into contiguous memory locations because the overhead of copying the data is amortized by the benefits of improved spatial locality when performing the block matrix multiplication.

Many techniques for changing the layouts of other kinds of data structures have been explored in the literature. For example, records and structures that span multiple cache lines can be allocated so that fields that are often accessed contemporaneously are packed together into the same cache line, if possible, rather than being allocated in different cache lines. In managed languages like Java and C#, the garbage collector can improve locality by moving data items that are accessed contemporaneously into the same page of virtual memory.

## Cache-Oblivious Programs

Divide-and-conquer algorithms for many problems are naturally cache-friendly because they repeatedly generate sub-problems of smaller size until they reach a base case, and then assemble the solution to the problem from the solutions to these sub-problems. If the base case of the divide-and-conquer algorithm is chosen so that the working set of that sub-problem fits in the cache, and the data movement required for assembling the solution from the solutions to the sub-problems is small, the program may have excellent locality. Divide-and-conquer algorithms for problems like matrix multiplication, FFT, and Cholesky factorization enjoy this

property and they are known as cache-oblivious programs because they have good temporal locality even though they are not explicitly optimized for any particular cache size or structure; this is in contrast to iterative algorithms for these problems, which require tiling to be made cache-friendly. Spatial locality in cache-oblivious matrix multiplication programs can be enhanced by using space-filling curves to order the storage of matrix elements.

## Prefetching

Caches can be classified as latency-avoidance mechanisms since the goal is to avoid long latency memory operations by exploiting locality of reference. The performance impact of long latency memory operations can also be reduced by latency-tolerance techniques such as prefetching. If the memory accesses of a program can be predicted well in advance of when the values are actually required in the computation, the processor might be able to prefetch these values from memory into the cache so that they are available immediately when they are needed for the computation. This is an instance of a more general optimization technique in parallel programming called *overlapping communication with computation*: the key idea is to perform data movement in parallel with computation to reduce the effective overhead of the data movement. Prefetching is particularly useful for streaming programs since these programs touch large volumes of data and have predictable data accesses but have little temporal locality, so caches are not effective for reducing the effective memory latency.

In some processors, prefetching is under the control of the programmer. For example, the iA32 SSE instruction set has prefetch instructions for prefetching data into some or all cache levels. The programmer or compiler is responsible for inserting prefetch instructions into the appropriate places in the program. In other processors, prefetching is implemented in hardware. Usually, the prefetch unit monitors the stream of memory addresses from the processor and attempts to detect sequences of fixed stride memory accesses. If it finds such a sequence, it uses that sequence to determine memory addresses for prefetching. To avoid unnecessary page faults, prefetching usually does not cross page boundaries.

## Exploiting Network Locality

For techniques to exploit network locality, see the Encyclopedia entry on ► [Topology Aware Task Mapping](#).

## Using Parallelism to Tolerate Memory Latency

When programs have little locality of reference and also make unpredictable memory accesses, neither caching nor prefetching is effective in reducing the performance impact of long latency memory accesses. Many irregular programs such as graph computations fall in this category. For such programs, it is possible to tolerate latency by exploiting parallelism, provided the processor can switch rapidly between parallel threads of execution. There are many implementations of this scheme, but a simple approach is the following. The processor issues instructions from a thread until that thread becomes idle waiting for a memory load to complete. At that point, the processor switches to a different thread that is ready to execute and begins issuing instructions from that thread, and so on. As long as there is enough parallelism in the program and the processor can switch quickly between threads, the performance impact of long latency loads from memory is minimized. In other designs, the processor switches between ready threads at every cycle.

This approach requires specially designed processors since the processor needs to be able to switch between threads very rapidly. Tera's MTA processor is an example. Each processor can support 128 simultaneous threads. *Simultaneous multi-threading* (SMT) is an implementation of this idea in general-purpose processors. The same principle also underlies the design of dataflow computers, although threads in that case are very fine-grain, comprising of a single instruction.

## Related Entries

- [Numerical Linear Algebra](#)
- [Code Generation](#)
- [Loop Nest Parallelization](#)
- [Nonuniform Memory Access \(NUMA\) Machines](#)
- [Parallelization, Automatic](#)
- [Parallelization, Basic Block](#)
- [Scheduling Algorithms](#)
- [Topology Aware Task Mapping](#)
- [Unimodular Transformations](#)

## Bibliographic Notes and Further Reading

Although the importance of exploiting locality of reference in programs was recognized from the very start of computing by pioneers like von Neumann, the first machine to incorporate cache memory was the IBM 360 Model 85 [8]. Work on transforming programs to optimize them for parallelism and locality of reference began at the University of Illinois, Urbana-Champaign in the 1960s [2, 12]. The automation of loop and data transformations was enabled by the development of powerful integer linear programming tools in the 1990s [5, 6, 13]. For the most part, these techniques are restricted to dense matrix programs. Data-shackling is a more general data-centric approach to reasoning about locality [11]. Data transformations for non-array data structures are described in [4].

Determining optimal values for parameters such as tile sizes is an active topic of research. The empirical optimization approach is incorporated in the popular ATLAS system, which produces automatically tuned Basic Linear Algebra Subroutines (BLAS) [1]. An approach using simple performance models and a detailed comparison with the empirical optimization approach can be found in [14]. Cache-oblivious algorithms [7] are based on the notion of I/O complexity [9].

Latency tolerance through multithreading was implemented in the Tera computer [3]. Network locality and network-oblivious algorithms are explored in [4].

## Bibliography

1. Automatically tuned linear algebra software (ATLAS). <http://math-atlas.sourceforge.net/>
2. Allen R, Kennedy K (2002) Optimizing compilers for modern architectures. Morgan Kaufmann Publishers Inc, San Francisco
3. Alversen G, Kahan S, Korry R, McCann C, Smith B (1995) Scheduling on the Tera MTA. In: Proceedings of the workshop on job scheduling strategies for parallel processing. Lecture notes in computer science, vol 949. Springer, Berlin
4. Bilardi G, Pietracarpina A, Pucci G, Silverstri F (2007) Network-oblivious algorithms. In: Proceedings of the 21st international parallel and distributed processing symposium. IEEE Publishers, New York
5. Chilimbi TM, Larus JR (1998) Using generational garbage collection to implement cache-conscious data placement. In: Proceedings of the 1st international symposium on memory management, ISMM '98. ACM Publishers, New York

6. Cierniak M, Li W (1995) Unifying data and control transformations for distributed shared memory machines. In: Proceedings of the ACM SIGPLAN 1995 conference on programming language design and implementation, PLDI '95. ACM Publishers, New York
7. Feautrier P (1992) Some efficient solutions to the affine scheduling problem, I, one dimensional time. *Int J Parallel Prog* 21(5):313–348
8. Frigo M, Leiserson C, Prokop H, Ramachandran S (1999) Cache-oblivious algorithms. In: Proceedings of the 40th IEEE symposium on foundations of computer science (FOCS 99), New York
9. Hennessy J, Patterson D (2003) Computer architecture: a quantitative approach. Morgan Kaufmann Publishers Inc, San Francisco
10. Hong J-W, Kung H (1981) I/O complexity: the red-blue pebble game. In: Proceedings of the 13th annual ACM symposium on the theory of computing, Milwaukee
11. Intel 64 and ia-32 architectures software development manual. Order number 253666–033US, December 2009. Intel Corporation
12. Kodukula I, Ahmed N, Pingali K (1997) Data-centric multi-level blocking. In: Proceedings of the ACM SIGPLAN 1997 conference on programming language design and implementation, PLDI '97, ACM Publishers, New York
13. Kuck D, Kuhn R, Padua D, Leisure B, Wolfe M (1981) Dependence graphs and compiler optimizations. In: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on principles of programming languages, Williamsburg, 26–28 Jan 1981. POPL '81. ACM Publishers, New York
14. Wolf M, Lam M (1991) A data locality optimizing algorithm. In: Proceedings of the ACM SIGPLAN 1991 conference on programming language design and implementation, PLDI '91. ACM Publishers, New York
15. Yotov K, Li X, Ren G, Garzaran M, Padua D, Pingali K, Stodghill P (2005). Is search really necessary to generate high-performance BLAS? *Proc IEEE* 93(2):358–386

## Lock-Free Algorithms

### ► Non-Blocking Algorithms

## Locks

- Synchronization
- Transactional Memory

## Logarithmic-Depth Sorting Network

### ► AKS Network

# Logic Languages

MANUEL CARRO<sup>1</sup>, MANUEL HERMENEGILDO<sup>1,2</sup>

<sup>1</sup>Universidad Politécnica de Madrid, Madrid, Spain

<sup>2</sup>IMDEA Software Institute, Madrid, Spain

## Synonyms

Concurrent logic languages; Distributed logic languages; Prolog

## Definition

The common name of *Parallel Logic Languages* groups those languages which are based on logic programming and which, while respecting as much as possible the declarative semantics, have an operational semantics which exploits parallelism or concurrency, either explicitly or implicitly, to gain in efficiency or expressiveness.

## Discussion

### Logic Programming

The application of logic and mechanized proofs to express problems and their solutions is at the origins of computer science [6]. The basis of this approach is to express the knowledge on some problem (e.g., how a sorted tree is organized) as a consistent theory in some logic and to model the desired goal (e.g., storing an item on the tree) as a formula. If the formula is true in the theory capturing the problem conditions, then the objective expressed by such formula is achievable (i.e., the item can be stored). The set of formulas which are true in the theory are those which state the set of objectives achievable (computable) by the program. This constitutes the *declarative semantics* of logic programs.

A usual proof strategy is to add to the initial theory a formula stating the *negation* of the desired goal and to prove the inconsistency of the resulting theory by generating a counterexample. The valuation(s) of the free variable(s) in the counterexample constitute a solution to the initial problem.

As an example, Fig. 1 presents a set  $\mathcal{T}$  of formulas describing insertion in a sorted tree, where constants, predicate names, and term names are written

in lowercase, and variables start with uppercase letters, in conformance with logic programming languages. Arithmetic predicates are assumed to be available. *void* represents the empty tree. An argument of the form  $tree(L, I, R)$  represents a tree node with item  $I$  and left and right children  $L$  and  $R$ , respectively. A fact  $insert(T_1, I, T_2)$  can be inferred from this theory if  $T_2$  is the resulting tree after inserting item  $I$  in tree  $T_1$ . Given the term  $T_1$  representing a sorted tree and an item  $I$  to be stored, the inconsistency of  $\mathcal{T} \wedge \neg \exists T_2 . insert(T_1, I, T_2)$  can be proved with a counterexample for  $T_2$  which will be, precisely, the tree which results from inserting  $I$  into  $T_1$ .

From the *procedural point of view*, the method used to generate a counterexample is basically a search, where at each step one of the inference rules in the logic is applied, maybe leaving other rules pending to be applied. Making this search as efficient as possible is paramount in order to make logic a practical programming tool. This can be achieved by reducing the set of inference rules to be applied and the number of places where they can be applied.

Concrete logic programming languages specify the deduction method (*operational semantics*) further as well as the syntax and (possibly) subsets of the logic. The most successful combination is *resolution* (only one inference rule) on *Horn clauses* (conjunctions of non-negated literals implying a non-negated literal, as in Fig. 1). In particular, the usual operational semantics (SLD resolution) for the logic programming language Prolog uses the resolution principle with two rules to decide which literals the inference rule is applied to. SLD resolution in Prolog:

- Explores conjunctions in every implication from left to right, as written in the program text (computation rule).
- Tries to match and evaluate clauses from top to bottom (search rule).

From a goal like  $insert(tree(void, 3, void), 4, T_2)$ , SLD resolution selects one of the implications with a matching head, renames apart its parameters (i.e., variables with new names which do not clash with previous ones are created), and expands the right-hand side. The

$$\begin{array}{lll}
 (\forall I.insert(void, I, tree(void, I, void))) & & \wedge (1) \\
 (\forall L, I, R, It, II.insert(tree(L, I, R), It, tree(II, I, R))) \leftarrow It < I \wedge insert(L, It, II) & \wedge (2) \\
 (\forall L, I, R, It, Ir.insert(tree(L, I, R), It, tree(L, I, Ir))) \leftarrow It > I \wedge insert(R, It, Ir) & \wedge (3) \\
 (\forall L, I, R, It.insert(tree(L, I, R), It, tree(L, I, R))) \leftarrow It = I & \wedge (4)
 \end{array}$$

**Logic Languages. Fig. 1** Insertion in a sorted tree

first matching clause is (2), and expanding its body results in:

$$T_2 = tree(II, 3, void) \wedge 4 < 3 \wedge insert(void, 4, II)$$

This conjunction of goals is called a *resolvent*. The left-most one is a unification [12] (i.e., an equation which expresses syntactical equality between terms) which, when successful, generates bindings for the variables in it. In this case, as  $T_2$  is a variable, it just succeeds by assigning the term  $tree(II, 3, void)$  to it. That leaves the resolvent:

$$4 < 3 \wedge insert(void, 4, II)$$

The next goal to be solved is  $4 < 3$ , which is false, and therefore the whole conjunction is false and another alternative for the last selection is to be taken. Bindings made since the last selection are undone, and clause (3) is selected:

$$T_2 = tree(void, 3, Ir) \wedge 3 < 4 \wedge insert(void, 4, Ir)$$

In this case, the arithmetic predicate succeeds and the resolvent is reduced to  $insert(void, 4, Ir)$ . This matches the head of the first clause and succeeds with the binding  $Ir = tree(void, 4, void)$ . The initial goal is then proved with the output binding:

$$T_2 = tree(void, 3, tree(void, 4, void))$$

which represents a tree with a 3 at its root and a 4 in its (unique) right subtree.

Logic programming languages include additional nonlogical constructs to perform I/O, guide the execution, and provide other facilities. They also offer a compact, programming-oriented syntax (e.g., avoiding as much as possible quantifiers; compare the rest of the

programs in this text with the mathematical representation of Fig. 1).

### Operational View and Comparison with Other Programming Paradigms

Due to the fixed computation and search rules, SLD resolution can be viewed as a relatively simple operational semantics that has significant similarities to that of other programming paradigms. This provides a highly operational view of resolution-based theorem proving.

In fact, in the procedural view, SLD is akin to a traditional execution that traverses clauses sequentially from left to right, in the same way execution traverses procedure bodies in traditional languages. Predicates take the role of procedures, clause head arguments are procedure arguments, clauses are similar to case statements, etc. Predicate invocation takes the role of procedure calling and unification amounts to parameter passing (both for input and output). Logical variables are similar to (declarative) pointers that can be used to pass data or pointers inside data structures. Thus, pure deterministic logic programs can be seen as traditional programs with pointers and dynamic memory allocation but without destructive update. In addition, in nondeterministic programs clause selection offers a built-in backtracking-based search mechanism not present in other paradigms, which allows exploring the possible ways to accomplish a goal. Finally, unification provides a rich, bidirectional pattern matching-style parameter passing and data access mechanism. Similarly, logic programming (specially variants that support higher order) can be seen as a generalization of functional programming where pattern matching is bidirectional, logic variables are allowed, and more than one definition is possible for each function. See, for example, [8] for a more detailed discussion of the correspondence between logic programming and other programming paradigms.



## Toward Parallelism and Concurrency

*Parallel logic programming languages* aim at retaining the observable semantics while speeding up the execution w.r.t. a sequential implementation. *Concurrent logic languages* (►Sect. Concurrency in Logic Programming) focus more on expressiveness and often adopt an alternative semantics. Opportunities for parallel and concurrent execution come from relaxing the left-to-right, top-to-bottom order of the computation and search rules: it is, in principle, possible to select *several different literals* in a logical conjunction to resolve against simultaneously (*and-parallelism*), or explore *several different clauses* at the same time (*or-parallelism*) [4, 7].

The distinction between parallel and concurrent logic languages is not as clear-cut as implied so far. In fact, parallel execution models have been proposed for concurrent logic languages in which independent parts of the execution of concurrent processes are executed in different processors simultaneously. Conversely, the language constructs developed for parallelism have also been used for concurrency and distributed execution. The current trend is toward designing languages that combine in useful ways parallelism, concurrency, and search capabilities.

Most parallel logic programming languages have been implemented for shared-memory multicollectors. There is no *a priori* reason not to execute a logic program in a distributed fashion, as demonstrated by distributed implementations [11]. However, the need to maintain a consistent binding store seen by all the agents involved in an execution imposes an overhead which does not appear in the shared memory case. The focus will, therefore, be on linguistic constructions which, although not necessarily requiring shared-memory computers, have been mainly exploited in this kind of architectures.

## Exploiting Parallelism in Logic Programming

Independently from whether parallelism is exploited among conjunctions of literals or among different clauses, it can be detected in several ways:

**Implicit parallelism:** The compiler and run-time system decide what can be executed in parallel with no programmer intervention, analyzing, for example, data dependencies. One disadvantage of this

approach is that it may impose considerable run-time overhead, and the programmer cannot give clues as to what can (or should) be executed in parallel.

**Explicit parallelism:** The compiler and run-time system know beforehand which clauses/goals are to be executed in parallel typically via programmer annotations. This has the disadvantage that it puts the responsibility of parallelization (a hard task) solely in the hands of the programmer.

**Combinations:** This is realized in practice as a combination of handwritten annotations, static analysis, and run-time checks.

The last option is arguably the one with the most potential, as it aims at both reducing run-time effort and simplifying the task of the programmer, while still allowing annotating programs for parallel execution. In the following sections, variants of these sources of parallelism will be revised briefly, assuming that annotations to express such parallelism at the source language level are available and the programs have been annotated with them. Methods for introducing these annotations automatically (briefly mentioned in ►Sect. Automatic Detection of Parallelism) are the subject of another entry (►Parallelization, Automatic).



## And-Parallelism

And-parallelism stems from simultaneously selecting several literals in a resolvent (i.e., several steps in a procedure body). While the semantics of logic programming makes it possible to safely select *any* goal from the resolvent, practical issues make it necessary to carefully choose which goals can be executed in parallel. For example, the evaluation of arithmetic expressions in classical Prolog needs variables to be adequately instantiated, and therefore the sequential execution order should be kept in some fragments; input/output also needs to follow a fixed execution order; and tasks which are too small may not benefit from parallel execution [8].

Whether a sequential execution order has to be respected (either for efficiency or correctness) can be decided based on *goal independence*, a general notion which captures whether two goals can proceed without mutual interference. Deciding whether the amount of

work inside a goal is enough to benefit from parallel execution can be done with the use of (statically inferred) cost (complexity) functions. With them, the expected sequential and parallel execution time can be compared at compile time or at run time (but before executing the parallel goals).

### An Illustrating Example

The code in Fig. 2 implements a parallel matrix by matrix multiplication. Commas denote sequential conjunctions (i.e., A, B means “execute goal A first, and when successfully finished, execute goal B”), while goals separated by an ampersand (A & B) can be safely executed in parallel.

Matrices are represented using nested lists, and matrix multiplication is decomposed into loops performing vector by matrix and vector by vector multiplication, which are in turn expressed by means of recursion. Every iteration step can be performed independently of the others in the same loop and therefore parallelism can be expressed by stating that predicate calls in the body of the recursive clauses (e.g., mat\_vec\_multiply in mat\_mat\_multiply) can be executed in parallel with the corresponding recursive call (mat\_mat\_multiply). Note that the arithmetic operation R is Prod + NewRes is performed after executing the other goals in the clause, because it needs NewRes to be computed.

```
mat_mat_multiply([], _, []).
mat_mat_multiply([V0|Rest], V1, [R|Os]) :-
 mat_vec_multiply(V1, V0, R) &
 mat_mat_multiply(Rest, V1, Os).

mat_vec_multiply([], _, []).
mat_vec_multiply([V0|Rest], V1, [R|Os]) :-
 vec_vec_multiply(V0, V1, R) &
 mat_vec_multiply(Rest, V1, Os).

vec_vec_multiply([], [], 0).
vec_vec_multiply([H1|T1], [H2|T2], R) :-
 Prod is H1*H2 &
 vec_vec_multiply(T1, T2, NewRes),
 R is Prod + NewRes.
```

Logic Languages. Fig. 2 Matrix multiplication

The & construct is similar to a FORK-JOIN, where the existence of A & B implicitly marks a FORK for the goals A and B and a JOIN after these two goals have finished their execution. Such parallel execution can be nested to arbitrary depths. The strict fork-join structure is also not compulsory and other, more flexible parallelism primitives are also available.

A relevant difference with procedural languages is the possible presence of search, that is, a goal can yield several solutions on backtracking, corresponding to the selection of different clauses, or even fail without any solution. Parallel execution should cope with this: when A & B can yield several solutions, an adequate operational semantics (and underlying execution mechanism) has to be adopted. Several approaches exist depending on the relationship between A and B. Perhaps, the simplest one is to execute in parallel and in separate environments A and B, gather all the solutions, and then build the cross product of the solutions. However, even if A and B can be safely executed in parallel, this approach is often not satisfactory for performance reasons.

### Independent And-Parallelism

Independent and-parallelism (IAP) only allows the parallel execution of goals which do not interfere through, for example, bindings to variables, input/output, assertions onto the shared database, etc. In absence of side effects, independence is customarily expressed in terms of allowing only compatible accesses to shared variables.

Independent and-parallelism can be *strict* or *non-strict*. The strict variant (SIAP) only allows parallel execution between goals which do not share variables at run time. The non-strict variant (NSIAP) relaxes this requirement to allow parallel execution of goals sharing variables as long as at most one of the parallel goals attempts to bind/check them or they are bound to compatible values. The matrix multiplication (Fig. 2) features SIAP if it is assumed that the input data (the first two matrices) does not contain any variable at all (i.e., these matrices are *ground*) and therefore independence is ensured as no variables can be shared.

As a further example, Fig. 3, left, is an implementation of the QuickSort algorithm, where it is assumed that a ground list of elements is given in order to be sorted. In this case, partition splits the input list into two lists which are recursively sorted and joined

```
qsort([], []).
qsort([X|Xs], Sorted) :-
 partition(Xs, X, Large, Small),
 qsort(Small, SmallSorted),
 qsort(Large, LargeSorted),
 append(Small, [X|Large], Sorted).
```

```
% Partition is common to all
% QuickSort versions
partition([], _, [], []).
partition([X|Xs], P, [X|Ls], Ss) :-
 X > P, partition(Xs, P, Ls, Ss).
partition([X|Xs], P, Ls, [X|Ss]) :-
 X <= P, partition(Xs, P, Ls, Ss).
```

**Logic Languages.** Fig. 3 QuickSort: sequential version (*left*) and SIAP version (*right*)

```
qsort(Unsrt, Sorted) :-
 qs(Unsrt, Sorted, []).
qsort([], S, S).
qsort([X|Xs], SortedSoFar, Tail) :-
 partition(Xs, X, Large, Small),
 qsort(Small, SortedSoFar, Rest),
 qsort(Large, Rest, Tail).
```

```
qsort(Unsrt, Sorted) :-
 qs(Unsrt, Sorted, []).
qsort([], S, S).
qsort([X|Xs], SortedSoFar, Tail) :-
 partition(Xs, X, Large, Small),
 qsort(Small, SortedSoFar, Rest) &
 qsort(Large, Rest, Tail).
```

**Logic Languages.** Fig. 4 QuickSort, using difference lists and annotated for non-strict and-parallelism

afterward. The code on the right is the SIAP parallelization, correct under these assumptions, where the two calls to QuickSort are scheduled for parallel execution.

A perhaps more interesting example is the implementation of QuickSort in Fig. 4. The sequential version on the left uses a technique called *difference lists*. The sorted list is, in this case, not ended with a `nil`, but with a free variable (a “logical pointer”) which is carried around. When this variable is instantiated, the tail of the difference list is instantiated. This makes it possible to append two lists in constant time. A procedural counterpart would carry around a pointer to the last element of a list, so that it can be modified directly. The two calls to `Qsort` share a variable and cannot be executed in parallel following the SIAP scheme. However, since only the second call will attempt to bind it they can be scheduled for NSIAP execution.

A last example of IAP (this time including some search) appears in Fig. 5. It is a solution to the problem

of placing  $N$  queens on an  $N \times N$  chessboard so that no two queens attack each other. It places the queens column by column picking (with `select/3`) a position from a list of candidates. `not_attack/3` checks that this candidate does not attack already placed queens. Search is triggered by the multiple solutions of `select/3`, and the program can return all the solutions to the queens problem via backtracking.

Checking that queens are attacked is independent from trying to place the rest of the queens, and so the two goals can be run in parallel, performing some speculative work. Unlike the previous examples, the `not_attack/3` check can fail and force the failure of a series of parallel goals which should be immediately stopped.

Applying the execution scheme sketched in ▶Sect. An Illustrating Example (collecting solutions and making a cross product) is not appropriate: if a candidate queen cannot be placed on the chessboard, the work invested in trying to build the rest of the solutions is

```

queens(N, Qs) :-

 range(1, N, Ns),

 queens(Ns, [], Qs).

queens([], Qs, Qs).

queens(UnplacedQs, SafeQs, Qs) :-

 select(UnplacedQs, UnplacedQs1, Q),

 not_attack(SafeQs, Q, 1) &

 queens(UnplacedQs1, [Q|SafeQs], Qs).

not_attack([], _, _).

not_attack([Y|Ys], X, N) :-

 X =\= Y+N, X =\= Y-N,

 N1 is N+1,

 not_attack(Ys, X, N1).

select([X|Xs], Xs, X).

select([Y|Ys], [Y|Zs], X) :-

 select(Ys, Zs, X).

```

**Logic Languages. Fig. 5** N-Queens annotated for and-parallelism

wasted. If only one solution is needed and the candidate queen is correctly placed, the work spent in finding all the solutions is wasted.

One workaround is to adapt the sequential operational semantics to the parallel case and perform recomputation. Assuming the conjunction A & B & C, forward execution is done as in a FORK-JOIN scheme, and backtracking is performed from right to left. New solutions are first sought for C and combined with the existing ones for A and B. When the solutions for C are exhausted, B is restarted to look for another solution while C is relaunched from the beginning. When the goal A finitely fails, the whole parallel conjunction fails. This generates the solutions in the same order as the sequential execution, which may be interesting for some applications.

Additionally, in the case of IAP, if one of the goals (say, B) fails without producing any solution, the parallel conjunction can immediately fail. This is correct because B does not see bindings from A, and therefore no solution for A can cause B not to fail. This semi-intelligent backtracking (combined with some constraints on scheduling) guarantees the no-slowdown property: an IAP execution will never take longer than a sequential one (modulo overheads due to scheduling and goal launching).

### Dependent And-Parallelism

IAP's independence restriction has its roots in technical difficulties: a correct, efficient implementation of a language with logical variables and backtracking and in which multiple processing units are concurrently trying to bind these variables is challenging – that is the scenario in Dependent And-Parallelism (DAP). On one

hand, every access to (potentially) shared variables and their associated internal data structures has to be protected. Additionally, when two processes disagree on the value given to a shared variable, one of them has to backtrack to generate another binding. Parallel goals sharing the variable whose binding is to be undone may have to backtrack if the binding was used to select which clause to take.

However, since DAP subsumes IAP, it offers *a priori* more opportunities for parallelism: in any of the Quick-Sort examples, the lists generated by `partition` can be directly fed to the `qsort` goals, which could run in parallel between them and with the `partition` predicate. When a new item is generated, the corresponding `qsort` call will wake up and advance the sorting process.

If the goals to be executed in parallel do not bind shared variables differently (i.e., they *collaborate* rather than *compete*), the problem of ensuring binding coherence disappears (it is, in fact, NSIAP). Moreover, if they are deterministic, there is no need to handle backtracking, and the parallel goals act as if they were producers and consumers. This is called (deterministic) *stream and-parallelism*, and can be (efficiently) exploited. The QuickSort program is an example when it is called with a free variable as second argument, because this ensures that sorting will not fail and will produce a single solution. On the other hand, an initial goal such as `?- qsort([1,2], [2,1])` will fail, and cannot be executed using stream parallelism.

Despite its complexity, proposals which fully deal with DAP exist. One approach to avoid considering all variables as potentially shared is to mark possibly shared variables at compile time, so that only these are tested at

run time [16] (note that in the example above X and Y are *aliased* after the execution of q):

```
p(X, Y) :-
q(X, Y),
dep([X, Y]) => r(X) & s(Y) & t(Y).

q(A, A).
```

Shared variables are given a special representation to, for example, make it possible to lock them. In order to control conflicts in the assignment to variables, goals are dynamically classified as producers or consumers of variables as follows: for a given variable X, the leftmost goal which has access to X is its producer, and can bind it. The rest of the goals are consumers of the variable, and they suspend should they try to bind the variable. In the previous code, r would initially be the producer of X (which is aliased to Y by q/2) and s and t its consumers. When r finishes, s becomes the producer of Y. The overhead of handling this special variable representation is large and impacts sequential execution: experiments show a reduction of 50% in sequential speed.

### Goal-Level and Binding-Level Independence

While IAP and DAP differ in the kind of problems they attack and the techniques needed to implement them, there are unifying concepts behind them: both IAP and DAP need, at some level, operations which are executed atomically and without interference. In the case of DAP, binding a shared variable and testing that this binding is consistent with those made by other processes is an atomic operation. Stream and-parallelism avoids testing

```
:- parallel select/3.

queens(N, Qs) :-
range(1, N, Ns),
queens(Ns, [], Qs).

queens([], Qs, Qs).
queens(UnplacedQs, SafeQs, Qs) :-
select(UnplacedQs, UnplacedQs1, Q),
not_attack(SafeQs, Q, 1),
queens(UnplacedQs1, [Q|SafeQs], Qs).
```

for consistency by requiring that bindings by different processes do not interfere; thereby it exhibits independence at the level of individual bindings, while in IAP noninterference appears at the level of complete goals. The requirement that variables are not concurrently written to makes variable locking unneeded for IAP. Other proposals (►Sect. Eventual and Atomic Tells) have a notion of independence at the level of groups of unifications explicitly marked in the program source.

### Or-Parallelism

*Or-Parallelism* originates from executing in parallel the different clauses which can be used to solve a goal in the resolvent. Multiple clauses, as well as the continuation of the calling goal, can thus be simultaneously tried. Or-parallelism allows searching for solutions to a query faster, by exploring in parallel the search space generated by the program. For example, in the queens program of Fig. 6 the two clauses of the `select/3` predicate can proceed in parallel *as well as* the calls to `not_attack/3` and `queens/3` which correspond to every solution of `select/3`: branches *continue in parallel* after the predicate which forked the execution has finished and into the continuations of the clauses calling such predicate. This means that each possible selection of a queen will be explored in parallel.

In principle, all the branches of the search tree of a program are independent and thus or-parallelism can apparently always be exploited. In practice, dependencies appear due to side effects and solution order-dependent control structures (such as the “cut”), and must be taken into account. Control constructs that are more appropriate for an or-parallel context have also

```
not_attack([], _, _).
not_attack([Y|Ys], X, N) :-
X =\= Y+N, X =\= Y-N,
N1 is N+1,
not_attack(Ys, X, N1).

select([X|Xs], Xs, X).
select([Y|Ys], [Y|Zs], X) :-
select(Ys, Zs, X).
```

Logic Languages. Fig. 6 N-Queens annotated for or-parallelism

been proposed. Much work has also been devoted to developing efficient task scheduling algorithms and, in general, or-parallel Prolog systems [1, 7, 13].

Or-parallelism frequently arises in applications that explore a large search space via backtracking such as in expert systems, optimization and relaxation problems, certain types of parsing, natural language processing, scheduling, or in deductive database systems.

### Data Parallelism and Unification Parallelism

Most work in parallel logic languages assumed an underlying MIMD execution model of parallel programs which were initially written for sequential execution. However, several approaches to exploiting SIMD parallelism exist.

And-data-parallelism, as in [2], tries to detect loops which apply a given operation to all the elements of a data structure in order to unroll them and apply the operation in parallel to every data item in the structure. In order to access quickly the different elements in the data structure, parallel unification algorithms were developed. This kind of data parallelism can be seen as a highly specialized version of independent and parallelism [9]. Or-data-parallelism, as in [17], changes the standard head unification to keep several environments simultaneously. Unifications of several heads can then be executed in parallel.

### Combinations of Or- and And-Parallelism

Combining or- and and-parallelism is possible, and in principle it should be able to exploit more opportunities for parallel execution. As an example, Fig. 7 shows the *Queens* program annotated to exploit

```

:- parallel select/3.

queens(N, Qs) :-
 range(1, N, Ns),
 queens(Ns, [], Qs).

queens([], Qs, Qs).
queens(UnplacedQs, SafeQs, Qs) :-
 select(UnplacedQs, UnplacedQs1, Q),
 not_attack(SafeQs, Q, 1) &
 queens(UnplacedQs1, [Q|SafeQs], Qs).

not_attack([], _, _).
not_attack([Y|Ys], X, N) :-
 X =\= Y+N, X =\= Y-N,
 N1 is N+1,
 not_attack(Ys, X, N1).

select([X|Xs], Xs, X).
select([Y|Ys], [Y|Zs], X) :-
 select(Ys, Zs, X).

```

Logic Languages. Fig. 7 N-Queens annotated for and- and or-parallelism

and+or-parallelism, by combining the annotations in Figs. 5 and 6. The efficient implementation of and+or-parallelism is difficult due to the antagonistic requirements of both cases: while and-parallelism needs variables shared between parallel goals to be accessible by independent threads, or-parallelism needs shared variables to have distinct instantiations in different threads. Restricted versions of and+or-parallelism have been implemented: for example, in [5], teams of processors execute deterministic dependent and-parallel goals. When a nondeterministic goal is to be executed, the team moves to a phase which executes the goal using or-parallelism. Every branch of the goal is deterministic, from the point of view of the agent executing it, so deterministic dependent and-parallelism can be used again.

### Automatic Detection of Parallelism

A number of parallelizing compilers have been developed for (constraint) logic programming systems [3, 10, 14]. They exemplify a *combined* approach to exploiting parallelism (►Sect. Exploiting Parallelism in Logic Programming): the compiler aims at automatically uncovering parallelism in the program by producing a version annotated with parallel operators, which is then passed to another stage of the compiler to produce lower-level parallel code. At the same time, programmers can introduce parallel annotations in the source program that the parallelizer will respect (and check for correctness, possibly introducing run-time checks for independence). The final parallelized version could always have been directly written by hand, but of course the possibility of being able to automatically parallelize (parts of)

programs initially written for sequential execution is very appealing. The parallelization can leave some independence checks for run time, also combining static and dynamic techniques. Experimental results have shown that this approach can uncover significant amounts of parallelism and achieve good speedups in practice [3, 14] with minimal human intervention. See the entry ▶Parallelization, Automatic for more details on this topic.

### Extensions to Constraint Logic Programming

Constraint logic programming extends logic programming with constraint solving capabilities by generalizing unification while preserving the basic syntax and resolution-based semantics of traditional logic programming. These extensions open a large class of new applications and also bring improved semantics for certain aspects of traditional logic programming, such as fully relational support for arithmetic operations. Constraint logic programs also explore a search space and Or-parallelism is directly applicable in this context. Parallelism can also be exploited within the constraint solving process itself in a similar way to unification parallelism. Finally, and-parallelism can also be exploited, and the constraints point of view brings new, interesting generalizations of the concepts of independence.

### Concurrency in Logic Programming

Concurrent logic programming languages [15, 18] do not aim primarily at improving execution speed but rather at increasing language expressiveness. As a consequence, their concurrency capabilities remain untouched in single-processor implementations. A strong point of these languages is the capability to express reactive systems. This affects their view of non-determinism: sequential logic programs adopt *don't know* non-determinism, where predicate clauses express different ways to achieve a goal, but it is not known which one of these clauses leads to a solution for a given problem instance before their execution successfully finishes. Unsuccessful branches can be discarded (i.e., backtracked over) at any moment and their intermediate results are undone, except for side effects such as input/output or changes to the internal database. These computations are not reactive, as their only “real” result is the substitutions produced after success.

*Don't care* non-determinism reflects the fact that, among the clauses that are eligible to generate a solution, any of them will do the job, and it does not matter which one is selected. Once a clause is committed to, the computation it performs is visible by the rest of the processes collaborating in the computation – hence the reactivity of the language – and backtracking cannot make this effect disappear.

### Syntax and (Intuitive) Semantics

A general, simplified syntax which is adopted in several proposals for concurrent logic languages will be used here. A concurrent logic program is a collection of clauses of the form

$$\text{Head} :- \text{Ask} : \text{Tell} \mid \text{Body}.$$

where variables have been abstracted. Their meaning is:

- *Head* is a literal whose formal arguments are distinct variables.
- *Ask*, the clause guard, is a conjunction of *constraints* (equality, disequality, simple tests on variables, and arithmetic predicates in the simplest case). These constraints are used to check the state of the head variables and do not introduce new bindings. If the *Ask* constraints succeed, the clause can be selected. Otherwise, the clause suspends until it becomes selectable or until another clause of the same predicate is selected to be executed (only one clause of a predicate can be selected to execute a given goal). If all clauses for all pending goals are suspended, the computation is said to *deadlock*.
- *Tell* is a conjunction of constraints which, upon clause selection, are to be executed to create new bindings, which are seen inside and outside the clause.
- *Body* is a conjunction of goals which can be *true* (in which case it can be omitted) to express the end of a computation, a single literal to express the transformation of a process into another process, or a conjunction of literals to denote spawning of a series of processes interconnected by means of shared variables.

**Fig. 8** (taken from [15]), left, shows an example. A nondeterministic merger accepts inputs from two lists of elements and generates an output list taking elements from both lists while respecting the relative order in them. Clauses are selected first depending on the

```

merge(In1, In2, Out) :-
 In1 = [X|Ins] : Out = [X|Rest] |
 merge(Ins, In2, Rest).
merge(In1, In2, Out) :-
 In2 = [X|Ins] : Out = [X|Rest] |
 merge(In1, Ins, Rest).
merge(In1, In2, Out) :-
 In1 = [] : Out = In2 | true.
merge(In1, In2, Out) :-
 In2 = [] : Out = In1 | true.

use_merge(L) :-
 skip_starting(0, 2, Evens),
 skip_starting(1, 2, Odds),
 merge(Evens, Odds, L).

skip_starting(Base, Skip, List) :-
 List = [Base|Rest] |
 NextBase is Base + Skip,
 skip_starting(NextBase, Skip, Rest).

```

**Logic Languages.** Fig. 8 Stream merger (*left*) and how to use it (*right*)

availability of elements in the input lists: when there is an element available in some input list (e.g.,  $In1 = [X|Ins]$ ), it is used to construct stepwise the output list ( $Out = [X|Rest]$ ), and the process is recursively called with new arguments.

The use of such a merger is shown in Fig. 8, right. Predicate `use_merge` is expected to return in `L` the list of natural numbers. Three processes are spawned: two of them generate a list of even numbers starting at 0 (resp. odd and starting at 1) and the third one merges these lists. The shared variables `Evens` and `Odds` act as communicating channels which are read by `merge` as they are instantiated.

**A note on syntax:** Other concurrent languages have adopted a different syntax to distinguish *Tell* from *Ask* constraints and, in some cases, to specify input and output modes. For example, the original syntax by Ueda [19] uses head unification for *Ask* constraints, and the first clause of `merge` would therefore be:

```

merge([X|Ins], In2, Out) :-
 Out = [X|Rest] |
 merge(Ins, In2, Rest).

```

Other languages (e.g., Parlog) include mode declarations to specify which formal arguments are going to be read from or written to, and require the programmer to provide them to the compiler.

Another example of the expressiveness of concurrent logic languages (which is in some sense related to that of lazy functional languages) appears in Fig. 9, which shows a solution [15] to the generation of the Hamming sequence: the ordered stream of numbers of the form  $2^i3^j5^k$ , where  $i,j,k$  are natural numbers, without repetition. The standard procedural solution keeps

a pool of candidates to be the next number and outputs the smallest one when it is sure that no smallest candidate can be produced.

Fig. 9 assumes that head unifications are *Ask* guards. The concurrent logic solution starts with a *seed* for  $i = j = k = 0$  and spawns three `multiply` processes which generate, from an existing partial stream, streams `X2`, `X3`, and `X5` with a last element depending on the previous last. These streams are merged, filtering duplicates, by two `ord_merge` predicates. The result of the last `ord_merge` predicate is fed back to the three `multiply` processes which can take then a step to generate the next candidate item in the answer stream every time `Xs` is further instantiated.

### Variations on Semantics

There are some variability points on the semantics of concurrent logic languages which, in addition to slightly different syntax, cause large differences among the languages. These semantic differences also have an important impact on the associated implementation techniques. It is remarkable that the history of concurrent logic languages was marked by a “simplification” of the semantics, which brought about less expressiveness of the language (such as, for example, eliminating the possibility of generating multiple solutions) and, accordingly, less involved implementations [18]. This has been partly attributed to such very expressive mechanisms not being needed for the tasks these languages were mainly used for, and in part to the complication involved in their implementation. Other research aimed at including concurrency without giving up multiple solutions includes some concurrent constraint languages such as Oz and AKL, as well as more traditional

```

hamming(Xs) :-

 multiply([1|Xs], 2, X2),

 multiply([1|Xs], 3, X3),

 multiply([1|Xs], 5, X5),

 ord_merge(X2, X3, X23),

 ord_merge(X23, X5, Xs).

multiply(In, N, Out) :-

 In = [] : Out = [].

multiply(In, N, Out) :-

 In = [X|Xs] :

 Out = [Y|Ys], Y is X * N |

 multiply(Xs, N, Ys).

ord_merge([X|Xs], [Y|Ys], Out) :-

 Out = [X|Os] |

 ord_merge(Xs, Ys, Os).

ord_merge([X|Xs], [Y|Ys], Out) :-

 X < Y : Out = [X|Os] |

 ord_merge(Xs, [Y|Ys], Os).

ord_merge([X|Xs], [Y|Ys], Out) :-

 X > Y : Out = [Y|Os] |

 ord_merge([X|Xs], Ys, Os).

ord_merge([], Ys, Out) :- Out = Ys.

ord_merge(Xs, [], Out) :- Out = Xs.

```

Logic Languages. Fig. 9 Generation of the Hamming sequence

logic programming systems which include extensions for distributed and concurrent execution (e.g., Ciao, SWI, or Yap).

### Deep and Flat Guards

*Flat guard* languages restrict the *Ask* guards to simple primitive operations (i.e., unifications, arithmetic predicates, etc.). Deep guards, on the other hand, can invoke arbitrary user predicates. Any bindings generated by deep guards have to be isolated from the rest of the environment (e.g., the caller) in order not to influence other guards which may be suspended or under evaluation.

Deep guards are more expressive than flat guards but the implementation techniques are much more complex. The technology necessary is related to that of the implementation of or-parallel systems, as bindings to variables in deep guards need to be kept in separate environments, just as in the or-parallel execution of logic programs.

### Fairness

The question of whether processes which can progress will effectively do so is decomposed into two different concepts, *or-fairness* and *and-fairness*, respectively, related to which clause (among those satisfying their guards) and process (among the runnable ones) is selected to execute.

*Or-fairness* can be exemplified with the merger in Fig. 8. For infinite stream producers, it does not

guarantee that items from any of the input streams will eventually be written onto its output stream. Whether this happens will depend on the concrete semantics under which it is being executed. If this is guaranteed to eventually happen, then the language has *or-fairness*.

*And-fairness* concerns the selection of which process can progress among the runnable ones. In Fig. 8, right, the two instances of *skip\_starting* do not need to suspend waiting for any condition. Therefore, it would be possible that one of them runs without interruption, generating a stream of numbers which are consumed by *merge*, which would forward them to its output list. Had *ord\_merge* been used instead of *merge*, the stream producers would alternatively suspend and resume.

### Eventual and Atomic Tells

*Tell* guards can be executed at once before the goals in the body, or be separately scheduled. In the former case, *Tells* are said to be *atomic* because all the bindings in the *Tell* are generated at once. In the latter case, they are said to be *eventual*, as they are handled according to the *and-fairness* rules of the language. *Atomic Tell* makes it possible to write some algorithms more elegantly than *eventual Tell*. However, *eventual Tell* make it possible to treat unifications and constraints more homogeneously, and are more adequate to reason on the actual properties of distributed implementations of concurrent logic languages.

## Related Entries

- ▶ [Functional Languages](#)
- ▶ [Parallelization, Automatic](#)
- ▶ [Prolog Machines](#)

## Bibliographic Notes and Further Reading

Pointers to a brief selection of the related literature have been provided within the text and are included in the references. For a much more detailed coverage of this very large topic the reader is referred to the comprehensive surveys in [7, 15, 18].

## Bibliography

1. Ali K, Karlsson R (1990) The Muse approach to or-parallel Prolog. *Int J Parallel Program* 19(2):129–162
2. Bevemyr J, Lindgren T, Millroth H (1993) Reform Prolog: the language and its implementation. In: Warren DS (ed) *Proceedings of tenth international conference on logic programming*. MIT Press, Cambridge, pp 283–298
3. Bueno F, García de la Banda M, Hermenegildo M (March 1999) Effectiveness of abstract interpretation in automatic parallelization: a case study in logic programming. *ACM Trans Program Lang Syst* 21(2):189–238
4. Conery JS (1987) Parallel execution of logic programs. Kluwer, Norwell
5. Costa VS, Warren DHD, Yang R (1996) Andorra-I compilation. *New Gener Comput* 14(1):3–30
6. Green C (1969) Theorem proving by resolution as a basis for question-answering systems. In: *Machine intelligence*, 4
7. Gupta G, Pontelli E, Ali K, Carlsson M, Hermenegildo M (July 2001) Parallel execution of Prolog programs: a survey. *ACM Trans Program Lang Syst* 23(4):472–602
8. Hermenegildo M (December 2000) Parallelizing irregular and pointer-based computations automatically: perspectives from logic and constraint programming. *Parallel Comput* 26(13–14):1685–1708
9. Hermenegildo M, Carro M (July 1996) Relating data-parallelism and (and-) parallelism in logic programs. *Comput Lang J* 22(2/3):143–163
10. Hermenegildo M, Greene K (1991) The &-Prolog system: exploiting independent and-parallelism. *New Gener Comput* 9(3,4):233–257
11. Kacsuk P (1992) Distributed data driven Prolog abstract machine (3DPAM). In: Kacsuk P, Wise MJ (eds) *Implementations of distributed Prolog*. Wiley, Aachen, pp 89–118
12. Lloyd JW (1987) Foundations of logic programming, 2nd extended edn. Springer, New York

13. Lusk E, Butler R, Disz T, Olson R, Stevens R, Warren DHD, Calderwood A, Szeredi P, Brand P, Carlsson M, Ciepielewski A, Hausman B, Haridi S (1988) The aurora or-parallel Prolog system. *New Gener Comput* 7(2/3):243–271
14. Muthukumar K, Bueno F, García de la Banda M, Hermenegildo M (February 1999) Automatic compile-time parallelization of logic programs for restricted, goal-level, independent and-parallelism. *J Log Program* 38(2):165–218
15. Shapiro EY (September 1989) The family of concurrent logic programming languages. *ACM Comput Surv* 21(3):412–510
16. Shen K (November 1996) Overview of DASWAM: exploitation of dependent and-parallelism. *J Log Program* 29(1–3):245–293
17. Smith D (1996) MultiLog and data or-parallelism. *J Log Program* 29(1–3):195–244
18. Tick E (May 1995) The deevolution of concurrent programming languages. *J Log Program* 23(2):89–124
19. Ueda K (1987) Guarded Horn clauses. In: Shapiro EY (ed) *Concurrent Prolog: collected papers*. MIT Press, Cambridge, pp 140–156

## LogP Bandwidth-Latency Model

- ▶ [Bandwidth-Latency Models \(BSP, LogP\)](#)

## Loop Blocking

- ▶ [Tiling](#)

## Loop Nest Parallelization

UTPAL BANERJEE

University of California at Irvine, Irvine, CA, USA

## Synonyms

- [Parallelization](#)

## Definition

*Loop Nest Parallelization* refers to the problem of finding parallelism in a perfect nest of sequential loops. It typically deals with transformations that change the execution order of iterations of such a nest  $L$  by creating a different nest  $L'$ , such that  $L'$  is equivalent to  $L$  and some of its loops can run in parallel.

## Discussion

### Introduction

For a better understanding of the current essay, the reader should first go through the essay ► [Unimodular Transformations](#) in this encyclopedia.

The program model is a perfect nest  $\mathbf{L}$  of  $m$  sequential loops. An *iteration* of  $\mathbf{L}$  is an instance of the body of  $\mathbf{L}$ . The program consists of a certain set of iterations that are to be executed in a certain sequential order. This execution order imposes a *dependence structure* on the set of iterations, based on how they access different memory locations. A loop in  $\mathbf{L}$  *carries a dependence* if the dependence between two iterations is due to the unfolding of that loop. A loop can *run in parallel* if it carries no dependence.

In this essay, one considers transformations that change  $\mathbf{L}$  into a perfect nest  $\mathbf{L}'$  with the same set of iterations executing in a different sequential order. (The number of loops may differ from  $\mathbf{L}$  to  $\mathbf{L}'$ .) The new nest  $\mathbf{L}'$  is *equivalent* to the old nest  $\mathbf{L}$  (i.e., they give the same results), if whenever an iteration depends on another iteration in  $\mathbf{L}$ , the first iteration is executed after the second in  $\mathbf{L}'$ . If a loop in  $\mathbf{L}'$  carries no dependence, then that loop can run in parallel. The goal is to transform  $\mathbf{L}$  into an equivalent nest  $\mathbf{L}'$ , such that some inner and/or outer loops of  $\mathbf{L}'$  can run in parallel.

Two major loop nest transformations are discussed here: unimodular and echelon. Unimodular transformations have already been introduced in this encyclopedia in the essay under that name. There it is shown by examples how to achieve inner and outer loop parallelization via unimodular transformations. Detailed algorithms for achieving those goals are given in this essay. Echelon transformations are explained after that, and it is pointed out how a combination of both transformations can maximize loop parallelization.

### Mathematical Preliminaries

Lexicographic order and Fourier's method of elimination, as explained in the essay ► [Unimodular Transformations](#), will be used in the following. In this section, some of the elementary definitions and algorithms of matrix theory are stated, customized for integer matrices. From now on, a matrix is an integer matrix, unless explicitly designated to be otherwise.

The transpose of a matrix  $\mathbf{A}$  is denoted by  $\mathbf{A}'$ . The  $m \times m$  unit matrix is denoted by  $\mathcal{I}_m$ . A *submatrix* of a given matrix  $\mathbf{A}$  is the matrix obtained by deleting some rows and columns of  $\mathbf{A}$ . If  $\mathbf{A}$  is an  $m \times p$  matrix and  $\mathbf{B}$  an  $m \times q$  matrix, then the *column-augmented matrix*  $(\mathbf{A}; \mathbf{B})$  is the  $m \times (p + q)$  matrix whose first  $p$  columns are the columns of  $\mathbf{A}$  and the last  $q$  columns are the columns of  $\mathbf{B}$ . Thus,

$$\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 0 & 0 \\ 0 & 5 \end{pmatrix} \text{ and } \mathbf{B} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

$$\text{yield } (\mathbf{A}; \mathbf{B}) = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ 0 & 5 & 0 \end{pmatrix}.$$

A *row-augmented matrix* is defined similarly.

The (*main*) *diagonal* of an  $m \times n$  matrix  $(a_{ij})$  is the vector  $(a_{11}, a_{22}, \dots, a_{pp})$  where  $p = \min(m, n)$ . The *rank* of a matrix  $\mathbf{A}$ , denoted by  $\text{rank}(\mathbf{A})$ , is the maximum number of linearly independent rows (or columns) of  $\mathbf{A}$ .

For a given  $m \times n$  matrix  $\mathbf{A}$ , let  $\ell_i$  denote the column number of the leading (first nonzero) element of row  $i$ . (For a zero row,  $\ell_i$  is undefined.) Then  $\mathbf{A}$  is an *echelon matrix*, if for some integer  $\rho$  in  $0 \leq \rho \leq m$ , the following holds:

1. The rows 1 through  $\rho$  are nonzero rows.
2. The rows  $\rho + 1$  through  $m$  are zero rows.
3. For  $1 \leq i \leq \rho$ , each element in column  $\ell_i$  below row  $i$  is zero.
4.  $\ell_1 < \ell_2 < \dots < \ell_\rho$ .

If there is such a  $\rho$ , then  $\rho = \text{rank}(\mathbf{A})$ . A zero matrix is an echelon matrix for which  $\rho = 0$ . Three nonzero echelon matrices with the values  $\rho = 4, 3, 2$ , respectively, are given below:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}, \begin{pmatrix} 5 & 2 & 3 \\ 0 & -1 & 0 \\ 0 & 0 & 3 \\ 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 5 & 2 & 3 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

For any given matrix, there are three types of *elementary row operations*:

1. *Reversal*: multiply a row by  $-1$ .
2. *Interchange*: interchange two rows.
3. *Skewing*: add an integer multiple of one row to another row.

The *elementary column operations* are defined similarly.

An *elementary matrix* is any matrix obtained from a unit matrix by one elementary row operation. Thus, there are three types of elementary matrices. The elementary matrices can also be derived by column operations. For example, the same matrix is obtained from the  $3 \times 3$  unit matrix if we interchange rows 1 and 3, or interchange columns 1 and 3.

A *unimodular matrix* is a square integer matrix with determinant 1 or  $-1$ . Each elementary matrix is unimodular, and each unimodular matrix can be expressed as the product of a finite number of elementary matrices (Lemma 2.3 in [2]). For a given matrix  $\mathbf{A}$ , performing an elementary row operation is equivalent to forming a product of the form  $\mathbf{EA}$ , where  $\mathbf{E}$  is the corresponding elementary matrix. Applying a finite sequence of row operations to  $\mathbf{A}$  is the same as forming a product of the form  $\mathbf{UA}$ , where  $\mathbf{U}$  is a unimodular matrix.

*Reducing* an  $m \times n$  matrix  $\mathbf{A}$  to *echelon form* means finding an  $m \times m$  unimodular matrix  $\mathbf{U}$  and an  $m \times n$  echelon matrix  $\mathbf{S}$ , such that  $\mathbf{UA} = \mathbf{S}$ . Figure 1 gives an algorithm that reduces any given matrix  $\mathbf{A}$  to echelon form. Note that applying the same row operation to the two matrices  $\mathbf{U}$  and  $\mathbf{S}$  separately is equivalent

to applying that operation to the column-augmented matrix  $(\mathbf{U}; \mathbf{S})$ .

For a given matrix  $\mathbf{A}$ , one may need to find a unimodular matrix  $\mathbf{V}$  and an echelon matrix  $\mathbf{S}$  such that  $\mathbf{A} = \mathbf{VS}$ . This  $\mathbf{V}$  could be the inverse of the unimodular matrix  $\mathbf{U}$  of Algorithm 1. However, if  $\mathbf{U}$  is not needed, Algorithm 1 can be modified to yield directly a unimodular  $\mathbf{V}$  and an echelon  $\mathbf{S}$  such that  $\mathbf{A} = \mathbf{VS}$ . This is the algorithm of Fig. 2. In Algorithm 2, whenever a row operation is applied to  $\mathbf{S}$ , it is followed by a column operation applied to  $\mathbf{V}$ . The row operation is equivalent to forming a product of the form  $\mathbf{ES}$ , where  $\mathbf{E}$  is an elementary matrix. The column operation that follows is equivalent to forming the product  $\mathbf{VE}^{-1}$ . Hence, the relation  $\mathbf{A} = \mathbf{VS}$  remains unchanged since  $\mathbf{VS} = (\mathbf{VE}^{-1})(\mathbf{ES})$ .

Each algorithm can be modified by using extra reversal operations, as needed, so that any given rows of the computed echelon matrix are nonnegative.

The program model for all transformations is a perfect nest of loops  $\mathbf{L} = (L_1, L_2, \dots, L_m)$  shown in Fig. 3. The limits  $p_r$  and  $q_r$  are integer-valued linear functions of  $I_1, I_2, \dots, I_{r-1}$ . All distance vectors are assumed to be uniform. If there is no distance vector  $\mathbf{d}$  with  $\mathbf{d} >_r \mathbf{0}$ , then the loop  $L_r$  carries no dependence and is able to run in parallel.

## Unimodular Transformations

Let  $\mathbf{U}$  denote an  $m \times m$  unimodular matrix. The unimodular transformation of the loop nest  $\mathbf{L}$  induced by  $\mathbf{U}$  is

**Algorithm 1** Given an  $m \times n$  integer matrix  $\mathbf{A}$ , this algorithm finds an  $m \times m$  unimodular matrix  $\mathbf{U}$  and an  $m \times n$  echelon matrix  $\mathbf{S} = (s_{ij})$ , such that  $\mathbf{UA} = \mathbf{S}$ . It starts with  $\mathbf{U} = \mathcal{I}_m$  and  $\mathbf{S} = \mathbf{A}$ , and works on the matrix  $(\mathbf{U}; \mathbf{S})$ . Let  $i_0$  denote the row number in which the last processed column of  $\mathbf{S}$  had a nonzero element.

```

set $\mathbf{U} \leftarrow \mathcal{I}_m$, $\mathbf{S} \leftarrow \mathbf{A}$, $i_0 \leftarrow 0$
do $j = 1, n, 1$
 if there is at least one nonzero s_{ij} with $i_0 < i \leq m$
 then
 set $i_0 \leftarrow i_0 + 1$
 do $i = m, i_0 + 1, -1$
 do while $s_{ij} \neq 0$
 set $\sigma \leftarrow \text{sgn}(s_{i-1,j} * s_{ij})$
 $z \leftarrow \lfloor |s_{i-1,j}| / |s_{ij}| \rfloor$
 subtract σz times row i from row $(i-1)$ in $(\mathbf{U}; \mathbf{S})$
 interchange rows i and $(i-1)$ in $(\mathbf{U}; \mathbf{S})$
```

**Algorithm 2** Given an  $m \times n$  integer matrix  $\mathbf{A}$ , this algorithm finds an  $m \times m$  unimodular matrix  $\mathbf{V}$  and an  $m \times n$  echelon matrix  $\mathbf{S} = (s_{ij})$ , such that  $\mathbf{A} = \mathbf{VS}$ . It starts with  $\mathbf{V} = \mathcal{I}_m$  and  $\mathbf{S} = \mathbf{A}$ , and works on the matrices  $\mathbf{V}$  and  $\mathbf{S}$ . Let  $i_0$  denote the row number in which the last processed column of  $\mathbf{S}$  had a nonzero element.

```

set $\mathbf{V} \leftarrow \mathcal{I}_m$, $\mathbf{S} \leftarrow \mathbf{A}$, $i_0 \leftarrow 0$
do $j = 1, n, 1$
 if there is at least one nonzero s_{ij} with $i_0 < i \leq m$
 then
 set $i_0 \leftarrow i_0 + 1$
 do $i = m, i_0 + 1, -1$
 do while $s_{ij} \neq 0$
 set $\sigma \leftarrow \text{sgn}(s_{i-1,j} * s_{ij})$
 $z \leftarrow \lfloor |s_{i-1,j}| / |s_{ij}| \rfloor$
 subtract σz times row i from row $(i - 1)$ in \mathbf{S}
 add σz times column $(i - 1)$ to column i in \mathbf{V}
 interchange rows i and $(i - 1)$ in \mathbf{S}
 interchange columns i and $(i - 1)$ in \mathbf{V}

```

Loop Nest Parallelization. Fig. 2 Modified Echelon Reduction Algorithm

```

 $L_1:$ do $I_1 = p_1, q_1$
 $L_2:$ do $I_2 = p_2, q_2$
 \vdots :
 $L_m:$ do $I_m = p_m, q_m$
 $H(\mathbf{I})$

```

Loop Nest Parallelization. Fig. 3 Loop Nest  $\mathbf{L}$

```

 $L'_1:$ do $K_1 = \alpha_1, \beta_1$
 $L'_2:$ do $K_2 = \alpha_2, \beta_2$
 \vdots :
 $L'_m:$ do $K_m = \alpha_m, \beta_m$
 $H'(\mathbf{K})$

```

Loop Nest Parallelization. Fig. 4 Loop Nest  $\mathbf{L}'$

a loop nest  $\mathbf{L}'$  of the form shown in Fig. 4, where  $\mathbf{K} = \mathbf{IU}$  and  $H'(\mathbf{K}) = H(\mathbf{KU}^{-1})$ . The transformed program has the same set of iterations as the original program, executing in a different sequential order. The transformation  $\mathbf{L} \Rightarrow \mathbf{L}'$  is valid if and only if  $\mathbf{dU} > \mathbf{0}$  for each distance vector  $\mathbf{d}$  of  $\mathbf{L}$ . In that case, the vectors  $\mathbf{dU}$  are precisely the distance vectors of the new nest  $\mathbf{L}'$ .

### Inner Loop Parallelization

Let  $D$  denote the set of distance vectors of  $\mathbf{L}$ . For the transformation  $\mathbf{L} \Rightarrow \mathbf{L}'$  to be valid, the matrix  $\mathbf{U}$  must be so chosen that  $\mathbf{dU} > \mathbf{0}$  for each  $\mathbf{d} \in D$ . After a valid transformation by  $\mathbf{U}$ , the set of distance vectors of  $\mathbf{L}'$  is

$\{\mathbf{dU} : \mathbf{d} \in D\}$ . A loop  $L'_r$  of  $\mathbf{L}'$  can run in parallel if there is no  $\mathbf{d} \in D$  satisfying  $\mathbf{dU} >_r \mathbf{0}$ . Thus, a unimodular matrix  $\mathbf{U}$  will transform the loop nest  $\mathbf{L}$  into an equivalent loop nest  $\mathbf{L}'$  where the loops  $L'_2, L'_3, \dots, L'_m$  can all run in parallel, if and only if  $\mathbf{U}$  satisfies the condition

$$\mathbf{dU} >_1 \mathbf{0} \quad (\mathbf{d} \in D). \quad (1)$$

The following informal algorithm shows in detail that a unimodular matrix with this property can always be constructed.

**Algorithm 3** Given a perfect nest  $\mathbf{L}$  of  $m$  loops, as shown in Fig. 3, with a set of distance vectors  $D$ , this algorithm finds an  $m \times m$  unimodular matrix  $\mathbf{U} = (u_{rt})$  that transforms  $\mathbf{L}$  into an equivalent loop nest  $\mathbf{L}'$ , such that the inner loops  $L'_2, L'_3, \dots, L'_m$  of  $\mathbf{L}'$  can all run in parallel.

1. If  $D = \emptyset$ , then all loops of  $\mathbf{L}$  can execute in parallel; take for  $\mathbf{U}$  the  $m \times m$  unit matrix and exit. Otherwise, go to step 2.
2. Condition (1) is equivalent to the following system of inequalities

$$d_1 u_1 + d_2 u_2 + \dots + d_m u_m \geq 1 \quad (\mathbf{d} \in D), \quad (2)$$

where  $\mathbf{d} = (d_1, d_2, \dots, d_m)$  and the fixed second subscript of  $u_{r1}$  has been dropped for convenience.

For each  $r$  in  $1 \leq r \leq m$ , define the set

$$D_r = \{\mathbf{d} \in D : \mathbf{d} >_r \mathbf{0}\}.$$

At least one of these sets is nonempty. Since  $d_1 = d_2 = \dots = d_{r-1} = 0$  for each  $\mathbf{d} \in D_r$ , one can break up the system (2) into a sequence of subsystems of the following type:

$$\left. \begin{array}{l} d_m u_m \geq 1 \quad (\mathbf{d} \in D_m) \\ d_{m-1} u_{m-1} + d_m u_m \geq 1 \quad (\mathbf{d} \in D_{m-1}) \\ \vdots \\ d_1 u_1 + d_2 u_2 + \dots + d_{m-1} u_{m-1} + d_m u_m \geq 1 \quad (\mathbf{d} \in D_1). \end{array} \right\}$$

Since  $d_r > 0$  for  $\mathbf{d} \in D_r$ , this sequence can be rewritten as

$$\left. \begin{array}{l} u_m \geq 1/d_m \quad (\mathbf{d} \in D_m) \\ u_{m-1} \geq (1 - d_m u_m)/d_{m-1} \quad (\mathbf{d} \in D_{m-1}) \\ \vdots \\ u_1 \geq (1 - d_2 u_2 - \dots - d_m u_m)/d_1 \quad (\mathbf{d} \in D_1). \end{array} \right\} \quad (3)$$

3. If  $D_m = \emptyset$ , then take  $u_m = 0$ . Otherwise, take  $u_m = 1$ . Suppose  $u_m, u_{m-1}, \dots, u_{r+1}$  have been chosen, where  $1 \leq r < m$ . If  $D_r = \emptyset$ , then take  $u_r = 0$ . Otherwise,  $u_r$  has a set of lower bounds of the form:

$$u_r \geq (1 - d_{r+1} u_{r+1} - d_{r+2} u_{r+2} - \dots - d_m u_m)/d_r \quad (\mathbf{d} \in D_r).$$

Take for  $u_r$  the smallest nonnegative integer that would work:

$$u_r = \left[ \max_{\mathbf{d} \in D_r} \{(1 - d_{r+1} u_{r+1} - d_{r+2} u_{r+2} - \dots - d_m u_m)/d_r\} \right]^+.$$

4. Find  $s$  in  $1 \leq s \leq m$  such that  $u_s$  is the first nonzero element in the sequence:  $u_m, u_{m-1}, \dots, u_1$ . Note that  $u_s = 1$ . Set

$$\mathbf{U} = \begin{pmatrix} u_1 & 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ u_{s-1} & 0 & \cdots & 1 & 0 & \cdots & 0 \\ 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 1 \end{pmatrix},$$

such that row  $s$  of  $\mathbf{U}$  is  $(1, 0, 0, \dots, 0)$ , its first column is  $(u_1, u_2, \dots, u_m)$ , and the submatrix obtained by deleting row  $s$  and the first column of  $\mathbf{U}$  is the  $(m-1) \times (m-1)$  unit matrix. Since  $\det \mathbf{U} = 1$ ,  $\mathbf{U}$  is unimodular.

5. Note that the inverse of  $\mathbf{U}$  is given by

$$\mathbf{U}^{-1} = \begin{pmatrix} 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \\ 1 & \cdots & 0 & -u_1 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & -u_{s-1} & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & 0 & \cdots & 1 \end{pmatrix},$$

such that column  $s$  is  $(1, -u_1, \dots, -u_{s-1}, 0, \dots, 0)$ , row 1 is  $(0, \dots, 0, 1, 0, \dots, 0)$ , and the submatrix obtained by deleting column  $s$  and the first row is the  $(m-1) \times (m-1)$  unit matrix. From the equation  $(I_1, I_2, \dots, I_m) = (K_1, K_2, \dots, K_m)\mathbf{U}^{-1}$ , it follows that

$$\left. \begin{array}{lcl} I_1 & = & K_2 \\ & \vdots & \\ I_{s-1} & = & K_s \\ I_s & = & K_1 - u_1 K_2 - \dots - u_{s-1} K_s \\ I_{s+1} & = & K_{s+1} \\ & \vdots & \\ I_m & = & K_m. \end{array} \right\}$$

In the system of inequalities

$$p_r(I_1, \dots, I_{r-1}) \leq I_r \leq q_r(I_1, \dots, I_{r-1}) \quad (1 \leq r \leq m),$$

substitute for each  $I_r$  the corresponding expression in  $K_1, K_2, \dots, K_m$ . Get the loop limits  $\alpha_r(K_1, \dots, K_{r-1})$  and  $\beta_r(K_1, \dots, K_{r-1})$  of Fig. 4 by

solving this system using Fourier's method of elimination.

6. Now each distance vector  $\mathbf{d}$  of  $\mathbf{L}$  satisfies the condition  $\mathbf{d}\mathbf{U} >_1 \mathbf{0}$ . Hence, the transformed nest  $\mathbf{L}'$  of Fig. 4 created by this matrix  $\mathbf{U}$  is equivalent to the original nest  $\mathbf{L}$ , and the inner loops of  $\mathbf{L}'$  can all execute in parallel.

*Remark 1* There exist infinitely many vectors  $\mathbf{u} = (u_1, u_2, \dots, u_m)$  satisfying (3) and  $\gcd(u_1, u_2, \dots, u_m) = 1$ . For each such vector  $\mathbf{u}$ , there exist infinitely many unimodular matrices with  $\mathbf{u}$  as the first column. (See Sect. 3.3 in [2].) Any such matrix will satisfy the goal of Algorithm 3. Among all possible choices for  $\mathbf{u}$ , an ideal vector is one that minimizes the number of iterations  $(\beta_1 - \alpha_1 + 1)$  of the outermost loop  $L'_1$  that must run sequentially. This poses an optimization problem; see Chap. 3 in [3] for details.

Since  $\mathbf{K} = \mathbf{IU}$  and  $(u_1, u_2, \dots, u_m)$  is the first column of  $\mathbf{U}$ , one has  $K_1 = u_1 I_1 + u_2 I_2 + \dots + u_m I_m$ . An equation of the form  $K_1 = c$ , where  $c$  is a constant, represents a hyperplane in the vector space  $\mathbf{R}^m$  with coordinate axes  $I_1, I_2, \dots, I_m$ . The integral values of  $K_1$  from  $\alpha_1$  to  $\beta_1$  then represent a packet of parallel hyperplanes through the index space of the loop nest  $\mathbf{L}$ , perpendicular to the vector  $\mathbf{u}$ . Geometrically speaking, Algorithm 3 finds a sequence of parallel hyperplanes, such that the planes are taken sequentially, and iterations for all index points on each plane are executed in parallel. This algorithm is derived from Lamport's 1974 CACM paper [8]. The name *Hyperplane Method*, often used to describe it, also comes from [8].

*Example 1* To better understand how Algorithm 3 creates the matrix  $\mathbf{U}$  from a given set of distance vectors  $D$ , consider a loop nest  $\mathbf{L} = (L_1, L_2, L_3, L_4)$  with the set:

$$D = \{(0, 0, 0, 2), (0, 3, 1, -2), (0, 4, -6, 0), (1, -5, 3, 1), (2, 1, 0, 0), (3, 0, -2, 1)\}.$$

The only loop that carries no dependence is  $L_3$ . Hence, only  $L_3$  can run in parallel. The goal is to find a unimodular matrix  $\mathbf{U}$  that will transform  $\mathbf{L}$  into an equivalent loop nest  $\mathbf{L}'$ , where all the inner loops can execute in parallel. The first column  $(u_1, u_2, u_3, u_4)$  of  $\mathbf{U}$  must satisfy the inequality

$$d_1 u_1 + d_2 u_2 + d_3 u_3 + d_4 u_4 \geq 1$$

for each  $(d_1, d_2, d_3, d_4) \in D$ . After simplification, one gets a set of six inequalities:

$$\left. \begin{array}{l} u_4 \geq 1/2 \\ u_2 \geq (1 - u_3 + 2u_4)/3 \\ u_2 \geq (1 + 6u_3)/4 \\ u_1 \geq 1 + 5u_2 - 3u_3 - u_4 \\ u_1 \geq (1 - u_2)/2 \\ u_1 \geq (1 + 2u_3 - u_4)/3. \end{array} \right\}$$

For each  $u_r$ , select the smallest possible nonnegative integral value that will work. So, take  $u_4 = 1$ . Since  $D_3 = \emptyset$  (i.e., there is no distance vector  $\mathbf{d}$  with  $\mathbf{d} >_3 \mathbf{0}$ ), take  $u_3 = 0$ . The constraints on  $u_2$  are then  $u_2 \geq 1$  and  $u_2 \geq 1/4$ . Take  $u_2 = 1$ . Finally, the lower bounds on  $u_1$  are given by  $u_1 \geq 5$ ,  $u_1 \geq 0$ , and  $u_1 \geq 0$ , so that one takes  $u_1 = 5$ . The unimodular matrix  $\mathbf{U}$  as constructed in Step 4 of Algorithm 3 is

$$\mathbf{U} = \begin{pmatrix} 5 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}.$$

Here  $s = 4$ . Row 4 of  $\mathbf{U}$  is  $(1, 0, 0, 0)$ , its first column is  $(u_1, u_2, u_3, u_4) = (5, 1, 0, 1)$ , and the submatrix obtained by deleting the fourth row and the first column is the  $3 \times 3$  unit matrix. The set of distance vectors of the equivalent loop nest  $\mathbf{L}' = (L'_1, L'_2, L'_3, L'_4)$  is the set of vectors  $\mathbf{d}\mathbf{U}$  for  $\mathbf{d} \in D$ , that is, the set

$$\{(2, 0, 0, 0), (1, 0, 3, 1), (4, 0, 4, -6), (1, 1, -5, 3), (11, 2, 1, 0), (16, 3, 0, -2)\}.$$

None of the inner loops  $L'_2, L'_3, L'_4$  carries a dependence, and hence they can all run in parallel.

### Outer Loop Parallelization

For a given  $m \times n$  matrix  $\mathbf{A} = (a_{ij})$ , the rows are denoted by  $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_m$  and the columns by  $\mathbf{a}^1, \mathbf{a}^2, \dots, \mathbf{a}^n$ . Note that the proof of the following theorem uses both the set  $D$  of distance vectors and the distance matrix  $\mathbf{D}$  of  $\mathbf{L}$ , while the previous algorithm used only  $D$ .

**Theorem 1** Consider a nest  $L$  of  $m$  loops. Let  $D$  denote the distance matrix of  $L$  and  $\rho$  the rank of  $D$ . Then there exists a valid unimodular transformation  $L \Rightarrow L'$  such that the outermost  $(m - \rho)$  loops and the innermost  $(\rho - 1)$  loops of  $L'$  can execute in parallel.

*Proof* Let  $D$  denote the set of distance vectors of  $L$ . A unimodular matrix  $U$  will induce a valid transformation  $L \Rightarrow L'$  if and only if  $\mathbf{d}U > \mathbf{0}$  for each  $\mathbf{d} \in D$ . After a valid transformation by  $U$ , the distance vectors of  $L'$  are the vectors  $\mathbf{d}U$ , where  $\mathbf{d} \in D$ . The outermost  $(m - \rho)$  loops and the innermost  $(\rho - 1)$  loops of  $L'$  can execute in parallel, if and only if  $\mathbf{d}U >_r \mathbf{0}$  is false for each  $\mathbf{d} \in D$  when  $r \neq m - \rho + 1$ . Let  $n = m - \rho$ . Then each  $\mathbf{d} \in D$  must satisfy  $\mathbf{d}U >_{n+1} \mathbf{0}$ .

The transpose  $D'$  of  $D$  has  $m$  rows and its columns are the distance vectors of  $L$ . By [Algorithm 1](#), find an  $m \times m$  unimodular matrix  $V$  and an echelon matrix  $S$  such that  $VD' = S$ . ( $S$  and  $D'$  have the same size.) The number of nonzero rows of  $S$  is the rank  $\rho$  of  $D$ , and the number of zero rows is  $n = m - \rho$ . Since the lowest  $n$  rows of  $S$  are zero vectors, it follows that each of the lowest  $n$  rows of  $V$  is orthogonal to each  $\mathbf{d} \in D$ .

Next, find an  $m$ -vector  $u$  by [Algorithm 3](#) such that  $\mathbf{d}u > 0$  for each  $\mathbf{d} \in D$ . Let  $A$  denote the  $m \times (n + 1)$  matrix where the columns  $a^1, a^2, \dots, a^n$  are the lowest  $n$  rows of  $V$ , and  $a^{n+1} = u$ . Then each  $\mathbf{d} \in D$  satisfies the equations:

$$\mathbf{d}a^1 = 0, \mathbf{d}a^2 = 0, \dots, \mathbf{d}a^n = 0, \mathbf{d}a^{n+1} > 0. \quad (4)$$

By [Algorithm 2](#), find an  $m \times m$  unimodular matrix  $U$  and an  $m \times (n + 1)$  echelon matrix

$$T = \begin{pmatrix} t_{11} & t_{12} & t_{13} & \cdots & t_{1n} & t_{1,n+1} \\ 0 & t_{22} & t_{23} & \cdots & t_{2n} & t_{2,n+1} \\ 0 & 0 & t_{33} & \cdots & t_{3n} & t_{3,n+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & t_{nn} & t_{n,n+1} \\ 0 & 0 & 0 & \cdots & 0 & t_{n+1,n+1} \\ 0 & 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & 0 \end{pmatrix},$$

such that  $A = UT$  and the diagonal element  $t_{n+1,n+1}$  is nonnegative. The unimodular transformation  $L \Rightarrow L'$  induced by  $U$  satisfies the theorem.

After writing the relation  $A = UT$  in the form

$$(a^1, a^2, \dots, a^n, a^{n+1}) = (u^1, u^2, \dots, u^n, u^{n+1}, u^{n+2}, \dots, u^m) \cdot T,$$

it becomes clear that

$$\left. \begin{aligned} a^1 &= t_{11}u^1 \\ a^2 &= t_{12}u^1 + t_{22}u^2 \\ &\vdots \\ a^n &= t_{1n}u^1 + t_{2n}u^2 + \cdots + t_{nn}u^n \\ a^{n+1} &= t_{1,n+1}u^1 + \cdots + t_{n,n+1}u^n + t_{n+1,n+1}u^{n+1}. \end{aligned} \right\} \quad (5)$$

Since  $a^1, a^2, \dots, a^n$  form distinct rows of a unimodular matrix  $V$ , they are linearly independent. Hence, the diagonal elements  $t_{11}, t_{22}, \dots, t_{nn}$  of  $T$  must be nonzero. Multiply the equations of (5) by any  $\mathbf{d} \in D$  and use (4) to get

$$\left. \begin{aligned} t_{11}(\mathbf{d}u^1) &= 0 \\ t_{12}(\mathbf{d}u^1) + t_{22}(\mathbf{d}u^2) &= 0 \\ &\vdots \\ t_{1n}(\mathbf{d}u^1) + t_{2n}(\mathbf{d}u^2) + \cdots + t_{nn}(\mathbf{d}u^n) &= 0 \\ t_{1,n+1}(\mathbf{d}u^1) + \cdots + t_{n,n+1}(\mathbf{d}u^n) + t_{n+1,n+1}(\mathbf{d}u^{n+1}) &> 0. \end{aligned} \right\}$$

Since  $t_{11}, t_{22}, \dots, t_{nn}$  are all nonzero and  $t_{n+1,n+1} \geq 0$ , this implies

$$\mathbf{d}u^1 = 0, \mathbf{d}u^2 = 0, \dots, \mathbf{d}u^n = 0, \mathbf{d}u^{n+1} > 0,$$

that is,  $\mathbf{d}U >_{n+1} \mathbf{0}$ , for each  $\mathbf{d} \in D$ . This completes the proof. ■

*Example 2* Consider a loop nest  $L = (L_1, L_2, L_3)$  whose distance matrix  $D$  and its transpose  $D'$  are given by

$$D = \begin{pmatrix} 6 & 4 & 2 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{pmatrix} \quad \text{and} \quad D' = \begin{pmatrix} 6 & 0 & 1 \\ 4 & 1 & 0 \\ 2 & -1 & 1 \end{pmatrix}.$$

Here  $m = 3$ . By [Algorithm 1](#), find two matrices

$$\mathbf{V} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & -2 \\ 1 & -1 & -1 \end{pmatrix} \quad \text{and} \quad \mathbf{S} = \begin{pmatrix} 2 & -1 & 1 \\ 0 & 3 & -2 \\ 0 & 0 & 0 \end{pmatrix},$$

such that  $\mathbf{V}$  is unimodular,  $\mathbf{S}$  is echelon, and  $\mathbf{VD}' = \mathbf{S}$ . Then  $\rho = 2$  and  $n = 1$ . Since the bottom row of  $\mathbf{S}$  is zero, it follows that the bottom row  $(1, -1, -1)$  of  $\mathbf{V}$  is orthogonal to each column of  $\mathbf{D}'$ , that is, to each distance vector  $\mathbf{d}$  of  $\mathbf{L}$ . This will be the first column of a  $2 \times 3$  matrix  $\mathbf{A}$  that is being constructed.

Next, one needs a vector  $\mathbf{u}$  such that  $\mathbf{du} > 0$ , or equivalently,  $\mathbf{du} \geq 1$  for each distance vector  $\mathbf{d}$ . The set of inequalities to be satisfied is:

$$\left. \begin{array}{l} 6u_1 + 4u_2 + 2u_3 \geq 1 \\ u_2 - u_3 \geq 1 \\ u_1 + u_3 \geq 1 \end{array} \right\}$$

or

$$\left. \begin{array}{l} u_2 \geq 1 + u_3 \\ u_1 \geq (1 - 4u_2 - 2u_3)/6 \\ u_1 \geq 1 - u_3. \end{array} \right\}$$

[Algorithm 3](#) returns the vector  $\mathbf{u} = (1, 1, 0)$ . This will be the second column of the matrix  $\mathbf{A}$ . Thus,

$$\mathbf{A} = \begin{pmatrix} 1 & 1 \\ -1 & 1 \\ -1 & 0 \end{pmatrix}.$$

By [Algorithm 2](#), find two matrices

$$\mathbf{U} = \begin{pmatrix} -1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \quad \text{and} \quad \mathbf{T} = \begin{pmatrix} -1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix},$$

such that  $\mathbf{U}$  is unimodular,  $\mathbf{T}$  is echelon (with a nonnegative diagonal element on second row), and  $\mathbf{A} = \mathbf{UT}$ . Since

$$(6, 4, 2)\mathbf{U} = (0, 10, 6), \quad (0, 1, -1)\mathbf{U} = (0, 1, 0),$$

$$\text{and } (1, 0, 1)\mathbf{U} = (0, 1, 1)$$

are all positive vectors, the transformation  $(L_1, L_2, L_3) \Rightarrow (L'_1, L'_2, L'_3)$  induced by  $\mathbf{U}$  is valid. The distance vectors of  $(L'_1, L'_2, L'_3)$  are  $(0, 10, 6)$ ,  $(0, 1, 0)$  and  $(0, 1, 1)$ . Since the loops  $L'_1$  and  $L'_3$  do not carry a dependence, they can run in parallel.

### Echelon Transformation

Some background needs to be prepared before echelon transformations can be defined. Let  $N$  denote the number of distance vectors of the loop nest  $\mathbf{L}$  of [Fig. 3](#), and  $\mathbf{D}$  its distance matrix. Apply [Algorithm 2](#) to the  $N \times m$  matrix  $\mathbf{D}$  to find an  $N \times N$  unimodular matrix  $\mathbf{V}$  and an  $N \times m$  echelon matrix  $\mathbf{S} = (s_{tr})$  with nonnegative rows, such that  $\mathbf{D} = \mathbf{VS}$ . Let  $\rho$  denote the number of positive rows of  $\mathbf{S}$ , so that  $\rho = \text{rank}(\mathbf{D})$ .

The top  $\rho$  rows of  $\mathbf{S}$  are positive rows and the bottom  $(N - \rho)$  rows are zero rows. Let  $\widehat{\mathbf{S}}$  denote the  $\rho \times m$  submatrix of  $\mathbf{S}$  consisting of the positive rows, and  $\widehat{\mathbf{V}}$  the  $N \times \rho$  submatrix of  $\mathbf{V}$  consisting of the leftmost  $\rho$  columns. Then,  $\mathbf{D}$  can be written in the form:

$$\mathbf{D} = \widehat{\mathbf{V}} \widehat{\mathbf{S}}. \quad (6)$$

**Lemma 1** *The rows of  $\widehat{\mathbf{V}}$  are positive vectors.*

*Proof* Let  $\mathbf{v}$  denote any row of  $\widehat{\mathbf{V}}$ . Then  $\mathbf{v}\widehat{\mathbf{S}} = \mathbf{d}$ , where  $\mathbf{d}$  is a distance vector. Since  $\mathbf{d}$  must be positive,  $\mathbf{v}$  cannot be the zero vector. Hence, it has the form  $\mathbf{v} = (0, \dots, 0, v_t, \dots, v_\rho)$ , where  $1 \leq t \leq \rho$  and  $v_t \neq 0$ . Let  $\ell = \ell_t$  denote the column number of the leading (first nonzero) element on row  $t$  of  $\widehat{\mathbf{S}}$ . Then,  $s_{t\ell} > 0$ , column  $\ell$  of  $\widehat{\mathbf{S}}$  has the form  $(s_{1\ell}, s_{2\ell}, \dots, s_{t\ell}, 0, \dots, 0)$ , and each column  $j$  for  $1 \leq j < \ell$  has the form  $(s_{1j}, s_{2j}, \dots, s_{t-1,j}, 0, \dots, 0)$ . Hence, the product  $\mathbf{v}\widehat{\mathbf{S}}$  has the form  $(0, \dots, 0, v_t s_{t\ell}, \dots)$ . Now,  $v_t s_{t\ell} \neq 0$  since  $v_t \neq 0$  and  $s_{t\ell} > 0$ . However, being the leading element of a distance vector,  $v_t s_{t\ell}$  must be positive. This means  $v_t > 0$ , since  $s_{t\ell} > 0$ . Thus,  $\mathbf{v}$  is a positive vector.  $\square$

**Lemma 2** *For  $1 \leq t \leq \rho$ , let  $\ell_t$  denote the column number of the leading element on row  $t$  of  $\widehat{\mathbf{S}}$ . For each  $\mathbf{I} \in \mathbf{Z}^m$ , there exists a unique  $\mathbf{Y} \in \mathbf{Z}^m$  and a unique  $\mathbf{K} \in \mathbf{Z}^\rho$ , such that*

$$0 \leq Y_{\ell_t} \leq s_{t\ell_t} - 1 \quad (1 \leq t \leq \rho) \quad (7)$$

and

$$\mathbf{I} = \mathbf{Y} + \mathbf{KS}. \quad (8)$$

*Proof* Note that the leading elements  $s_{t\ell_t}$  are all positive by construction. Let  $\mathbf{I} = (I_1, I_2, \dots, I_m) \in \mathbf{Z}^m$ . Define  $\mathbf{K} = (K_1, K_2, \dots, K_\rho) \in \mathbf{Z}^\rho$  by

$$\begin{aligned} K_1 &= \lfloor I_{\ell_1} / s_{1\ell_1} \rfloor \\ K_2 &= \lfloor (I_{\ell_2} - s_{1\ell_1} K_1) / s_{2\ell_2} \rfloor \\ &\vdots \\ K_\rho &= \lfloor (I_{\ell_\rho} - s_{1\ell_1} K_1 - s_{2\ell_2} K_2 - \dots - s_{\rho-1,\ell_{\rho-1}} K_{\rho-1}) / s_{\rho\ell_\rho} \rfloor. \end{aligned}$$

After defining  $\mathbf{K}$ , define  $\mathbf{Y} \in \mathbf{Z}^m$  by  $\mathbf{Y} = \mathbf{I} - \mathbf{KS}$ . Then  $\mathbf{Y}$  and  $\mathbf{K}$  are well defined and (8) holds.

For any real number  $x$ , one has  $0 \leq x - \lfloor x \rfloor < 1$ . If  $I$  and  $s$  are integers with  $s > 0$ , then  $0 \leq I/s - \lfloor I/s \rfloor < 1$  implies  $0 \leq I - s\lfloor I/s \rfloor \leq s - 1$ . Hence, if  $I = Y + s\lfloor I/s \rfloor$ , then  $0 \leq Y \leq s - 1$ . Since (8) implies

$$\begin{aligned} I_{\ell_1} &= Y_{\ell_1} + s_{1\ell_1} K_1 \\ I_{\ell_2} - s_{1\ell_1} K_1 &= Y_{\ell_2} + s_{2\ell_2} K_2 \\ &\vdots \\ I_{\ell_\rho} - s_{1\ell_1} K_1 - s_{2\ell_2} K_2 - \dots - s_{\rho-1,\ell_{\rho-1}} K_{\rho-1} &= Y_{\ell_\rho} + s_{\rho\ell_\rho} K_\rho, \end{aligned}$$

it follows from the definition of  $(K_1, K_2, \dots, K_\rho)$  that  $Y_{\ell_1}, Y_{\ell_2}, \dots, Y_{\ell_\rho}$  satisfy (7).  $\square$

The index variables  $I_1, I_2, \dots, I_m$  of the loop nest of Fig. 3 satisfy the constraints:

$$p_r(I_1, I_2, \dots, I_{r-1}) \leq I_r \leq q_r(I_1, I_2, \dots, I_{r-1}) \quad (1 \leq r \leq m).$$

For  $I_1, I_2, \dots, I_m$  in these inequalities, substitute the corresponding expressions in  $Y_1, Y_2, \dots, Y_m, K_1, K_2, \dots, K_\rho$  from (8). Add (7) to that system to get a system of inequalities in  $Y_1, Y_2, \dots, Y_m, K_1, K_2, \dots, K_\rho$ . Apply Fourier's method to the combined system and eliminate the variables

(in this order) to get bounds of the form:

$$\begin{aligned} \alpha_1 &\leq Y_1 \leq \beta_1 \\ \alpha_2(Y_1) &\leq Y_2 \leq \beta_2(Y_1) \\ &\vdots \\ \alpha_m(Y_1, Y_2, \dots, Y_{m-1}) &\leq Y_m \leq \beta_m(Y_1, Y_2, \dots, Y_{m-1}) \\ \alpha_{m+1}(Y) &\leq K_1 \leq \beta_{m+1}(Y) \\ \alpha_{m+2}(Y, K_1) &\leq K_2 \leq \beta_{m+2}(Y, K_1) \\ &\vdots \\ \alpha_{m+\rho}(Y, K_1, K_2, \dots, K_{\rho-1}) &\leq K_\rho \leq \beta_{m+\rho}(Y, K_1, K_2, \dots, K_{\rho-1}), \end{aligned}$$

where  $\mathbf{Y} = (Y_1, Y_2, \dots, Y_m)$ . For each index point  $\mathbf{I}$  of  $\mathbf{L}$ , there is a unique  $(m + \rho)$ -vector  $(\mathbf{Y}, \mathbf{K})$  satisfying these constraints, and conversely. The nest  $\mathbf{L}'$  of  $(m + \rho)$  loops shown in Fig. 5, where  $\mathbf{I} = \mathbf{Y} + \mathbf{KS}$  and  $H'(\mathbf{Y}, \mathbf{K}) = H(\mathbf{I})$ , has the same set of iterations as  $\mathbf{L}$ , but the order of execution is different. The transformation  $\mathbf{L} \Rightarrow \mathbf{L}'$  is called the *echelon transformation* of the loop nest  $\mathbf{L}$ .

Two iterations  $H(\mathbf{i})$  and  $H(\mathbf{j})$  in  $\mathbf{L}$  become iterations  $H'(\mathbf{y}, \mathbf{k})$  and  $H'(\mathbf{z}, \mathbf{l})$ , respectively, in  $\mathbf{L}'$ , where  $\mathbf{i} = \mathbf{y} + \mathbf{kS}$  and  $\mathbf{j} = \mathbf{z} + \mathbf{lS}$ . The transformed nest  $\mathbf{L}'$  is *equivalent* to the original nest  $\mathbf{L}$ , if whenever  $H(\mathbf{j})$  depends on  $H(\mathbf{i})$  in  $\mathbf{L}$ ,  $H'(\mathbf{y}, \mathbf{k})$  precedes  $H'(\mathbf{z}, \mathbf{l})$  in  $\mathbf{L}'$ , that is,  $(\mathbf{y}, \mathbf{k}) \prec (\mathbf{z}, \mathbf{l})$ .

**Theorem 2** *The loop nest  $\mathbf{L}'$  obtained from a nest  $\mathbf{L}$  by echelon transformation is equivalent to  $\mathbf{L}$ , and the  $m$  outermost loops of  $\mathbf{L}'$  can run in parallel.*

*Proof* To prove the equivalence of  $\mathbf{L}'$  to  $\mathbf{L}$ , consider two iterations  $H(\mathbf{i})$  and  $H(\mathbf{j})$  of  $\mathbf{L}$ , such that  $H(\mathbf{j})$  depends on  $H(\mathbf{i})$ . Let  $(\mathbf{y}, \mathbf{k})$  denote the value of  $(\mathbf{Y}, \mathbf{K})$  corresponding to the value  $\mathbf{i}$  of  $\mathbf{I}$ , and  $(\mathbf{z}, \mathbf{l})$  the value corresponding to  $\mathbf{j}$ . Since  $\mathbf{d} = \mathbf{j} - \mathbf{i}$  is a distance vector of  $\mathbf{L}$ , it follows from (6) that  $\mathbf{d} = \mathbf{v}\widehat{\mathbf{S}}$  for some row  $\mathbf{v}$  of  $\widehat{\mathbf{V}}$ .

$$\begin{aligned} L'_1: & \text{ do } Y_1 = \alpha_1, \beta_1 \\ & \vdots \\ L'_m: & \text{ do } Y_m = \alpha_m, \beta_m \\ L'_{m+1}: & \text{ do } K_1 = \alpha_{m+1}, \beta_{m+1} \\ & \vdots \\ L'_{m+\rho}: & \text{ do } K_\rho = \alpha_{m+\rho}, \beta_{m+\rho} \\ & H'(\mathbf{Y}, \mathbf{K}) \end{aligned}$$

$$K_\rho, K_{\rho-1}, \dots, K_1, Y_m, Y_{m-1}, \dots, Y_1$$

Loop Nest Parallelization. Fig. 5 Echelon Transformation

Then

$$\mathbf{j} = \mathbf{i} + \mathbf{d} = \mathbf{y} + \mathbf{k}\widehat{\mathbf{S}} + \mathbf{v}\widehat{\mathbf{S}} = \mathbf{y} + (\mathbf{k} + \mathbf{v})\widehat{\mathbf{S}}.$$

Since  $\mathbf{y}$  satisfies (7),  $(\mathbf{y}, \mathbf{k} + \mathbf{v})$  is the image of  $\mathbf{j}$  under the mapping  $\mathbf{I} \mapsto (\mathbf{Y}, \mathbf{K})$ . But the image of  $\mathbf{j}$  is also  $(\mathbf{z}, \mathbf{l})$  by assumption. Since this mapping is well defined, it follows that  $\mathbf{z} = \mathbf{y}$  and  $\mathbf{l} = \mathbf{k} + \mathbf{v}$ . Hence,

$$(\mathbf{z}, \mathbf{l}) - (\mathbf{y}, \mathbf{k}) = (\mathbf{z} - \mathbf{y}, \mathbf{l} - \mathbf{k}) = (\mathbf{0}, \mathbf{v}).$$

Since  $\mathbf{v}$  is positive by Lemma 2, this implies  $(\mathbf{y}, \mathbf{k}) \prec (\mathbf{z}, \mathbf{l})$ . Hence,  $\mathbf{L}'$  is equivalent to  $\mathbf{L}$  by definition.

It is clear from the above discussion that the distance vectors of the nest  $\mathbf{L}'$  are of the form  $(\mathbf{0}, \mathbf{v})$ , where  $\mathbf{v}$  is a row of  $\widehat{\mathbf{V}}$ . This means the  $m$  outermost  $Y$ -loops carry no dependence, and hence they can run in parallel.  $\square$

**Corollary 1** *In  $\mathbf{L}'$ , the distance matrix of the innermost nest of  $K$ -loops is  $\widehat{\mathbf{V}}$ .*

*Example 3* Consider the loop nest  $\mathbf{L}$ :

```
L1 : do I1 = 1,100
L2 : do I2 = 1,200
L3 : do I3 = I1,300
 X(I1,I2,I3) = X(I1-1,I2-3,I3) +
 X(I1-2,I2-10,I3) + X(I1-1,I2+1,I3)
```

whose distance matrix is

$$\mathbf{D} = \begin{pmatrix} 1 & 3 & 0 \\ 2 & 10 & 0 \\ 1 & -1 & 0 \end{pmatrix}.$$

By Algorithm 2, find two matrices

$$\mathbf{V} = \begin{pmatrix} 1 & 1 & 0 \\ 2 & 3 & 1 \\ 1 & 0 & 0 \end{pmatrix} \quad \text{and} \quad \mathbf{S} = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 0 \end{pmatrix},$$

such that  $\mathbf{V}$  is unimodular,  $\mathbf{S}$  is echelon, and  $\mathbf{D} = \mathbf{VS}$ . The leading elements of nonzero rows of  $\mathbf{S}$  are positive. It is clear that  $\rho = \text{rank}(\mathbf{D}) = \text{rank}(\mathbf{S}) = 2$ . The submatrix  $\widehat{\mathbf{S}}$  of  $\mathbf{S}$  consisting of its nonzero rows is given by

$$\widehat{\mathbf{S}} = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 4 & 0 \end{pmatrix}.$$

Here  $s_{1\ell_1} = 1$  and  $s_{2\ell_2} = 4$ . Define a mapping  $(I_1, I_2, I_3) \mapsto (Y_1, Y_2, Y_3, K_1, K_2)$  of the index space of  $\mathbf{L}$  into  $\mathbf{Z}^5$  by the equation

$$(I_1, I_2, I_3) = (Y_1, Y_2, Y_3) + (K_1, K_2) \cdot \begin{pmatrix} 1 & -1 & 0 \\ 0 & 4 & 0 \end{pmatrix}, \quad (9)$$

and the constraints

$$0 \leq Y_1 \leq 0 \quad \text{and} \quad 0 \leq Y_2 \leq 3. \quad (10)$$

Equation (9) is equivalent to the system:

$$\left. \begin{array}{lcl} I_1 & = & Y_1 + K_1 \\ I_2 & = & Y_2 - K_1 + 4K_2 \\ I_3 & = & Y_3. \end{array} \right\} \quad (11)$$

Substituting for  $I_1, I_2, I_3$  from (11) into the constraints defining the loop limits of  $\mathbf{L}$ , one gets the following set of inequalities:

$$\left. \begin{array}{lcl} 1 & \leq & Y_1 + K_1 & \leq & 100 \\ 1 & \leq & Y_2 - K_1 + 4K_2 & \leq & 200 \\ Y_1 + K_1 & \leq & Y_3 & \leq & 300. \end{array} \right\} \quad (12)$$

Eliminate the variables  $K_2, K_1, Y_3, Y_2, Y_1$  from (10) and (12) by Fourier's method:

$$\left. \begin{array}{lcl} \lceil (1 - Y_2 + K_1)/4 \rceil & \leq & K_2 & \leq & \lfloor (200 - Y_2 + K_1)/4 \rfloor \\ 1 - Y_1 & \leq & K_1 & \leq & \min(100 - Y_1, Y_3 - Y_1) \\ 1 & \leq & Y_3 & \leq & 300 \\ 0 & \leq & Y_2 & \leq & 3 \\ 0 & \leq & Y_1 & \leq & 0. \end{array} \right\}$$

Using the fact that  $Y_1 = 0$ , simplify expressions and get the following loop nest  $\mathbf{L}'$  equivalent to  $\mathbf{L}$ :

```
L2 : do Y2 = 0,3
L3 : do Y3 = 1,300
L4 : do K1 = 1, min(100, Y3)
L5 : do K2 = \lceil (1 - Y2 + K1)/4 \rceil,
 \lfloor (200 - Y2 + K1)/4 \rfloor
```

$$\begin{aligned} X(K_1, Y_2 - K_1 + 4K_2, Y_3) = & \\ & X(K_1 - 1, Y_2 - K_1 + 4K_2 - 3, Y_3) + \\ & X(K_1 - 2, Y_2 - K_1 + 4K_2 - 10, Y_3) + \\ & X(K_1 - 1, Y_2 - K_1 + 4K_2 + 1, Y_3) \end{aligned}$$

In  $L'$ , the  $Y_2$  and the  $Y_3$  loops can run in parallel. (The  $Y_1$ -loop with a single iteration has been omitted.)

The distance matrix of the nest of  $K$ -loops is the submatrix consisting of the two leftmost columns of  $V$ , that is, the matrix:

$$\widehat{V} = \begin{pmatrix} 1 & 1 \\ 2 & 3 \\ 1 & 0 \end{pmatrix}.$$

Since the  $K_2$ -loop carries no dependence, it can run in parallel.

*Remark 2* When  $\rho = m$ , no valid unimodular transformation can give a parallel outer loop (see Corollary 1 to Theorem 3.8 in [3]). A simple example would be a nest of two loops with two distance vectors  $(2, 0)$  and  $(0, 3)$ , for which  $\rho = 2 = m$ . However, echelon transformation of this nest will yield a parallel  $Y_1$ -loop with 2 iterations and a parallel  $Y_2$ -loop with 3 iterations.

If both transformations are available, the best strategy would be to apply the echelon transformation first to get a nest of  $m$  parallel outermost  $Y$ -loops and  $\rho$  sequential innermost  $K$ -loops. Then apply [Algorithm 3](#) to the nest of  $K$ -loops to get one sequential outermost loop followed by  $(\rho - 1)$  parallel inner loops.

## Related Entries

- [Code Generation](#)
- [Parallelism Detection in Nested Loops, Optimal](#)
- [Parallelization, Automatic](#)
- [Unimodular Transformations](#)

## Bibliographic Notes and Further Reading

*Unimodular Transformation.* Research on this topic goes back to Leslie Lamport's paper in 1974 [8], although he did not use this particular term. This essay is based on the theory of unimodular transformations as developed in the author's books on loop transformations [2, 3]. The general theory grew out of the theory of unimodular transformations of double loops described in [1]. (Watch for some notational differences between these references and the current essay.) The paper by Michael Wolf and Monica Lam [13] covers many useful aspects of unimodular transformations. See also

Michael Dowling [5], Erik H. D'Hollander [4], and François Irigoin and Rémi Triolet [6, 7].

Consider a sequential loop nest where each iteration (other than the first) depends on the previous one. The Hyperplane method would still transform it into a nest where the outermost loop is sequential and all inner loops are parallel. There is no great surprise here. It simply means that each inner loop in the new nest will have a single iteration. (Such a loop can run in parallel according to the definition given.) See [13] for the case when there are too many distance vectors.

*Echelon Transformation.* In his PhD thesis [9], David Padua developed the greatest common divisor method for finding a partition of the index space. Peir and Cytron [10] described a partition based on minimum dependence distances. Shang and Fortes [12] offered an algorithm for finding a maximal independent partition. D'Hollander's paper [4] builds on and incorporates these approaches. The general theory of echelon transformations presented here is influenced by these works. See also Polychronopoulos [11].

See [3] for details on unimodular, echelon, and other transformations used in loop nest parallelization, and more references.

## Bibliography

1. Banerjee U (1990) Unimodular transformations of double loops. In: Proceedings of the third workshop on languages and compilers for parallel computing, Irvine, 1–3 August 1990. Available as Nicolau A, Gelernter D, Gross T, Padua D (eds) (1991) Advances in languages and compilers for parallel computing. MIT, Cambridge, pp 192–219
2. Banerjee U (1993) Loop transformations for restructuring compilers: the foundations. Kluwer Academic, Norwell
3. Banerjee U (1994) Loop transformations for restructuring compilers: loop parallelization. Kluwer Academic, Norwell
4. D'Hollander EH (July 1992) Partitioning and labeling of loops by unimodular transformations. IEEE Trans Parallel Distrib Syst 3(4):465–476
5. Dowling ML (Dec 1990) Optimal code parallelization using unimodular transformations. Parallel Comput 16(2–3):157–171
6. Irigoin F, Triolet R (Jan 1988) Supernode partitioning. In: Proceedings of 15th annual ACM SIGACT-SIGPLAN symposium on principles of programming languages, San Diego, pp 319–329
7. Irigoin F, Triolet R (1989) Dependence approximation and global parallel code generation for nested loops. Parallel Distrib Algorithms (Cosnard M et al. eds), Elsevier (North-Holland), New York, pp 297–308

8. Lamport L (Feb 1974) The parallel execution of DO loops. *Commun ACM* 17(2):83–93
9. Padua DA (Nov 1979) Multiprocessors: discussions of some theoretical and practical problems. PhD thesis, Report 79-990, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana
10. Peir J-K, Cytron R (Aug 1989) Minimum distance: a method for partitioning recurrences for multiprocessors. *IEEE Trans Comput C-38(8)*:1203–1211
11. Polychronopoulos CD (Aug 1988) Compiler optimizations for enhancing parallelism and their impact on architecture design. *IEEE Trans Comput C-37(8)*:991–1004
12. Shang W, Fortes JAB (June 1991) Time optimal linear schedules for algorithms with uniform dependences. *IEEE Trans Comput* 40(6):723–742
13. Wolf ME, Lam MS (Oct 1991) A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans Parallel Distrib Syst* 2(4):452–471

## Loop Tiling

► [Tiling](#)

## Loops, Parallel

ROLAND WISMÜLLER

University of Siegen, Siegen, Germany

### Synonyms

[Doall loops](#); [Forall loops](#)

### Definition

Parallel loops are one of the most widely used concepts to express parallelism in parallel languages and libraries. In general, a parallel loop is a loop whose iterations are executed at least partially concurrently by several threads or processes. There are several different kinds of parallel loops with different semantics, which will be discussed below. The most prominent kind is the Doall loop, where the iterations are completely independent.

### Discussion

#### Introduction

In a task parallel program, where the execution of different pieces of code is distributed to parallel processors,

there are two principal ways of specifying parallel activities. The first one is to specify several *different* code regions, i.e., tasks, which should be executed in parallel. This construct is typically called *parallel regions* or *parallel case*. It offers only a very limited scalability, since the maximum degree of parallelism is determined by the number of regions. The second way is to use *parallel loops*. In a parallel loop, the parallel processors execute the *same* code region, namely, the loop body, but with different data. Thus, parallel loops are a special kind of SPMD programming. Typically, parallel loops are used within a shared memory programming model, for example, OpenMP and Intel's Threading Building Blocks. However, they have also been included in some distributed memory programming models, for example, Occam.

In a sequential loop, the loop body is executed for each element in the loop's index domain in a fixed order implied by the domain. For example, in

```
for (i=0; i<N; i++)
 s[i] = sin(PI*i/N);
```

the assignments to *s* occur in the order *s*[0], *s*[1], ..., *s*[*N*], although there is obviously no particular reason which mandates this order of the loop iterations. In a parallel loop, the restrictions on the ordering of the loop iterations are relaxed, which allows the iterations to execute – at least partially – in parallel. In the example, all iterations can execute in parallel, since they are completely independent of each other.

#### Types of Parallel Loops

Today's parallel programming interfaces offer different constructs for parallel loops, which typically fall into one of the following classes which are described by Polychronopoulos [7] and Wolfe [10]: First, there are parallel loops which can be viewed as sequential loops extended by additional properties. A *Doall loop* assures that its iterations can be executed completely independently, while a *Doacross loop* may contain forward dependences. These loops can also be executed sequentially without changing their semantics. In addition, some parallel languages offer a *Forall loop*; however, the term is used with a nonuniform meaning. For example, the Forall loop in Vienna Fortran [3] is actually a Doall loop, while in High Performance Fortran (HPF)

and Fortran95, it has a special nonsequential execution semantics, which is described below.

### Doall Loops

In a Doall loop, every iteration can be executed independently from any other iteration. This means that the iterations can be executed in any sequential order or concurrently.

More formally, a Doall loop is not allowed to contain loop carried dependences. This means that when  $W(i)$  denotes the set of variables the loop's body writes to in some iteration  $i$  and  $R(i)$  denotes the set of variables read in iteration  $i$  the following conditions, often called Bernstein's conditions, must hold for any two different iterations  $i_1$  and  $i_2$  of the loop:

$$\begin{aligned} W(i_1) \cap W(i_2) &= \emptyset \text{ and} \\ R(i_1) \cap W(i_2) &= \emptyset \text{ and } W(i_1) \cap R(i_2) = \emptyset \end{aligned}$$

In this context, the term variable means either a scalar variable or an element of an array or some other structured data type. As an example, consider the following loops:

```
for (i=0; i<N; i++) {
 b[i] = sin(PI*i/N);
 a[i] = a[i] + b[i];
}

for (i=0; i<N; i++) {
 b[i] = sin(PI*i/N);
 a[i] = a[i-1] + b[i];
}
```

Here, the first loop is a Doall loop, while the second one is not. For the first loop,  $W(i) = \{a[i], b[i]\}$  and  $R(i) = \{a[i], b[i], i\}$ . Thus, for different iterations  $i_1$  and  $i_2$ , the write and read sets never overlap. On the other hand, for the second loop,  $W(i) = \{a[i], b[i]\}$  and  $R(i) = \{a[i-1], b[i], i\}$  which leads to  $W(i_1) \cap R(i_2) = \{a[i_1]\}$  for  $i_2 = i_1 + 1$ .

As an additional requirement, the iteration domain of a Doall loop must be fixed when the loop is entered, that is, the loop body is neither allowed to exit the loop prematurely nor to change the loop variable or its upper bound.

When a Doall loop is to be executed by a fixed number of processors (or threads), the iterations can be arbitrarily scheduled to the processors without changing the loop's computational results. However, the scheduling can have a large impact on the performance. With today's cache-based memory systems, it is beneficial if each processor executes a contiguous block of loop iterations since this will maximize the locality of the memory accesses on each processor. For example, when a loop iterating from 1 to 100 is executed by four processors, the first one would execute iterations 1–25, the second one 26–50, and so on. This situation is visualized in Fig. 1.

The blockwise scheduling works well, however, only when all iteration have approximately the same execution time. When the execution time differs largely between iterations, blockwise scheduling can lead to a poor performance. A typical example for this situation is a triangular loop nest, where the outer loop is parallel:

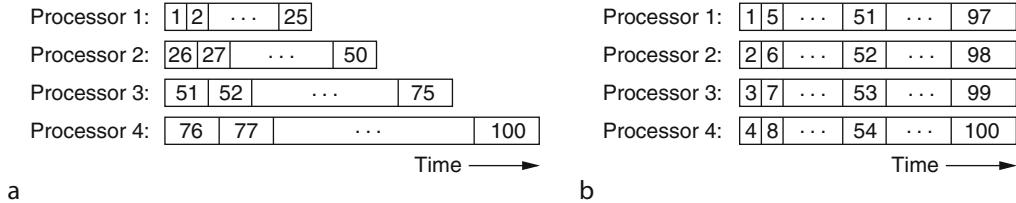
```
for (i=1; i<=100; i++)
 for (j=1; j<=i; j++)
 a[i][j] = some_computation(i, j);
```

With a blockwise scheduling, processor 4 gets the largest amount of work. Since there is typically a global synchronization (barrier) between all processors at the end of a parallel loop, the other three processors will have to wait, which results in a load imbalance. In such cases, a cyclic schedule, where single iterations are assigned to processors in a round-robin fashion offers a better load balancing, as it is shown in Fig. 2. However, the cache performance of such an execution may be poor. As a compromise between cache performance and load balancing, blocks of iterations may be assigned to the processors in a cyclic manner.

|              |    |    |     |     |
|--------------|----|----|-----|-----|
| Processor 1: | 1  | 2  | ... | 25  |
| Processor 2: | 26 | 27 | ... | 50  |
| Processor 3: | 51 | 52 | ... | 75  |
| Processor 4: | 76 | 77 | ... | 100 |

Time →

**Loops, Parallel. Fig. 1** Blockwise schedule for a simple parallel loop



**Loops, Parallel.** Fig. 2 Blockwise and cyclic schedule for a triangular loop (a) blockwise schedule, (b) cyclic Schedule

In addition to the static scheduling discussed above, dynamic scheduling may be used when the iterations' execution times vary irregularly. In this case, each processor fetches an iteration from a global work pool, executes it, and then fetches the next one, until the work pool is empty. Dynamic scheduling achieves very good load balancing at the price of the additional overhead for the synchronized access to the global work pool. Like in the static case, processors may fetch contiguous blocks of iterations from the pool in order to increase the cache performance and to decrease the overhead.

### Doacross Loops

A Doacross loop is a parallel loop, where individual iterations are scheduled to the processors in a round-robin fashion, either dynamically or statically. The iterations are started in the same order as in a sequential loop and then execute overlappingly in a pipelined fashion. This allows Doacross loops to possess forward dependences between iterations.

The following example shows a best case for a Doacross loop:

```
for (i=1; i<N; i++) {
 a[i] = f(i);
 b[i] = a[i] - a[i-1];
}
```

Although this loop contains a flow dependence, since the value of  $a[i]$  written in iteration  $i$  is read by  $a[i-1]$  in the following iteration  $i+1$ , it can be executed in parallel with optimal scalability. Figure 3 shows an execution under the assumption that the first statement of the loop body always needs two time units to execute, while the second one needs just one. Of course, in practice, some synchronization statements must be added to the loop in order to ensure the correct temporal order of the dependent statements:

```
for (i=1; i<N; i++) {
 a[i] = f(i);
 signal(i);
 wait(i-1);
 b[i] = a[i] - a[i-1];
}
```

Here, the operation `wait(i-1)` waits until the corresponding `signal(i-1)` has been executed, which indicates that the assignment to `a[i-1]` has finished.

There are also loops, called *serial loops*, where without code restructuring dependences in the loop body can completely inhibit any parallelism, for example,

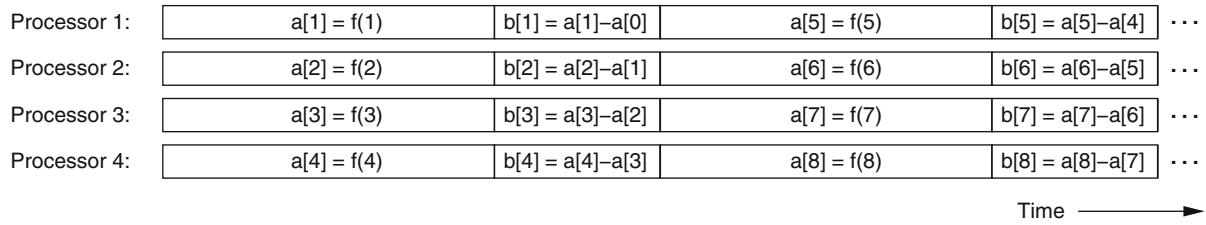
```
for (i=1; i<N; i++) {
 a[i] = b[i-1];
 b[i] = f(i);
}
```

As the schedule in Fig. 3 shows, the loop iterations cannot overlap due to the dependence, since the first statement of iteration  $i+1$  depends on the last statement of iteration  $i$ . However, in this special case, the two statements could be swapped, enabling nearly perfect parallelism. This kind of reorganization is typically performed by interactive restructuring tools or autoparallelising compilers.

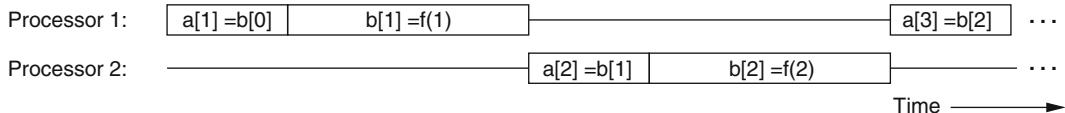
### Forall Loops

The Fortran D language [5] introduced a parallel loop, called *FORALL*, which deals with dependences in a special way: A read operation on a variable will return a new value only if that value was assigned in the current loop iteration, otherwise, it returns the old value of the variable at the time the loop was entered. Thus, conceptually each iteration works on its own copy of the data space.

In High Performance Fortran (HPF) [4], this construct was inherited with some modifications under the



**Loops, Parallel.** Fig. 3 Best case execution of a Doacross loop: full overlap



**Loops, Parallel.** Fig. 4 Worst case execution of a Doacross loop: no overlap

name *FORALL construct*, which was later included into the Fortran 95 language standard. The statements inside a FORALL construct are executed in their sequential order; however, each statement is executed for all elements of the index domain by first performing all read operations and only afterward doing the write operations. For example, in the construct

```
FORALL (i=2:n-1)
 a(i) = a(i-1) + a(i+1)
 b(i) = a(i)
END FORALL
```

first the expression  $a(i-1) + a(i+1)$  will be evaluated for all values of  $i$ , and only afterward all the assignments to  $a(i)$  will occur. Then, the second statement is executed in a similar way to copy  $a$  into  $b$ . Thus, the FORALL construct in HPF and Fortran 95 actually is rather a convenient way to express a sequence of vectorizable and/or parallelizable array assignments than a (parallel) loop.

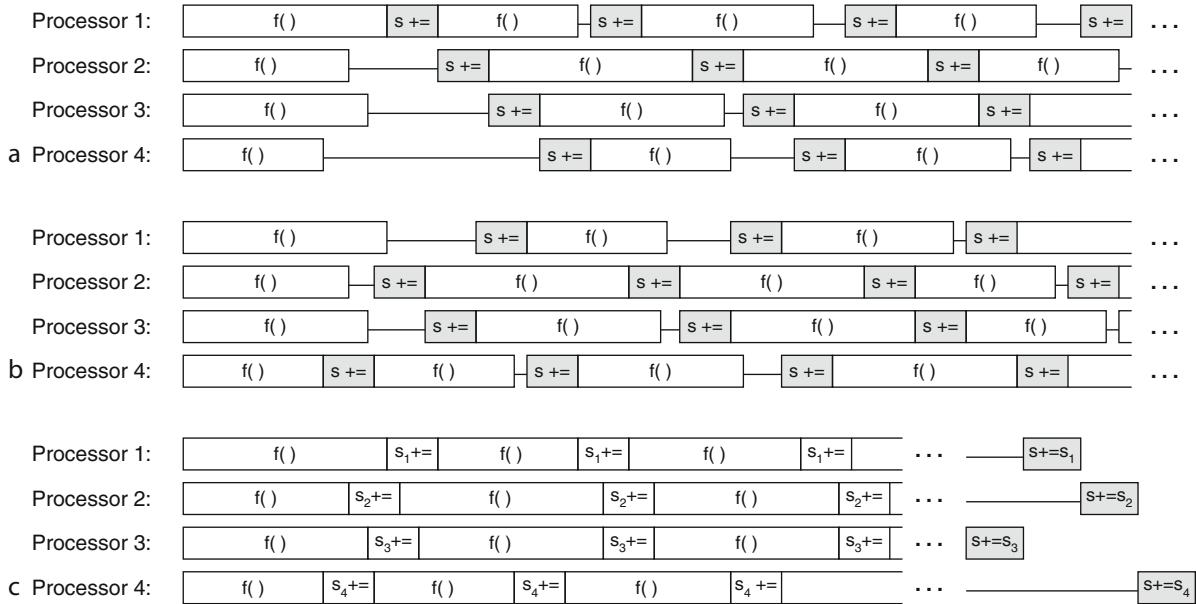
### Variables in Parallel Loops

In a shared memory environment, variables in a parallel loop may either be shared between all threads executing the loop or they may be private, which means that each thread has its own copy of the variable. Typically, the data the loop is working on is shared, while the loop variables of the parallel loop and all its inner loops must be private. Private variables can also be used to compute thread local intermediate results inside a loop, which are combined only after the loop terminated.

A special case of this situation is a reduction variable. A reduction variable is a variable  $v$  such that each assignment in the loop has the form  $v = v \oplus \dots$ , with  $\oplus$  being an associative operator, and  $v$  is not otherwise read in the loop. A typical example is the computation of the integral of a function:

```
double s = 0;
for (i=0; i<N; i++)
 s += f((i+0.5)/N) / N;
```

Strictly, this loop can only be executed in parallel as a Doacross loop, where the exact ordering of the summation is preserved. This is shown in Fig. 5a, where it is assumed that the execution time of  $f()$  shows some variation. If the reduction operation is associative and commutative, which is true for integer arithmetic, but due to rounding errors typically not for floating point arithmetic, the order of the summation may be changed arbitrarily without changing the result. In this case, the loop may be executed as a Doall loop, provided that the concurrent updates to the shared variable  $s$  are protected by a proper synchronisation for mutual exclusion. Fig. 5b shows that although the loop performs better than the Doacross loop, the degree of parallelism is still rather limited. A nearly optimal parallelization can be achieved using a private reduction variable. As indicated in Fig. 5c, each thread first computes a partial sum in its private instance of the variable. At the end of the loop, the partial sums are added to the global sum within a small critical section. Parallel programming interfaces typically allow to mark reduction variables in



**Loops, Parallel.** Fig. 5 Three ways of implementing a parallel reduction (a) Doacross loop, (b) Doall loop, (c) Doall loop with local summation

parallel loops and transparently introduce a local reduction variable as outlined above. A code example using OpenMP is given in the next section.

## Parallel Loops in OpenMP

OpenMP is a parallel programming model for shared memory computers, which extends traditional sequential languages with directives controlling the parallel execution. The main construct is the parallel region, which indicates that the region's code should be executed by a prespecified number of threads. Inside such a parallel region, loops may be designated as parallel loops, resulting in the loop iterations being distributed across the available threads. In C and C++, this is done using the directive

```
#pragma omp for
```

when the loop is already contained in a parallel region, or the combined directive

```
#pragma omp parallel for
```

which additionally creates a parallel region around the loop. These directives must be placed immediately before the loop. OpenMP puts a couple of restrictions on a parallel loop: it must be a `for`-loop with an

integer loop variable, constant increment, and a simple termination test. The loop must not exit prematurely, for example, via a `break` or `return` statement or an exception. In addition, the loop body must neither modify the loop variable nor the loop bounds.

The detailed behavior of the loop can be controlled by a number of optional clauses, which may be appended to the directive:

- The `schedule` clause controls how loop iterations are scheduled to threads. The default behavior is a static blockwise schedule as shown in Fig. 1. Using the clause `schedule(static, size)`, a fixed size for the blocks may be specified. The blocks then are assigned to threads in a round-robin fashion. Thus, for example, `schedule(static, 1)` results in a cyclic schedule like in Fig. 2b. It is also possible to specify a dynamic scheduling, again with an optional block size. The `schedule type guided` uses dynamic scheduling with exponentially decreasing block sizes, which can help to improve the load balancing while keeping the overhead reasonably. Finally, the definition of the `schedule type` can be postponed to the runtime, in order to quickly experiment with different schedules.

- Several other clauses concern the variables used inside the loop. The `shared` and `private` clauses explicitly define variables as being shared between threads or being thread private. The loop variable and all local variables declared inside the loop body are always private. Variables declared outside the loop are shared by default, although this behavior can be changed by using the `default` clause.

The value of a private variable is undefined at the beginning of the loop and is lost at its end. The clauses `firstprivate` and `lastprivate` allow the initialization and finalization of the value, respectively.

The `reduction` clause is used to specify a reduction variable together with the operation used in the reduction. The reduction will then be implemented as shown in Fig. 5c.

- Finally, two clauses control the synchronisation. By default, at the end of each parallel loop, all threads synchronize using a barrier. The `nowait` clause instructs OpenMP to omit this synchronization.

The `ordered` clause indicates that the loop contains an `ordered` directive, which is discussed below.

OpenMP executes parallel loops as Doall loops. However, OpenMP does not check whether the iterations are independent, thus, parallel loops in OpenMP may actually contain dependences. It is the responsibility of the programmer to ensure that the results of the parallel execution are correct. A means to achieve this is the use of synchronization constructs inside the loop. For mutually exclusive access to shared variables, OpenMP provides the `omp critical` and `omp atomic` directives, where the latter is an optimized form for statements like `x += expr`. Another synchronization directive is `omp ordered`, which ensures that the statement(s) controlled by the directive are executed in the same order as in the sequential loop. This directive can only be used in loops having an `ordered` clause and roughly results in the loop behaving like a Doacross loop. Such a loop should always use a `(static, 1)` schedule to achieve a reasonable performance.

In order to illustrate the usage of the mentioned directives, the following code examples show how the

parallel reduction outlined before can be implemented using OpenMP.

(a) As Doacross loop:

```
double s = 0;
#pragma omp parallel for \
ordered schedule(static,1)
for (i=0; i<N; i++) {
 double h = f((i+0.5)/N) / N;
#pragma omp ordered
 s += h;
}
```

(b) As Doall loop:

```
double s = 0;
#pragma omp parallel for
for (i=0; i<N; i++) {
 double h = f((i+0.5)/N) / N;
#pragma omp critical
 s += h;
}
```

(c) Local summation using the reduction clause:

```
double s = 0;
#pragma omp parallel for \
reduction(+: s)
for (i=0; i<N; i++)
 s += f((i+0.5)/N) / N;
```

(d) Explicit local summation:

```
double s = 0;
#pragma omp parallel
{
 double ls = 0;
#pragma omp for
 for (i=0; i<N; i++)
 ls += f((i+0.5)/N) / N;
#pragma omp atomic
 s += ls;
}
```

In the code examples (a) and (b), a private auxiliary variable is used in order to allow the calls to `f()` being executed in parallel. In example (d), `ls` is a private variable, since it is declared inside the parallel region. The execution behavior of these loops is exactly as outlined in Fig. 5.



## Parallel Loops in Intel(R) Threading Building Blocks

The Threading Building Blocks (TBB) are a template library for ISO C++, which includes a variety of constructs supporting the parallel programming with shared memory. Among these constructs are two different kinds of Doall loops, as well as a reduction operation. Since TBB is just a library and does not require any compiler support, parallel loops must be expressed by calling a library function which receives the loop body in the form of a functor, that is, a function object.

For example, the body of the Doall loop shown in the beginning of this entry can be expressed in a class like this:

```
class Body1 {
 double *a;
 double *b;
public:
 void operator()(const tbb::blocked_range<int>& r) const {
 for (int i=r.begin(); i<r.end(); i++) {
 b[i] = sin(PI*i/N);
 a[i] = a[i] + b[i];
 }
 }
 Body1(double *arg_a, double
 *arg_b) { a = arg_a; b = arg_b; }
};
```

The class must define an ()-operator, which must execute the loop body for the index range specified by its parameter. In addition to the class `blocked_range` for a contiguous one-dimensional interval, TBB also provides classes for two- and three-dimensional ranges. A parallel for-loop with the specified body is then written as a function call, for example,

```
tbb::parallel_for(tbb::blocked_range<int>(0,N), Body1(a,b));
```

Here, `a` and `b` are the two array variables, which are shared between the threads executing the loop. Since the loop body is instantiated as an object, pointers to the shared data must be passed to the constructor and stored in attributes of the body class. TBB executes the loop by recursively splitting the index range into a set of subranges, for which the ()-operator will be

invoked. The execution of each subrange forms a task, which is dynamically scheduled to one of the available worker threads. By default, their number is one less than the number of CPU cores, thus reserving one core for the manager thread. The way how the index range is partitioned can be controlled by passing a partitioner as an optional argument to the `parallel_for` function. The default partitioner uses a heuristics to determine a suitable size of the subranges. It can be fine-tuned using an optional parameter in the constructor of `blocked_range`, which allows to specify a minimum size for the subranges. In order to achieve a good speedup, it should be chosen such that the ()-operator executes at least some 10,000 instructions. For parallel loops which are executed repeatedly, TBB provides an additional partitioner that tries to improve cache affinity.

A second kind of parallel loop construct available in TBB implements loops based on C++ iterators. Iterators are extensively used in C++ class libraries in order to iterate over all elements of a collection, for example, a linked list. Although the `parallel_do` loop must usually fetch the work items serially, since most iterators only support a strictly sequential traversal of a collection, the processing of these work items in the loop body may be initiated in an arbitrary order. Thus, also `parallel_do` implements a Doall loop where all iterations must be independent. However, the body of a `parallel_do` is allowed to extend the iteration space by inserting items into the collection. This is illustrated in the following example:

```
class Body2 {
public:
 void operator()(Item& item, tbb::parallel_do_feeder<Item>& feed)
const {
 // may generate a new item
 Item res = processItem(item);
 if (res != NULL) {
 // add item to collection
 feed.add(res);
 }
}
...
std::list<Item> l;
... // initialize list
```

```
tbb::parallel_do(l.begin(),
l.end(), Body2());
```

The loop iterates over all elements of a linked list and invokes the function `processItem` on each of these elements. The processing of an item may result in a new item being generated, which is added to the collection via a `parallel_do_feeder` object passed as an optional argument to the `()`-operator. Since TBB makes extensive use of templates, the items may have an arbitrary data type.

Besides independent loops, TBB allows to implement parallel loops with reductions by using the `parallel_reduce` function. Reduction loops are executed by recursively splitting the index range into subranges, computing the local values for the leaf subranges, and then recursively combining these local results into the global one. Since the initialization of the local result and the combining operation depend on the concrete loop, the class defining the loop's body must provide an additional constructor and a join operation. The following code implements the integration used as an example before:

```
class Sum {
public:
 double sum;
 void operator()(const tbb::
blocked_range<int>& r) {
 for (int i=r.begin(); i<r.end(); i++)
 sum += f((i+0.5)/N) / N;
 }
 // splitting constructor
 Sum(const Sum& s, tbb::split) {
 sum = 0;
 }
 // combining operation
 void join(const Sum& s) {
 sum += s.sum;
 }
 Sum() { sum = 0; }
};

Sum s;
tbb::parallel_reduce(tbb::
blocked_range<int>(0,N), s);
```

The dummy argument of type `tbb::split` is used to distinguish the splitting constructor from a copy constructor.

## Related Entries

- ▶ [Array Languages](#)
- ▶ [Dependence Analysis](#)
- ▶ [HPF \(High Performance Fortran\)](#)
- ▶ [Intel® Threading Building Blocks \(TBB\)](#)
- ▶ [Loop Nest Parallelization](#)
- ▶ [Metrics](#)
- ▶ [OpenMP](#)
- ▶ [Parallelization, Automatic](#)
- ▶ [Reduce and Scan](#)
- ▶ [SPMD Computational Model](#)

## Bibliographic Notes and Further Reading

In the 1980s, many parallel and vector supercomputers, for example, Cray X-MP or Alliant FX/8, started to offer parallel loops as a programming construct, which was implemented by their proprietary compilers. Karp [6] provides an overview of the state of the art of parallel programming at that time, including a discussion of parallel loops.

Since parallel loops are typically supported by a compiler, either by the use of directives like in OpenMP, or by autoparallelization, in-depth discussions can be found in books on parallelizing compilers. Wolfe presents a very good overview on data dependences and techniques for loop parallelization in [10]. In [7], Polychronopoulos discusses different methods for scheduling parallel loops in detail.

Since Fortran is the predominant language in the area of numerical computations, there have been many research efforts to integrate parallel loops into this language, most of them addressing distributed memory parallel computers. The group around Ken Kennedy at Rice University developed Fortran D [5], while in parallel Hans Zima and his coworkers devised Vienna Fortran [3]. Both languages influenced the High Performance Fortran language, whose first specification appeared as a special issue of the Scientific Computing journal [4].

Programming with OpenMP is discussed in ample detail in several books [1, 2, 9]. There is also a textbook [8] about the Intel Threading Building Blocks, which

contains many examples. Additional information about OpenMP and TBB can be found in the related entries of this encyclopedia.

## Bibliography

1. Chandra R et al (2001) Parallel programming in OpenMP. Academic Press, San Diego
2. Chapman B, Jost G, van der Pas R (2008) Using OpenMP – portable shared memory parallel programming. MIT Press, Cambridge, MA
3. Chapman BM, Mehrotra P, Zima HP (1992) Programming in Vienna Fortran. *Sci Prog* 1(1):31–50
4. High Performance Fortran Forum (1993) Section 4: Data parallel statements and directives. *Sci Comput* 2(1–2):55–86
5. Hiranandani S, Kennedy K, Koelbel C, Kremer U, Tseng C-W (Aug 1992) An overview of the Fortran D programming system. Languages and compilers for parallel computing. Lecture Notes in Computer Science, vol 589. Springer, Berlin/Heidelberg, pp 18–34
6. Karp AH (1987) Programming for parallelism. *Computer* 20(5):43–57
7. Polychronopoulos CD (1988) Parallel programming and compilers. Kluwer Academic, Norwell
8. Reinders J (2007) Intel threading building blocks. O'Reilly, Sebastopol, CA
9. Wilkinson B, Allen M (2005) Parallel programming – techniques and applications using networked workstations and parallel computers, 2nd edn. Pearson Education, Prentice-Hall Eaglewood Cliffs
10. Wolfe M (1989) Optimizing supercompilers for supercomputers. Pitman, London

---

## LU Factorization

- [Dense Linear System Solvers](#)
- [Sparse Direct Methods](#)

# M

## Manycore

► [Green Flash: Climate Machine \(LBNL\)](#)

## MapReduce

► [Massive-Scale Analytics](#)

## MasPar

### Synonyms

[SIMD \(Single Instruction, Multiple Data\) Machines](#)

MASPAR [1] were SIMD machines built by MasPar Computer Corporation. The first system was delivered in 1990.

### Related Entries

► [Flynn's Taxonomy](#)

## Bibliography

1. Tom Blank (1990) Compcon Spring '90: the MasPar MP-1 Architecture, San Francisco, CA, USA, 26 Feb–2 Mar 1990, pp 20–24

## Massively Parallel Processor (MPP)

► [Distributed-Memory Multiprocessor](#)

## Massive-Scale Analytics

AMOL GHOTING<sup>1</sup>, JOHN A. GUNNELS<sup>2</sup>, MARK S. SQUILLANTE<sup>3</sup>

<sup>1</sup>IBM Thomas. J. Watson Research Center, Yorktown Heights, NY, USA

<sup>2</sup>IBM Corp., Yorktown Heights, NY, USA

<sup>3</sup>IBM, Yorktown Heights, NY, USA

### Synonyms

[Deep analytics](#); [Large-scale analytics](#); [MapReduce](#)

## Definition

Massive-scale analytics refers to a combination of mathematical methods and high-performance computational methods for the analysis of vast amounts of data in order to gain crucial insights and facilitate decision making across a broad spectrum of application domains, sometimes in an automated data-driven fashion. This requires a unified approach that addresses the two key dimensions of massive-scale analytics: data and computation.

## Discussion

### Introduction

The volumes of data available to various organizations throughout society over the past many years have been growing at an explosive rate. Important advances in computer sciences and technologies have made this possible, which in turn have resulted in the development and growth of data warehouses and reporting capabilities to view information concerning various aspects of an enterprise. These basic reporting capabilities have helped organizations improve the accuracy and timeliness of the data used for the purposes of enterprise decision making. However, significantly increasing the rewards for investments in data collection and data warehouse construction well beyond basic reporting requires the development of massive-scale analytics that provide deeper insights and understanding of the enterprise and optimization of its performance. The overall goal of massive-scale analytics is to enable organizations with data-driven decision-making capabilities based on a combination of advanced mathematical and computational methods, including automated tools and applications.

There are many challenges to realizing the full benefits of massive-scale analytics. This includes important challenges along the two key dimensions of applying massive-scale analytics against huge volumes of data, namely the data dimension which involves machine

learning, knowledge discovery, and data management, together with the computational dimension which involves high-performance and massively parallel computing. The main challenges for the data dimension entail extracting knowledge and insight from complex queries against massive data sets in an efficient and effective manner. The main challenges for the computational dimension entail efficient access to massive data sets and efficient parallel algorithms for computing advanced analytics over these data sets. Far too often, massive-scale analytics are pursued solely from the perspective of one of these dimensions. However, it is only through a general approach which simultaneously addresses these two key dimensions in a unified fashion that the full potential of massive-scale analytics can be realized. In addition to leveraging and combining solutions along both dimensions for creating and developing predictive models, this general massive-scale analytics approach includes the stochastic analysis and optimization of such predictive models to support decision making, scenario analysis, and optimization across a wide variety of applications in a domain-specific manner. This additional aspect of massive-scale analytics raises a number of important challenges to obtain scalable solutions for complex multidimensional stochastic models and decision processes.

Some of the application areas where massive-scale analytics are being actively pursued include advertising and marketing, business intelligence, energy (power grid), health care (genomics, public health), natural sciences (biology), security (internet, national), social networks, workforce management, and transportation. The list of such applications continues to grow at a rapid pace and this trend is expected to endure for the foreseeable future.

## Data Dimension

The massive scale analytics challenge along the data dimension is to provide knowledge and insight on data sets using methods that can gracefully scale with increasing data set sizes. Research on this front has been conducted in the following distinct areas.

## Platforms for Scalable Data Management and Analysis

Successfully processing and analyzing massive data sets requires a platform that can store and allow one to

process data efficiently. Early platforms for scalable data management were provided through database systems. Database systems typically provide a relational data model for data representation, a system for data storage and indexing, and a query processor that can answer queries against the data stored in the system. Parallel database systems improve performance through parallelization of various operations, such as loading data, building indexes, and evaluating queries [11]. Although data may be stored in a distributed fashion, the distribution is governed solely by performance considerations. Such systems improve processing by using multiple CPUs and disks in parallel. Additional programmability is afforded through *user defined functions* – these allow users to embed their custom parallel computations inside a database system. Parallel database systems have been successfully used in the scalable implementation of analytics solutions that go beyond simple database queries [27]. More recently, researchers have also delved into the problem of building database systems that are optimized for data warehouse workloads [31] rather than transactional workloads.

While parallel relational database systems work very well for structured data, more recently, with the advent of applications that need to digest unstructured data (such as plain text, webpages, XML files, RSS feeds) and the need for increased programmability, there has been the development of new systems, programming abstractions, and query languages to store, process, and analyze such data. An example of such an effort is the increasingly popular Apache Hadoop project that uses the MapReduce programming model [10] to provide a simplified means of processing large files on a parallel system through user-defined *Map* and *Reduce* primitives. A MapReduce job consists of two phases – a Map phase and a Reduce phase. During the Map phase, the user-defined *Map* primitive transforms the input data into *(key, value)* pairs in parallel. These pairs are stored and then sorted by the system so as to accumulate all *values* for each *key*. During the Reduce phase, the user-defined *Reduce* primitive is invoked on each unique *key* with a list of all the *values* for that *key*; usually, this phase is used to perform aggregations. Finally, the results are output in the form of *(key, value)* pairs. Each *key* can be processed in parallel during the Reduce phase. A user can implement a variety of parallel analytics computations by submitting one or more MapReduce jobs to

the system. Such systems have been designed to run on large commodity clusters and recover from both data as well as compute node failures. For many domains, the MapReduce paradigm is too low level and there has been the development of custom data management and analysis platforms on top of MapReduce ranging from key-value stores like Cassandra to machine learning platforms such as System-ML [15].

### Algorithms for Scalable Knowledge Discovery and Data Mining

Spurred by advances in data collection technologies, the field of data mining has emerged, merging ideas from statistics, machine learning, databases, and high performance computing. The main challenge in data mining is to extract knowledge and insight from massive data sets in a fast and efficient manner. Data mining methods must scale near-linearly with the data set size. Thus, as data set sizes increase, these techniques must allow one to process increasing volumes of data and at the same time provide a reasonable response time by employing parallelism. Data mining commonly involves four classes of tasks: association rule mining, classification, clustering, and regression. Classification, clustering, and regression have been well studied in the machine learning and statistics communities, but the algorithms were not always designed to scale linearly with the data set size. Over the past 2 decades, the data mining community has borrowed methods from these communities and devised new methods to provide actionable knowledge from large data sets efficiently. Parallel solutions have also been devised that scale using data parallelism. Furthermore, when data parallelism is not an option, techniques that learn on partitions of the data independently and then use the resulting models collectively as an ensemble have also been devised [12]. The reader can look at the entry on parallel data mining for further details.

### Approximate Data Processing and Mining

An alternate strategy to deal with increasing data set sizes is to develop approximate versions of existing algorithms. The approximation should be tunable in the sense that the process should take less time if one can make do with a less accurate answer. This allows one to process increasingly larger data sets in the same amount of time by requesting a less accurate answer.

Database and data mining researchers have developed a suite of approximation techniques to process very large data sets in linear and sometimes even sub-linear time. These techniques make use of summary structures and/or sampling to provide approximate answers, oftentimes with a deterministic or probabilistic error bound. Approximate data processing and mining techniques have also been extended to process data streams where one can only perform one pass over the data. Examples of data streams include computer network traffic, phone conversations, ATM transactions, web searches, and sensor data. The reader can refer to the papers by Garofalakis et al. [13] and Babcock et al. [3] for an introduction to approximate query processing techniques and the application of these techniques to processing data streams, respectively, and the book by Charu Aggarwal [1] for an introduction to issues and solutions related to data stream mining.

### Computational Dimension

#### Architectural Resources

The peak computational ability of systems has been growing at a rate greater than that indicated by Moore's Law, doubling more frequently than every 18 months. For example, the fastest disclosed computer in the world was running just over one teraflop ( $10^{12}$  FLOPS) in 1997 according to the TOP500 list [33]. In 2008 the analogous system at the top of this list broke the petaflop ( $10^{15}$  FLOPS) barrier [23]. Thus, for traditional scientific computations, e.g., physics simulations, floating-point arithmetic is a bountiful resource. As analytics, even deep analytics, applications are rarely more computationally intensive than such scientific computations, it is unlikely that these algorithm's performance will be limited by floating-point resource availability. Memory tends to be the bottleneck for analytics, as it is often the limiting factor for scientific applications. On large HPC systems memory access has a complex characterization. On any system, the standard performance measures of memory are the latency to access and the bandwidth available and both characteristics tend to degrade as one's focus moves from the on-chip cache to the larger main memory and the attached high-capacity nonvolatile memory (typically a hard disk drive). With large, distributed-memory systems, off-chip, off-node, and off-rack memory characteristics come into play. It

is typically the case that as one increases the working set of the problem under consideration it is necessary to use the memory systems of other nodes and to carry out implicit or explicit message passing between those nodes to share data. It is usually the case that the larger the domain under consideration, the greater the latency and the lower the bandwidth available. However, this relationship is not always true. For example, the bandwidth of a hard drive system, especially when viewed as the shared resource that it is almost assured to be, can be lower than the intranode network on a large HPC system. Further, there are architectural features that can ameliorate the situation. Examples include massive multithreading and out-of-order execution, which both tend to greatly mask issues related to high-latency accesses.

### Software and Programmability

In order to efficiently utilize high performance computing resources for analytics the algorithms employed must be parallel. Software that is already executing in parallel, through automatically (compiler) extracted parallelism, lightweight explicit parallelism (e.g., OpenMP, HPF, and X10 [6, 7, 19]), or somewhat intrusive and highly orchestrated parallelism (e.g., Pthreads and MPI [17, 25]) must be extended to encompass larger (higher thread and node count) systems. This can present significant challenges as most analytics codes contain sequential portions (due to the complexity of the algorithm or assumptions about resources available, such as the amount of memory available in a single coherence domain). There is also a good deal of analytics software designed and written for strictly sequential execution. For the most part, this software must be heavily redesigned and the algorithms employed, rethought. As the number of processes increases, Amdahl's law makes it necessary to parallelize an ever-increasing percentage of the overall algorithm. Thus, in either case, a good deal of work is likely required in order to enable analytics software for this domain. This has motivated research into such enabling technologies as the Parallel Global Address Space (PGAS) languages [32] as well as investment in creating software tools in this space (e.g., IBM's High Performance Computing Toolkit [8] and Intel's Parallel Studio [20]).

### HPC System Utilization Modes

There are two fundamental ways in which large HPC systems are currently leveraged for analytics applications. It is usually the case that a small part of a larger system is devoted to a single program execution. This allows analytics to leverage high performance computing systems using current software and, given the nature of stochastic modeling, is an excellent way to leverage such systems. More rarely, analytics codes are rewritten to take advantage of highly parallel (1,000-way and beyond) systems in a unified fashion. This usually requires new algorithmic insights or the revival of algorithms that were once dismissed as impractical (due to their serial performance) as well as significant software engineering [16, 18].

### Stochastic Decomposition

In addition to deep insights and understanding of the enterprise, a unified approach that addresses both data and computational dimensions of massive-scale analytics will provide an organization with predictive modeling, scenario analysis, and decision-making capabilities. This involves the stochastic analysis and optimization of an underlying mathematical representation of such predictive models and decision processes. The multidimensional aspects of these underlying stochastic processes are a major source of complexity, often involving various dependencies and dynamic interactions among the different dimensions of the multidimensional process. Hence, the stochastic analysis and optimization of predictive models and decision processes in a scalable manner as part of massive-scale analytics will often need to leverage a number of additional mathematical and computational methodologies. This includes Monte Carlo methods, a class of computational algorithms based on the use of repeated probabilistic sampling to compute the results of interest; the reader is referred to the entry on Monte Carlo methods for further details. Another related general methodology is based on stochastic decomposition, which essentially consists of decomposing the complex multidimensional stochastic process into a combination of various forms of simpler processes with reduced dimensionality. All of the mathematical and computational methodologies that support stochastic analysis and optimization

for massive-scale analytics can also provide benefits to and exploit benefits from the areas covered in sections ▶Data Dimension and ▶Computational Dimension.

### Nearly Completely Decomposable Systems

One general class of stochastic decomposition approaches is based on the theory of nearly completely decomposable stochastic systems. Consider a discrete-time Markov process with transition probability matrix such that block submatrices along the main diagonal consist of relatively large probability mass, while the elements of the remaining block submatrices are very small in comparison. Stochastic matrices of this type are called nearly completely decomposable [9], in which case the transition probability matrix can be written in a form comprised of stochastic and completely decomposable micro (conditional) models and stochastic macro (unconditioning) models. When the system size is very large, computing the stationary distribution (as well as functionals of the stationary distribution) directly from the transition probability matrix can be prohibitively expensive in both time and space. On the other hand, the overall solution for nearly completely decomposable stochastic matrices can be efficiently computed based on extensions of the Simon-Ando approximations for the stationary distribution of the corresponding Markov process. As a specific example, Aven, Coffman, and Kogan [2] consider model instances where the functional of the stationary distribution represents the stationary page fault probability for a stochastic computer program model and a finite storage capacity. A solution for instances of such computer storage models arising in parallel computing environments, where standard nearly completely decomposable methods breakdown and yield improper solutions, is derived by Peris et al. [26] and Squillante [28] based on first passage times, recurrence times, taboo probabilities and first entrance methods.

### Fixed-Point Based Solutions

Another general class of stochastic decomposition approaches is based on models of each dimension of the multidimensional process in isolation together with a

set of fixed-point equations to capture the dependencies and dynamic interactions among the multiple dimensions. One of the most well-known examples of this general approach is the Erlang fixed-point approximation developed to address the computational complexity of the exact product-form solution for the classical Erlang loss model. This approximate solution is based on a stochastic decomposition in which the multidimensional exact solution is replaced by a system of nonlinear equations in terms of an exact solution for each dimension in isolation. It is well known that there exists a solution of the Erlang fixed-point equations and that this solution converges to the exact solution of the original Erlang loss model in the limit as the number of dimensions grow large under a certain type of scaling; refer to, e.g., Kelly [22]. The corresponding capacity planning problem to optimize an objective function of the stationary loss probabilities and capacities has also been considered within this context by Kelly [22]. The asymptotic exactness of the Erlang fixed-point approximation and optimization based on this approximation is an important aspect of this general decomposition approach for the analysis and optimization of complex multidimensional stochastic processes.

M

### Exploiting Priority Structural Properties

Yet another general class of stochastic decomposition approaches is based on exploiting various priority structural properties to reduce the dimensionality of the stochastic process in a recursive manner. Although this general approach was originally developed for single-server queueing systems with multiple queues under a priority scheduling discipline (see, e.g., [14, 21]), it has been extended and generalized in many different ways for the stochastic analysis and optimization of multidimensional stochastic systems. The basic idea consists of a recursive mathematical procedure starting with the two highest priority dimensions of the process that involves: (a) analyzing the probabilistic behavior of the so-called completion-time process, which characterizes the intervals between consecutive points of interaction between the starts of busy periods for the lower priority dimension; (b) obtaining the distributional characteristics of related busy-period processes through an analysis of associated stochastic processes and modified dimensional probability distributions in isolation; and

(c) determining the solution of the two-dimensional priority process from a combination of these results. These steps are repeated to obtain the solution for the  $(n+1)$ -dimensional priority process using the results for the  $n$ -dimensional priority process, until reaching the final solution for the original multidimensional stochastic process. Several related extensions of this general approach have been developed for the stochastic analysis and optimization of various parallel computing environments. These approaches generally exploit distinct priority structures in the underlying multidimensional stochastic process together with the probabilistic behavior of dependence structures and dynamics resulting from the complex massively parallel computing environment; for one such example, refer to [30].

## Related Entries

►Data Mining

## Bibliographic Notes and Further Reading

For a comprehensive reference on the application of popular machine learning methods to large amounts of data, with a heavy emphasis on parallelization using modern parallel computing frameworks (MPI, MapReduce, CUDA, DryadLINQ, etc.), the reader can refer to the book by Bekkermann et al. [5]. The interested reader is referred to the book by Zaki and Ho [34] for more details on parallelization of data mining algorithms for association rule mining, clustering, and classification. For additional details on high-performance computational algorithms and architectures, the interested reader is referred to [24]. The reader can refer to [4] for a brief overview of HPC operating system issues. For additional details on the stochastic analysis and optimization of multidimensional stochastic processes, the interested reader is referred to [29] and the references cited therein. Nearly completely decomposable stochastic systems and their solutions have received considerable attention in the literature; see, e.g., [2, 9]. Kelly [22] studies the Erlang loss model and the so-called Erlang fixed-point approximation. Gaver [14] and Jaiswal [21] consider stochastic decomposition based on priority structural properties.

## Bibliography

1. Aggarwal C (2007) Data streams: models and algorithms. Springer, New York
2. Aven OI, Coffman EG Jr, Kogan YA (1987) Stochastic analysis of computer storage. D. Reidel, Amsterdam
3. Babcock B, Babu S, Datar M, Motwani R, Widom J (2002) Models and issues in data stream systems. In: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems, ACM, New York, pp 1–16
4. Beckman P, Iskra K, Yoshii K, Coghlan S (2006) Operating system issues for petascale systems. SIGOPS Oper Syst Rev 40:29–33
5. Bekkermann R, Bilenko M, Langford J (2011) Scaling up machine learning. Cambridge University Press, New York
6. Bikshandi G, Almasi G, Kodali S, Saraswat V, Sur S (2009) A comparative study and empirical evaluation of global view HPL program in X10. In: 3rd conference on partitioned global address space programming models 2009 (PGAS), ACM, New York
7. Chapman B, Jost G, van der Pas R (2007) Using open MP: portable shared memory parallel programming (scientific and engineering computation). The MIT Press, Cambridge, MA
8. Chung I-H, Seelam S, Mohr B, Labarta J (2009) Tools for scalable performance analysis on petascale systems. Parallel Distrib Process Symp, Int 0:1–3
9. Courtois PJ (1977) Decomposability. Academic, New York
10. Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Commun ACM 51(1):107–113
11. DeWitt D, Gray J (1992) Parallel database systems: the future of high performance database systems. Commun ACM 35(6):85–98
12. Dietterich T (2000) Ensemble methods in machine learning. In: Kittler J, Roli F (eds) First International Workshop on multiple classifier systems. Springer, New York, pp 1–15
13. Garofalakis M, Gibbons P (2001) Approximate query processing: taming the terabytes. In: Proceedings of very large databases, Rome, Italy
14. Gaver DP Jr. (1962) A waiting line with interrupted service, including priorities. J R Stat Soc, Ser B 24:73–90
15. Ghosh A, Krishnamurthy R, Pednault E, Reinwald B, Sindhwanvi V, Tatikonda S, Tian Y, Vaithyanathan S (2011) SystemML: declarative machine learning on MapReduce. In: Proceedings of the IEEE international conference on data engineering
16. Ghosh A, Makarychev K (2009) Proceedings of the ACM/IEEE conference on high performance computing, SC 2009, 14–20 Nov 2009, Portland, OR. In: SC. ACM, New York
17. Gropp W, Lusk E, Thakur R (1999) Using MPI-2: advanced features of the message passing interface. MIT Press, Cambridge, MA
18. Gunnels J, Lee J, Margulies S (2010) Efficient high-precision matrix algebra on parallel architectures for nonlinear combinatorial optimization. Math Program Comput 2:103–124. doi: 10.1007/s12532-010-0014-4
19. Gupta M, Midkiff S, Schonberg E, Seshadri V, Shields D, Wang K-Y, Ching W-M, Ngo T (1995) An HPF compiler for the IBM SP2. In: Proceedings of the ACM Conference on Supercomputing, December 1995

20. Intel Corporation Intel parallel studio – compiler, libraries and analysis tools (2011). <http://software.intel.com/en-us/articles/intel-parallel-studio-home/>
21. Jaiswal NK (1968) Priority queues. Academic, New York
22. Kelly FP (1991) Loss networks. Ann Appl probab 1(3):319–378
23. Kistler M, Gunnels J, Brokenshire D, Benton B (2008) Programming the Linpack benchmark for the IBM PowerXCell 8i processor. Sci. Program 17:43–57
24. Kumar V, Grama A, Gupta A, Karypis G (1994) Introduction to parallel computing: design and analysis of algorithms. Benjamin-Cummings, Redwood City
25. Nichols B, Buttlar D, Farrell JP (1996) Pthreads programming. O'Reilly & Associates, Sebastopol
26. Peris VG, Squillante MS, Naik VK (1994) Analysis of the impact of memory in distributed parallel processing systems. In: Proceedings of ACM SIGMETRICS conference on measurement and modeling of computer systems, ACM, New York, pp 5–18
27. Sarawagi S, Thomas S, Agrawal R (2000) Integrating association rule mining with relational database systems: alternatives and implications. Data Min Knowl Discov 4(2):89–125
28. Squillante MS (2005) Stochastic analysis of resource allocation in parallel processing systems. In: Gelenbe E (ed) Computer system performance modeling in perspective: a tribute to the work of Prof. Sevcik KC. Imperial College Press, London, pp 227–256
29. Squillante MS (2011) Stochastic analysis and optimization of multiserver systems. In: Ardagna D, Zhang L (eds) Run-time models for self-managing systems and applications, chapter 1 Springer, Berlin
30. Squillante MS, Zhang Y, Sivasubramaniam A, Gautam N, Franke H, Moreira J (2002) Modeling and analysis of dynamic coscheduling in parallel and distributed environments. In: Proceedings of ACM SIGMETRICS conference on measurement and modeling of computer systems, ACM, New York, pp 43–54
31. Stonebraker M, Abadi D, Batkin A, Chen X, Cherniack M, Ferreira M, Lau E, Lin A, Madden S, O'Neil E (2005) C-store: a column-oriented DBMS. In: Proceedings of the 31st international conference on very large data bases, VLDB Endowment, Trondheim, Norway, pp 553–564
32. Su J, Yellick K (2008) Automatic communication performance debugging in PGAS languages. In: Adve V, Garzarán MJ, Petersen P (eds) Languages and compilers for parallel computing. Springer, Berlin/Heidelberg, pp 232–245
33. TOP500 Organization Performance development – TOP500 supercomputing sites. [http://www.top500.org/lists/2010/11/performance\\_development](http://www.top500.org/lists/2010/11/performance_development). Accessed on November, 2010
34. Zaki M, Ho C (2000) Large-scale parallel data mining. Springer, New York

## Maude

JOSÉ MESEGUER

University of Illinois at Urbana-Champaign, Urbana, IL, USA

### Synonyms

Rewriting logic

### Definition

Maude [4, 5] is a declarative language whose modules are theories in rewriting logic [13], and whose computation is logical deduction by concurrent rewriting modulo the equational structural axioms of each theory. Since virtually all models of concurrent computation can be naturally and easily expressed as rewrite theories [12–14], Maude and its tool environment can be used both as a parallel language, and as a flexible *semantic framework* to define, execute, formally analyze, and implement other parallel languages.

### Discussion

#### Rewriting Logic and Maude in a Nutshell

Many different models of concurrency have been proposed and used: Petri nets, parallel functional programming, Actors, the  $\pi$ -calculus and other process calculi, and so on. Rewriting logic [13] is not just one more model of concurrency. It is instead a *logical framework* for concurrency in which different models and different parallel languages can be formally specified, executed, and analyzed as *rewrite theories*. This can be done without imposing a *representational bias* that favors some specific concurrency model, and without complex *encodings* that distort and obscure the specific features of each model.

The reasons why this natural expression of different concurrency models within rewriting logic is possible are essentially twofold, namely, the ease and generality with which: (1) different kinds of *distributed states* and (2) different kinds of *local concurrent transitions* can be specified. A *rewrite theory*  $(\Sigma, E, R)$  consists of an equational theory  $(\Sigma, E)$  and a set of rewrite rules  $R$ , and specifies a concurrent system. The distributed states of such a system are specified by the equational theory

## Matrix Computations

► Linear Algebra, Numerical

$(\Sigma, E)$ , where  $\Sigma$  is a collection of typed operators which includes the *state constructors* that build up a distributed state out of simpler state components, and where  $E$  specifies the algebraic identities that such distributed states enjoy. That is, the distributed states are specified as an *algebraic data type* defined as the initial algebra semantics (see, e.g., [15]) of the equational theory  $(\Sigma, E)$ . Concretely, this means that a distributed state is mathematically represented as an  $E$ -equivalence class  $[t]_E$  of terms (i.e., algebraic expressions) built up with the operators declared in  $\Sigma$ , *modulo* provable equality using the equations  $E$ , so that two state representations  $t$  and  $t'$  describe the *same* state if and only if one can prove the equality  $t = t'$  using the equations  $E$ . Instead, the *local concurrent transitions* of the concurrent system specified by  $(\Sigma, E, R)$  are described by its set  $R$  of *rewrite rules*. Each rewrite rule in  $R$  has the form  $t \rightarrow t'$ , where  $t$  and  $t'$  are algebraic expressions in the syntax of  $\Sigma$ . The lefthand side  $t$  describes a *local firing pattern*, and the righthand side  $t'$  describes a corresponding *replacement pattern*. That is, any fragment of a distributed state which is an instance of the firing pattern  $t$  can perform a local concurrent transition in which it is replaced by the corresponding instance of the replacement pattern  $t'$ . Both  $t$  and  $t'$  are typically *parametric* patterns, describing not single states, but parametric families of states. The parameters appearing in  $t$  and  $t'$  are precisely the *mathematical variables* that  $t$  and  $t'$  have, which can be instantiated to different concrete expressions by a mapping  $\theta$ , called a *substitution*, sending each variable  $x$  to a term  $\theta(x)$ . The instance of  $t$  by  $\theta$  is then denoted  $\theta(t)$ .

The most basic *logical deduction steps* in a rewrite theory  $(\Sigma, E, R)$  are precisely atomic concurrent transitions, called *atomic rewrites*, corresponding to applying a rewrite rule  $t \rightarrow t'$  in  $R$  to a state fragment that is an instance of the firing pattern  $t$  by some substitution  $\theta$ . That is, up to  $E$ -equivalence, the state is of the form  $C[\theta(t)]$ , where  $C$  is the rest of the state not affected by this atomic transition. Then the resulting state is precisely  $C[\theta(t')]$ , so that the atomic transition has the form  $C[\theta(t)] \rightarrow C[\theta(t')]$ . Rewriting is *intrinsically concurrent*, because many other atomic rewrites can potentially take place in the rest of the state  $C$  (and in the substitution  $\theta$ ), at the same time that the local atomic transition  $\theta(t) \rightarrow \theta(t')$  happens. That is, in general one may have complex concurrent transitions of the form  $C[\theta(t)] \rightarrow C'[\theta'(t')]$ , where the rest of the state  $C$  has evolved to  $C'$  and the substitution  $\theta$  has evolved to  $\theta'$

by other (possibly many) atomic rewrites simultaneous with the atomic rewrite  $\theta(t) \rightarrow \theta(t')$ . The rules of deduction of rewriting logic [2, 13] (which in general allow rules in  $R$  to be *conditional*) precisely describe all the possible, complex concurrent transitions that a system can perform, so that concurrent computation and logical deduction *coincide*.

Maude [4, 5] is a language and system implementation directly based on rewriting logic so that: (1) a Maude Module is exactly a rewrite theory  $(\Sigma, E, R)$  satisfying some simple executability conditions that make it easy to compute with the rules  $R$  modulo the equations  $E$  (see [5] for details) and (2) computation in Maude is logical deduction by rewriting.

*A Simple Example.* All the ideas discussed above can be illustrated by means of a simple example of a concurrent object-based system specified in Maude. Maude's syntax is user-definable; therefore, all state-building operators can be declared by the user with any desired “mixfix” syntax. A concurrent state made up of objects and messages can be thought of as a “soup” in which objects and messages are freely floating and can come into contact with each other in communication events. Mathematically, this means that the concurrent state, called a *configuration*, is modeled as a *multiset* or *bag* built up by a multiset union operator which satisfies the axioms of *associativity* and *commutativity*, with the empty multiset as its *identity element*. In Maude one can, for example, choose to express multiset union with *empty syntax*, that is, just by *juxtaposition*. One can achieve this by declaring the type (called a *sort*) Configuration of configurations, which contains the sorts Object and Msg as *subsorts*, and the empty configuration, say none, and the configuration union operator by the declarations:

```
sorts Object Msg Configuration .
subsorts Object Msg < Configuration .
op none : -> Configuration [ctor] .
op _ _ : Configuration Configuration ->
Configuration
[ctor config assoc comm id: none] .
```

Each operator is declared with the *op* keyword, followed by its syntax, the list of its argument sorts, an arrow  $->$ , and its result sort. And all declarations are finished by a space and a period. In the case of the configuration union operator, there are two argument positions, which are marked by underbars. Before and/or after such underbars, any desired syntax

tokens can be declared. In this case an empty syntax (juxtaposition) has been chosen, so that no tokens at all are declared. If, instead, one wanted to describe configurations as comma-separated collections of objects and messages, one could give the alternative syntax declaration:

```
op _,_ : Configuration Configuration ->
 Configuration
 [ctor config assoc comm id: none] .
```

Note that constants like `none` are viewed as *operators with no arguments*. The keyword `config` declares that this is a union operator for configurations of objects and messages. This allows Maude to execute such configurations with an object and message fair strategy, so that all objects having messages addressed to them do eventually process such messages. The `assoc comm id: none` declarations declare the associativity axiom  $(x y) z = x (y z)$ , the commutativity axiom  $x y = y x$ , and the identity axiom  $x \text{ none} = x$ . Maude then supports rewriting *modulo* such axioms, so that a rule can be applied to a configuration regardless of parentheses, and regardless of the order of arguments. The `ctor` keyword declares that both `none` and `_ _` are state-building *constructors*, as opposed to functions defined on constructors such as, for example, a cardinality function, say `|_|`, that counts the total number of objects and messages present in a configuration, and which could be defined with the following syntax and equations:

```
op |_| : Configuration -> Nat .
var O : Object . var M : Msg . var C :
 Configuration .
eq | none | = 0 .
eq | O C | = 1 + | C | .
eq | M C | = 1 + | C | .
```

where `Nat` is the sort of natural numbers in the `NAT` module. Note that the above equations are applied by Maude modulo the associativity, commutativity, and identity axioms for `_ _`. For example, by instantiating `C` to `none`, the second equation defines the cardinality of a single object `O` to be  $1 + | \text{none} | = 1$ .

Consider an object-based system containing three classes of objects, namely, `Buffer`, `Sender`, and `Receiver` objects, so that a sender object sends to the corresponding receiver a sequence of values (say natural numbers) which it reads from its own buffer, while the receiver stores the values it gets from the sender in its

own buffer. In Maude's Full Maude language extension (see Part II of [5]), such object classes can be declared as subsorts of the `Object` sort in *class declarations*, which specify the names and sorts of the *attributes* of objects in the class. The above three classes can be defined with class declarations:

```
class Buffer | q : NatList, owner : Oid .
class Sender | cell : Nat?, cnt : Nat,
 receiver : Oid .
class Receiver | cell : Nat?, cnt : Nat .
```

An object of a class `C1` declared with attributes `a1` of sort `A1`, ..., `an` of sort `An`, is a record-like structure of the form:

```
< oid : C1 | a1 : v1, ... , an : vn >
```

where each `vi` is a term of sort `Ai`. For example, assuming that one uses quoted identifiers as object identifiers (by importing the `QID` module and giving the subsort declaration `Qid < Oid`), and that the supersort `Nat?` of `Nat` containing an empty value `mt`, and the sort `NatList` of lists of natural numbers have been declared as:

```
sorts Nat? NatList .
subsorts Nat < Nat? NatList .
op mt : -> Nat? [ctor] .
op nil : -> NatList [ctor] .
op _,_ : NatList NatList ->
 NatList [ctor assoc id: nil] .
```

then the following is an initial configuration of a sender and a receiver object, each with their own buffer, and each with their cell currently empty:

```
< 'a : Buffer | q : 1 . 2 . 3 , owner : 'b >
< 'b : Sender | cell : mt , cnt : 0 ,
 receiver : 'd >
< 'c : Buffer | q : nil , owner : 'd >
< 'd : Receiver | cell : mt , cnt : 1 >
```

A sender object can send messages to its corresponding receiver object. The programmer has complete freedom to define the format of such messages by declaring an operator of sort `Msg`, using the `msg` keyword instead of the more general `op` keyword to emphasize that the resulting terms are messages. For example, one can choose the following format:

```
msg to_:_from_cnt_ : Oid Nat Oid Nat -> Msg.
```

where a message, say, `to 'd' : 3 from 'b' cnt 1`, means that '`b`' sends to '`d`' the data item 3, with counter 1,

indicating that this is the *first* element transmitted. This last information is important, since message passing in a configuration is usually asynchronous, so that messages could be received out of order. Therefore, receiver objects need to use the counter information to properly reassemble a list of transmitted data. Of course, out-of-order communication is just *one* possible situation that can be modeled. If, instead, one wanted to model in-order communication, the distributed state could contain *channels*, similar for example to the buffer objects, so that axioms of associativity and identity are satisfied when inserting messages into a channel, but *not commutativity*, which is the axiom allowing out-of-order communication in an asynchronous configuration of objects and messages. All that has been done up to now is to define the *distributed states* of this object-based system as the algebraic data type associated to an equational theory, namely, the equational theory  $(\Sigma, E)$ , where  $\Sigma$  is the signature whose sorts have been declared with the `sort` (and `class`) keywords, with subsort relations declared with the `subsort` keyword, and whose operators have been declared with the `op` (or `msg`) keywords. And where the equations  $E$  have been declared in one of two ways: (1) either as explicit equations, specified with the `eq` keyword, or as (2) *equational axioms* of associativity and/or commutativity and/or identity associated to specific operators, declared with the `assoc`, `comm` and `id:` keywords.

What about the concurrent *transitions* for buffers, senders, and receivers? As already mentioned, they are specified by a set  $R$  of rewrite rules which describe the local concurrent transitions possible in the system. One can, for example, define such transitions by the following four rewrite rules (note that object attributes not changed by a rule need not be mentioned in it):

```

vars X Y Z : Oid . vars N E : Nat .
vars L L' : NatList .

rl [read] : < X : Buffer | q : L . E ,
owner : Y > < Y : Sender | cell : mt,
cnt : N >
=> < X : Buffer | q : L, owner : Y >
< Y : Sender | cell : E, cnt : N + 1 > .

rl [write] : < X : Buffer | q : L , owner :
Y > < Y : Receiver | cell : E >
=> < X : Buffer | q : E . L , owner : Y >
< Y : Receiver | cell : mt > .

```

```

rl [send] : < Y : Sender | cell : E , cnt :
N , receiver : Z >
=> < Y : Sender | cell : mt, cnt : N >
(to Z :: E from Y cnt N) .

rl [receive] : < Z : Receiver | cell : mt ,
cnt : N > (to Z :: E from Y cnt N)
=> < Z : Receiver | cell : E , cnt :
N + 1 > .

```

That is, senders can read data from the buffer they own and update their count; and receivers can write their received data in their own buffer. Also, each time a sender has a data element in its cell, it can send it to its corresponding receiver with the appropriate count; and a receiver with an empty cell can receive a data item from its sender, provided it has the correct counter. Note that rewriting is intrinsically concurrent; for example, '`b`' could be sending the next data item to '`d`' at the same time that '`d`' is receiving the previous data item or is writing it into its own buffer. Note also that the rules `send` and `receive` describe the *asynchronous message passing* communication between senders and receivers typical of the Actor model [1]. Instead, the `read` and `write` rewrite rules describe *synchronization events*, in which a buffer and its owner object synchronously transfer data between each other. This illustrates the flexibility of rewriting logic as a semantic framework: No assumption of either synchrony or asynchrony is built into the logic. Instead, many different styles of concurrency and of in-order or out-of-order communication can be directly modeled without any difficulty.

The behavior of the above system can be simulated by means of Maude's `rewrite` (abbreviated `rew`) command. For example, one can see the result of a terminating execution of the initial configuration described above by giving to Full Maude the following command:

```

Maude> (rew < 'a : Buffer | q : 1 . 2 . 3 ,
owner : 'b >
< 'b : Sender | cell : mt , cnt : 0 ,
receiver : 'd > < 'c : Buffer | q :
nil , owner : 'd >
< 'd : Receiver | cell : mt , cnt : 1 > .)
result Configuration :
< 'a : Buffer | owner : 'b,q : nil >
< 'b : Sender | cell : mt,cnt :
3,receiver : 'd >
< 'c : Buffer | owner : 'd,q :(1 . 2 . 3)>
< 'd : Receiver | cell : mt,cnt : 4 >

```

Since a concurrent system can be non-terminating, a `rewrite` command may never terminate; for this reason, the `rewrite` command can be given an extra numerical parameter, indicating the maximum number of rewrite steps that should be simulated. Furthermore, a concurrent system can often be highly non-deterministic, so that the result of a `rewrite` command provides just *one* fair simulation of the system among possibly many others. Instead, functions defined by *equations* with the `eq` keyword (as opposed to the concurrent transitions defined by *rules* with the `r1` keyword) should have a single final result. The evaluation of functional expressions to their unique final result using the module's equations from left to right is supported by a different command, namely, the `reduce` (abbreviated `red`) command. For example, one can compute the cardinality of the above initial configuration by giving to Full Maude the command:

```
Maude> (red | < 'a : Buffer | q : 1 . 2 . 3 ,
 owner : 'b >
< 'b : Sender | cell : mt , cnt : 0 ,
 receiver : 'd > < 'c : Buffer | q : nil ,
 owner : 'd >
< 'd : Receiver | cell : mt , cnt : 1 > | ..)

result NzNat :
 4
```

Simulating a system with the `rewrite` command is useful but insufficient. One would like, for example, to verify that for *all* behaviors some undesired situation (for example a deadlock) does not happen; or one might like to find what different states of a certain kind (final states, or states satisfying a certain pattern and/or condition) can be reached from a given initial state. This can be achieved by using Maude's `search` command, which performs breadth-first search from an initial state looking for states specified by a *pattern* (a constructor term with variables, so that the desired states are instances of it), and possibly by an additional semantic condition specified with the `such that` keyword. For example, one may wish to check the invariant that all states reachable from our initial configuration exhibit *in-order reception* of the list `1 . 2 . 3` of numbers initially stored in the sender's buffer; that is, that any list stored in the receiver's buffer is always a *postfix* of the list `1 . 2 . 3`. This can be done by first equationally defining a `postfix` predicate on pairs of lists as follows:

```
op postfix : NatList NatList -> Bool .
eq postfix(L,L' . L) = true .
eq postfix(L,L') = false [owise] .
```

where an equation declared with the “otherwise” (`owise`) keyword abbreviates a conditional equation that is applied only when all other equations defining the same function cannot be applied (see Sect. 4.5.4 of [5]). One can then give the following `search` command to Full Maude to verify the desired invariant by searching for a state violating such an invariant (note Maude's support for on-the-fly introduction of mathematical variables by appending to a variable's name a colon and the sort of the variable, as done below for `C:Configuration` and `L:NatList`):

```
Maude> (search < 'a : Buffer | q : 1 . 2 . 3 ,
 owner : 'b >
< 'b : Sender | cell : mt , cnt : 0 ,
 receiver : 'd > < 'c : Buffer | q : nil ,
 owner : 'd >
< 'd : Receiver | cell : mt , cnt : 1 >
=>* C:Configuration < 'c : Buffer | q :
L:NatList , owner : 'd > such that
postfix(L:NatList,1 . 2 . 3) = false .)
```

No solution.

In this example, the number of reachable states from the initial configuration is finite, so that the `search` command can exhaustively search all reachable states. However, the `search` command can be given even when the number of reachable states is infinite. It will then return all the states it finds satisfying the given pattern and semantic condition. In particular, if one searches for the violation of an invariant and the invariant is indeed violated, it will find such a violation in finite time.

Besides the possibility of using the `search` command for verifying invariants, whenever the set of states reachable from an initial state is finite, Maude also provides the possibility of *model checking* any desired linear-time logic (LTL) formula. This can be easily done by equationally defining the desired state predicates used as atomic propositions in the given LTL formula, importing Maude's `MODEL-CHECKER` module, and then giving a command to model check the desired formula from the given initial state. For example, for the above initial configuration one may wish to model check the liveness property that all system behaviors eventually terminate with the sender's buffer empty, the sender's and receiver's cells empty, and the receiver's buffer containing the list originally stored in the sender's

buffer. A detailed description of Maude’s LTL model checking capabilities can be found in Chap. 13 of [5].

## Rewriting Logic Semantics of Programming Languages

The flexibility of rewriting logic to naturally express many different models of concurrency can be exploited not just at the theoretical level, for expressing such models both deductively, and denotationally in the model theory of rewriting logic [13, 14]: It can also be applied to give *formal definitions of concurrent programming languages* by specifying the concurrent model of a language  $\mathcal{L}$  as a rewrite theory  $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$ , where: (1) the signature  $\Sigma_{\mathcal{L}}$  specifies both the syntax of  $\mathcal{L}$  and the types and operators needed to specify semantic entities such as the store, the environment, input-output, and so on; (2) the equations  $E_{\mathcal{L}}$  can be used to give semantic definitions for the *deterministic* features of  $\mathcal{L}$  (a sequential language typically has only deterministic features and can be specified just equationally as  $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}})$ ); and (3) the rewrite rules  $R_{\mathcal{L}}$  are used to give semantic definitions for the concurrent features of  $\mathcal{L}$  such as, for example, the semantics of threads. By specifying the rewrite theory  $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$  as a Maude module, it becomes not just a mathematical definition but an *executable* one, that is, an *interpreter* for  $\mathcal{L}$ . Furthermore, one can leverage Maude’s generic search and LTL model checking features to automatically endow  $\mathcal{L}$  with powerful *program analysis capabilities*. For example, the search command can be used in the module  $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$  to detect any violations of invariants, e.g., a deadlock or some other undesired state, of a program in  $\mathcal{L}$ . Likewise, for terminating concurrent programs in  $\mathcal{L}$  one can model check any desired LTL property. All this can be effectively done not just for toy languages, but for real ones such as Java and the JVM, and with performance that compares favorably with state-of-the-art tools such as Java PathFinder [10, 11]. There are essentially three reasons for this surprisingly good performance. First, rewriting logic’s distinction between equations  $E_{\mathcal{L}}$ , used to give semantics to deterministic features of  $\mathcal{L}$ , and rules  $R_{\mathcal{L}}$ , used to specify the semantics of concurrent features, provides in practice an enormous *state space reduction*. Note that a state of  $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$  is, by definition, an  $E_{\mathcal{L}}$ -equivalence class  $[t]_{E_{\mathcal{L}}}$ , which in practice is represented as the state of the program’s execution after all deterministic execution steps possible at a given

stage have been taken. That is, the equations  $E_{\mathcal{L}}$  have the effect of “fast forwarding” such an execution by skipping all intermediate deterministic steps until the next truly concurrent interaction is reached. For example, for  $\mathcal{L} = \text{Java}$ ,  $E_{\text{Java}}$  has hundreds of equations, but  $R_{\text{Java}}$  has just 5 rules. The second reason is of course that the underlying Maude implementation is a high-performance one that can reach millions of rewrite steps per second. The third reason is that the intrinsic flexibility of rewriting logic means that it does not prescribe a fixed style for giving semantic definitions. Instead, *many different styles*, for example, small-step or big-step semantics, reduction semantics, CHAM-style semantics, or continuation semantics, can all be naturally supported [20]. But not all styles are equally efficient; for example, small-step semantics makes heavy use of conditional rewrite rules, insists on modeling every single computation step as a rule in  $R_{\mathcal{L}}$ , and is in practice horribly inefficient. Instead, the continuation semantics style described in [20] and used in [10] is very efficient.

The good theoretical and practical advantages of using rewriting logic to give semantic definitions to programming languages have stimulated an international research effort called the *rewriting logic semantics project* (see [16, 17, 20] for some overview papers). Not only have semantic definitions allowing effective program analyses been given for many real languages such as Java, the JVM, Scheme, and C, for hardware description languages such as ABEL and Verilog, and for software modeling languages such as MOF, Ptolemy and AADL: It has also been possible to build a host of sophisticated program analysis tools for real languages based on different kinds of *abstract semantics*. The point is that instead of a “concrete semantics”  $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$ , describing the actual execution of programs in a language  $\mathcal{L}$ , one can just as easily define an “abstract semantics”  $(\Sigma_{\mathcal{L}}^A, E_{\mathcal{L}}^A, R_{\mathcal{L}}^A)$  describing any desired abstraction  $A$  of  $\mathcal{L}$ . To give just two examples, one can give an abstract semantics for type checking by replacing operations on concrete values by operations on their corresponding types [9]; and one can likewise give an abstract semantics to scientific programs by operating not on their numerical values, but on their intended *units of measurement* [3]. All this means that many different forms of program analysis, much more scalable than the kind of search and model checking based on a language’s concrete semantics,

become available essentially for free by using Maude to execute and analyze one's desired abstract semantics  $(\Sigma_L^A, E_L^A, R_L^A)$ .

Two further developments of the rewriting logic semantics project, both pioneered by Grigore Roșu with several collaborators, are worth mentioning. One is the K *semantic framework* for programming language definitions [19], which provides a very concise and highly modular notation for such definitions. The K-Maude tool then automatically translates language definitions in K into their corresponding rewrite theories in Maude for execution and program analysis purposes. Another is *matching logic* [18], a program verification logic with substantial advantages over both Hoare logic and separation logic which uses a language's rewriting logic semantics, including the possibility of using patterns to symbolically characterize sets of states, to mechanize the formal verification of programs, including programs that manipulate complex data structures using pointers.

## Maude's Implementation and Formal Environment

Maude is implemented in C++ as a high-performance semi-compiled interpreter. Both equations and rules are (semi-)compiled into matching and replacement automata [7, 8]. This makes it possible to *trace* every single step of rewriting with both equations and rules. In the sequential Maude implementation, the execution of concurrent systems is *simulated* by the `rewrite` or `frewrite` commands, which each apply a module's rules with a specific fair strategy [5]. However, because of its built-in support for sockets (see Sect. 11.4 in [5]), Maude is also a *parallel language*, which can be used to program in a declarative, high-level way interesting distributed applications, for example, mobile languages (see Chap. 16 of [5]). Using sockets, a concurrent configuration of objects and messages, which at the sequential level would be represented by a single multiset data structure, can now be distributed and executed over several machines, where each machine holds its own *local configuration* of objects and messages as a multiset, and rewrites it sequentially in a fair way. Objects in a local configuration can send messages not only to other objects in the same local configuration, but to remote objects in other machines using sockets. The fact that the rewriting of objects and messages is automatically fair for each object means that

*no explicit scheduling is needed*; that is, by the intrinsic fairness with which rewriting of objects and messages is implemented, all objects in the different local configurations always get to process the messages sent to them either from other objects in the same local configuration or from remote objects. An upcoming next generation of Maude, currently under development at SRI International and the University of Illinois, will add to the present capabilities of Maude for distributed computing the additional ability of exploiting the finer parallelism provided by multicore machines. This means that the local configurations of objects and messages running in each multicore machine will themselves also be executed concurrently, with different objects in the same local configuration executing on different cores.

Besides the search and LTL model checking features directly supported by the Maude implementation, which allow reachability analysis and LTL formal verification of Maude programs, Maude also has a *formal environment* of tools supporting other types of formal verification, including: (1) inductive theorem proving for equational theories (called *functional modules* in Maude) using its ITP tool; (2) proofs of termination using its MTT tool; (3) proofs of local confluence for equations with its CRC tool (which ensure that functional evaluations with equations using the `reduce` command are deterministic), and of coherence of rules with respect to equations with its ChC tool (which ensure that execution of equations and of rules "commutes" in an appropriate sense); (4) proofs of *sufficient completeness* in its SCC tool, ensuring that enough equations have been given to fully define each non-constructor function symbol; and (5) execution and model checking analysis of real-time systems with its Real-Time Maude tool. A common feature of all these tools (and also of the Full Maude language extension, see Part II of [5]) is their systematic use of *reflection* to manipulate the Maude modules that they analyze as terms within rewriting logic itself. This powerful form of logical reflection is based on the fact that rewriting logic and equational logic are both reflective logics [6], and is efficiently supported by Maude's META-LEVEL module (see Chaps. 14–15 of [5]). Besides the formal tools used to analyze Maude modules, Maude can likewise be used to build domain-specific tools for specific languages or logical systems, including the already-mentioned tools for programming languages.

Some of these other tools, as well as the Maude formal tools, are discussed in Chap. 21 of [5]. Maude itself, its documentation, and its formal tools are all freely available at [maude.cs.uiuc.edu](http://maude.cs.uiuc.edu).

## Related Entries

- [Actors](#)
- [CSP \(Communicating Sequential Processes\)](#)
- [Functional Languages](#)
- [Linda](#)
- [Logic Languages](#)
- [Petri Nets](#)
- [Pi-Calculus](#)
- [Process Algebras](#)

## Bibliography

1. Agha G (1986) Actors: model of concurrent computation in distributed systems. MIT Press, Cambridge, MA
2. Bruni R, Meseguer J (2006) Semantic foundations for generalized rewrite theories. *Theor Comput Sci* 360(1–3):386–414
3. Chen F, Roşu G, Venkatesan RP (2003) Rule-based analysis of dimensional safety. In: Nieuwenhuis R (ed) Rewriting techniques and applications: 14th international conference (RTA'03: Proc.), Valencia, 9–11 June 2003. Lecture notes in computer science, vol 2706. Springer, Berlin, pp 197–207
4. Clavel M, Durán F, Eker S, Lincoln P, Martí-Oliet N, Meseguer J, Quesada J (2002) Maude: specification and programming in rewriting logic. *Theor Comput Sci* 258:187–243
5. Clavel M, Durán F, Eker S, Meseguer J, Lincoln P, Martí-Oliet N, Talcott C (2007) All about Maude – a high-performance logical framework. Lecture notes in computer science, vol 4350. Springer, Berlin
6. Clavel M, Meseguer J, Palomino M (2007) Reflection in membership equational logic, manysorted equational logic, Horn logic with equality, and rewriting logic. *Theor Comput Sci* 373:70–91
7. Eker S (1996) Fast matching in combination of regular equational theories. In: Meseguer J (ed) Proc. First Intl. Workshop on Rewriting Logic and its Applications, Pacific Grove, 3–6 Sept 1996. Electronic Notes in Theoretical Computer Science, vol 4. Elsevier, Amsterdam
8. Eker SM (2003) Associative-commutative rewriting on large terms. In: Nieuwenhuis R (ed) Rewriting techniques and application: 14th international conference (RTA'03: Proc.), Valencia, 9–11 June 2003. Lecture notes in computer science, vol 2706. Springer, Berlin, pp 14–29
9. Ellison C, Serbanuta T, Rosu G (2009) A rewriting logic approach to type inference. In: Proc. of WADT 2008, Pisa, 13–16 June 2008. Lecture notes in computer science, vol 5486. Springer, Berlin, pp 135–151
10. Farzan A, Cheng F, Meseguer J, Roşu G (2004) Formal analysis of Java program in JavaFAN. In: Alur R, Peled DA (eds) Proc. CAV'04: 16th international conference, Boston, 13–17 July 2004. Lecture notes in computer science, vol 3114. Springer, Berlin, pp 501–505
11. Farzan A, Meseguer J, Roşu G (2004) Formal JVM code analysis in JavaFAN. In: Proc. AMAST'04: 10th international conference, Stirling, 12–16 July 2004. Lecture notes in computer science, vol 3116. Springer, Berlin, pp 132–147
12. Martí-Oliet N, Meseguer J (2002) Rewriting logic: roadmap and bibliography. *Theor Comput Sci* 285:121–154
13. Meseguer J (1992) Conditional rewriting logic as a unified model of concurrency. *Theor Comput Sci* 96(1):73–155
14. Meseguer J (1996) Rewriting logic as a semantic framework for concurrency: a progress report. In: Proc. CONCUR'96: 7th international conference, Pisa, 26–29 Aug. 1996. Lecture notes in computer science, vol 1119. Springer, Berlin/New York, pp 331–372
15. Meseguer J, Goguen J (1985) Initiality, induction and computability. In: Nivat M, Reynolds J (eds) Algebraic methods in semantics. Cambridge University press, Cambridge, pp 459–541
16. Meseguer J, Roşu G (2004) Rewriting logic semantics: from language specifications to formal analysis tools. In: Basin D, Rusinowitsch M (eds) Proc. Int'l. Joint Conf. on Automated Reasoning IJCAR'04, Cork, 4–8 July 2004. Lecture notes in computer science, vol 3097. Springer, Berlin, pp 1–44
17. Meseguer J, Roşu G (2007) The rewriting logic semantics project. *Theor Comput Sci* 373:213–237
18. Roşu G, Ellison C, Schulte W (2011) Matching logic: an alternative to Hoare/Floyd logic. In: AMAST 2010: 13th international conference, Lac-Beauport, 23–25 June 2010. Lecture notes in computer science, vol 6486. Springer, Berlin
19. Roşu G, Serbanuta T (2010) An overview of the K semantic framework. *J Log Algebra Program* 79(6):397–434
20. Serbanuta T, Roşu G, Meseguer J (2009) A rewriting logic approach to operational semantics. *Inf Comput* 207(2):305–340

## Media Extensions

- [Vector Extensions, Instruction-Set Architecture \(ISA\)](#)

## Meiko

JAMES H. COWNIE<sup>1</sup>, DUNCAN ROWETH<sup>2</sup>

<sup>1</sup>Intel Corporation (UK) Ltd., Swindon, UK

<sup>2</sup>Cray (UK) Ltd., UK

## Synonyms

[Computing surface; CS-2](#)

## Definition

Meiko was a UK-based parallel computer company that was founded in 1985 by six employees of Inmos. It

grew to a peak of 150 employees based in Bristol and Boston Massachusetts. It ceased to trade in 1996; shortly afterward, some of the communications technology and a small engineering team were transferred to Quadrics.

## Discussion

Meiko machines were all distributed memory clusters. There were three distinct machine generations during the 10 years of Meiko's life: Transputer systems, hybrids that used Transputers for communication and an Intel i860 or SPARC processor, and, finally, SPARC systems in which the Transputers had been replaced by Meiko designed communications ASICs.

At the time of their construction, the takeover of high performance computing by cluster machines had yet to take place, the fastest machines at the time being vector supercomputers from Cray and Japanese vendors. Meiko machines contributed to the demonstration that it was possible to use cluster machines with no shared memory for high performance computing.

## Customers

Meiko had over 400 different customers, some of the more important are mentioned here.

## HPC Users

Major users of Meiko machines for high performance scientific computing included

- CERN who used the machines in the NA48 experiment data recording (a project continued by Quadrics)
- Edinburgh Parallel Computer Centre who used the machines for general scientific supercomputing, and QCD calculations
- Hewlett Packard printer division who used the machines for simulation of inkjet printer heads
- Lawrence Livermore National Laboratory who used the machines in the stockpile stewardship program
- Southampton University who used the machines in the EECS department
- Toyota who used the machines for producing photo-realistic, ray traced images of cars from design data

Many other customers in Europe also used the machines, some of which were funded as part of

the EEC's Esprit program, the GPMIMD project in particular.

## Transputer-Based Machines

Meiko's first-generation machines, known as "The Computing Surface," were based on the Inmos T414 Transputer. Early systems consisted of up to 160 32-bit integer processors connected by 20MBit/s serial links. They were programmed in Occam. With the 1987 introduction of the Inmos T800 Transputer, which had an integrated FPU, floating point performance increased to 2MFlops per CPU, 320 MFlops for the system as a whole. Customers wanted to be able to program these machines in conventional programming languages, so that existing codes could be more easily ported to the machine. Recognizing this, Meiko produced Fortran 77 and C compilers for the Transputer. In these languages, communication was achieved by calls to message passing libraries that mapped directly onto the underlying Transputer hardware channel communication instructions.

These early machines were normally connected to a host (a DEC micro-VAX, or Sun server), which booted the user program directly into the Computing Surface, and provided I/O. These machines were therefore really attached compute servers, rather than stand-alone computers.

In the initial T414 machines, the topology of the Transputer link network was set by plugging wires in the back of the machine. Later machines used a Meiko designed ASIC to allow the link configuration and hence the network topology to be changed between jobs. The topology of the machine was, however, fixed for the duration of a single job – Inmos added adaptive routing support to later generations of product. Since Transputers had no memory protection, or operating system, multitasking was achieved by physically partitioning the machine and running separate jobs on separate processors.

## Hybrid Machines

When Intel released the i860 (80860) processor in 1989, Meiko built boards that used these in conjunction with Transputers. The main program executed on the i860, with communication handled by the Transputers. While the peak performance of the system was high (60-80 MFlops per processor) achieving this proved

difficult, with some hand coded assembler routines achieving 30–40 MFlops but most compiled code getting less than 10.

Meiko also produced SPARC boards with Transputer communications. These were normally used to replace an external host machine, running SunOS (and later Solaris). These boards could all be used in machines with Transputers as nodes, providing a simple upgrade path for existing customers.

Inmos' follow up Transputer design, the T9000, was not a success and Meiko switched to designing its own interconnect for use with conventional microprocessors.

### Sun SPARC-Based Machines

In 1993, Meiko introduced the “Computing Surface 2” (CS-2) machines. These were based on 40 MHz SPARC processors with Meiko's own interconnect, using two custom ASICs, the Elan network interface and the Elite switch. The CS-2 system had an asymptotic bandwidth of 42 MBytes per second per link, a put latency of 10 µs and a message passing latency of 80 µs.

The communication network used by the CS-2 was built from Elite switches, each of which was a sixteen-way full crossbar. Since the links were unidirectional, these provided an eight-way bidirectional crossbar. The network was source routed, with each NIC being able to select at random (or in round robin order) from a table of four possible routes to each destination. Packets in the network were routed by byte-stripping, each Elite stripping the first byte from the incoming message and using that to determine the outgoing route. The Elite chip also included the ability to broadcast an incoming message to a contiguous range of outgoing links, and to combine acknowledgments from a range of links into a single message.

In the CS-2 machine, the Elite switches were connected in a full fat-tree configuration to provide a scalable interconnect with linear increase in bisectional bandwidth and logarithmic increase in latency as the machine size increased.

In addition to the SPARC boards, Meiko produced a board using a SPARC processor and a pair of Fujitsu vector processors. The node had a double precision peak performance of 100 MFlop with 128 MBytes of

memory and 1.2 GB/s of memory bandwidth per vector unit (three words per flop!) and an additional 800 MB/s port shared by the SPARC and the Elan. The high peak performance was attractive (particularly to LLNL, the first customer) but the node design complicated the programming model – a new vectorizing compiler was developed by The Portland Group. In the initial design, the vector processors were not cache coherent with the SPARCs, despite sharing memory with them. This made programming hard, or forced the SPARC processor to execute with most of the memory marked as uncachable. Of course, this slowed the non-vectorizable portions of the code, making overall performance poor. (A simple application of Amdahl's Law with a slowdown applied to the non-vectorized component shows that if the slowdown is  $S$ , even with an infinitely fast vector system a proportion  $(1 - 1/S)$  of the program must vectorize to achieve parity with the scalar system; for a slowdown of  $10\times$  this leads to a requirement that more than 90% of the computation vectorizes). Later versions implemented cache coherence, but the complexity of the logic and PCB manufacture required made the design uneconomic. Meiko had a competing board design that used four SPARC CPUs. It was lower cost, much easier to program and generally sustained higher performance, but it lacked the high floating point peak and only sold to a small number of customers.

The CS-2 system pictured comprised 24 modules each with four boards. It was typically configured with either 64 vector or 256 scalar nodes together with I/O card and switching. The largest system shipped (to LLNL) consisted of five such racks with a total of 224, mostly vector, nodes.

A fourth-generation system based on SPARC processors and a new network was under development in 1995 when the company ran out of money. Significant elements of this design were used in QsNet, although Quadrics moved away from the SPARC/MBUS design to the recently standardized PCI host interface. Ever since then, network adapters for HPC systems have been on “wrong side” of a commodity host interface with reduced bandwidth and increased latency as a consequence.



**Meiko.** Fig. 1 Meiko CS-2 “Hugh” at Lawrence Livermore National Lab (Photograph courtesy of Lawrence Livermore National Laboratory)

M

## Uses Outside High Performance Computing

Meiko worked with Oracle to port their database (Oracle 7 Parallel Server) to the CS-2, and a number of database server machines were sold late in the life of the company. These systems were used on early data warehousing applications by UK retailers including WH Smiths and Bass Taverns. Sales data was transferred from the company mainframe overnight enabling marketing users to develop and test complex decision support queries on recent data. Other users ran a mix of online transaction processing (OLTP) and decision support on their CS-2 systems. Meiko was unable to capitalize on this product – customers for database systems of its class were not generally willing to buy them from a small UK company.

## Influence

The experience of the Meiko designs has spread widely. In 2009, ex-employees of Meiko are to be found at companies such as AMD, Cray, HP, Intel, and Sun. They

have influenced designs as diverse as AMD’s “HyperTransport” interconnect, current database machines, and parallel file system design. For instance, the technical lead for the Lustre parallel file system product at Sun is one of the founders of Meiko who led the parallel Oracle project 15 years earlier.

## Contributions

### Direct User Space Communication

The Meiko CS-2 machine was one of the first machine designs to implement direct, cache-coherent, secure multi-user communication from user address space to user address space with no OS intervention. Other contemporary machines like the Thinking Machines CM-5 (also based on SPARC chips with a “fat-tree” interconnect) allowed only single user use of the compute nodes, did not execute a full OS on the node, and used the CPU to transfer data.

Avoiding OS intervention on message passing is critical if low latency is to be achieved, as simply entering and leaving kernel mode is expensive. Since

most message passing codes have many small messages, achieving high compute performance requires low-latency message passing.

OS intervention is normally required in I/O operations for these reasons

- Protection: to ensure that the user is not attempting to transmit (or receive into) parts of their address space which are unmapped, or that the user is not attempting to access a device which they have no permission to access.
- Pinning: to ensure that the physical address of data which is being communicated does not change while the communication is occurring, since DMA I/O devices traditionally deal with physical addresses.
- Device Sharing: to ensure that multiple users do not incorrectly interleave access to the same physical device, all access is serialized through the OS device driver.

The Elan communications interface was designed to avoid these issues. The protection and pinning issues were avoided by giving the Elan its own set(s) of page tables, so that all requests (both local and remote) could be made in terms of user virtual addresses. The Elan could then convert these into physical addresses, or (if the page was not present), raise a local page fault to cause the OS to load the page (if it was paged out), or kill the process (if the access was invalid). The page tables in the Elan were maintained by the Solaris kernel as it modified its own page tables.

Device sharing and security were achieved by giving the Elan multiple, duplicate, copies of its control registers at different physical addresses. When a process initially requested access to the Elan, the resource manager would allocate a set of control registers to the process, and map them directly into the process' address space. It would also program the Elan so that it knew which page tables to use when it received a request through that set of control registers, and what global identifier it should use to identify incoming and outgoing messages from that process. Thus when a remote request arrived at the Elan it could match the global identifier and route it into the correct local process' address space with no OS intervention.

The Elan chip sat on the SPARC MBUS, so could participate in the cache-coherency traffic in the local node. There was therefore no requirement that data to

be transmitted be flushed to memory, or that data must be received into uncached store.

To eliminate the involvement of the CPU in processing message protocols, the Elan include its own processor (which executed a subset of the SPARC instruction set) and its own DMA engine. Using the techniques already outlined, this processor could execute code in the context of a user process, without requiring that the code be in a kernel-space device driver. Thus, each parallel job in the system could use whatever protocol it wished, the code to implement it being loaded as a normal shared library.

### **Remote Memory Access**

The fundamental communication primitive provided by the Elan was a remote memory access (put or get). This allows a less synchronized style of communication than message passing, since communication occurs when the initiator of the request requires it, it does not require any cooperation from the target. This leads to a programming style similar to that of the partitioned global address space languages today (UPC, Co-Array Fortran, Titanium), rather than a message passing style.

The message passing libraries (such as MPI) were built on top of the remote memory access, rather than building remote memory access on top of message passing, as one would do in an active message style system.

However, the model here differs from a fully shared memory system because remote memory is not available to the local processor directly through normal load and store operations. Rather accesses to remote memory must be explicitly requested from the network interface. Communication here is thus explicit, whereas in a shared memory system it is implicit in normal memory accesses.

This model fits well with the “partitioned global address space” languages, since there the compiler always knows whether a memory access is local or remote, and can generate appropriate code to fetch remote data as required.

### **Network Collective Operations**

The Elite chip was designed to support network collective operations, broadcasting packets to a contiguous range of links, combining the results and returning a single answer to the initiating process. This technique

was used to implement fast broadcast and barrier functions, without the use of a separate network as in the CM-5. The combining operation could be used, for instance, to test whether all processes had the same value at a specific virtual address, by sending a single message and receiving a single acknowledgment. The aggregation of the result occurring in the network, rather than requiring aggregation up a tree of processors, with each one combining results from those below it. The root process performed network conditionals until it received the acknowledgment that all processes had entered the barrier. It then broadcast a completion notification. These techniques were developed by Quadrics in future generations of Elite router.

## Software

### Development Tools

For the Transputer machines, Meiko implemented their own development environment, which allowed user code to configure the machine topology and then load code into it.

For the SPARC-based machines, Meiko ported the TotalView debugger at the request of LLNL, and enhanced it to interact with the resource manager of the CS-2. The SPARC systems included an early parallel filesystem implemented as a Solaris virtual filesystem.

Meiko pioneered development of C and Fortran compilers for Transputers and later HPF compilers for the SPARC systems. It supported PVM and PARMACS on the CS-2, contributed to the MPI standard, and the MPICH MPI implementation.

### Operating System

Meiko implemented kernel patches to the Solaris operating system to maintain the Elan page tables in synchrony with the kernel page tables. Quadrics continued to develop these for Linux, and their ultimate descendant was finally adopted into Linus' kernel in 2008.

### Related Entries

- ▶ [CSP \(Communicating Sequential Processes\)](#)
- ▶ [HPF \(High Performance Fortran\)](#)
- ▶ [MPI \(Message Passing Interface\)](#)
- ▶ [PGAS \(Partitioned Global Address Space\) Languages](#)
- ▶ [PVM \(Parallel Virtual Machine\)](#)

## Bibliographic Notes and Further Reading

The definitive papers on the design and programming of the Meiko CS-2 are [2] and [1].

The Inmos Transputer is described in [6].

The Thinking Machines CM-5, which forms an interesting comparison with the Meiko CS-2 is described in [4].

UPC, one of the first PGAS languages is described in [3].

The first message passing interface (MPI) standard is described in [5].

## Bibliography

1. Barton E, Cownie J, McLaren M (1994) Parallel Comput 20: 497–507
2. Beecroft J, Homewood M, McLaren M (1994) Parallel Comput 20:1627–1638
3. El-Ghazawi T, Carlson W, Sterling T, Yelick K (2003) UPC: distributed shared-memory programming. Wiley-Interscience, Hoboken
4. Hillis WD, Tucker LW (1993) Commun ACM 36:31–40
5. MPI (1993) Technical report Knoxville, TN, USA
6. Whitby-Stevens C (1985) In: ISCA '85: Proceedings of the 12th annual international symposium on computer architecture, IEEE Computer Society, Los Alamitos, pp 292–300

## Memory Consistency Models

### ► Memory Models

## Memory Models

SARITA V. ADVE<sup>1</sup>, HANS J. BOEHM<sup>2</sup>

<sup>1</sup>University of Illinois at Urbana-Champaign, Urbana, IL, USA

<sup>2</sup>HP Labs, Palo Alto, CA, USA

### Synonyms

Fences; Memory consistency models; Memory ordering

### Definition

In the context of shared-memory systems, the memory model specifies the values that a shared-memory read in a program may return. It is part of the interface between

a shared-memory program and any hardware or software that may transform that program – it both specifies the possible behaviors of the memory accesses for the programmer and constrains the legal transformations and executions for the implementer.

For high-level language programs, the memory model describes the behavior of accesses to shared variables or heap objects, while for machine code, the memory model describes the behavior of hardware instructions that access shared memory. Without an unambiguous memory model, it is not possible to reason about the correctness of a shared-memory program, compiler, dynamic optimizer, or hardware.

## Discussion

### Sequential Consistency

In a single-threaded program, a read is expected to return the value of the last write to the same address, where last is uniquely defined by the program text or program order. This allows the programmer to view the memory accesses as occurring one at a time (atomically) in program order. It also allows the hardware or compiler to aggressively reorder memory accesses as long as program order is preserved between a write and other accesses to the same address as the write.

A natural extension for shared-memory programs is the sequential consistency memory model [12] which offers simple interleaving semantics. With sequential consistency, all memory accesses appear to be performed in a single total order preserving the program order of each thread. Each read access to a memory location yields the value stored by the last preceding write access to the same memory location. Although sequential consistency appears to be an appealing programming model, it suffers from two deficiencies:

- It can be difficult or inefficient to implement. Hardware often makes memory accesses visible to other threads out of order, for example, by buffering stores. Compilers often do the same, for example, by moving an access to a loop-invariant variable out of the loop. Unlike with single-threaded programs, these optimizations can be observed by the program, and may thus violate sequential consistency for multi-threaded programs.

- The definition of sequential consistency is based on the interleaving of individual memory accesses, and is thus only meaningful if the programmer understands how memory is accessed, for example, a byte versus a word at a time. This is sensible at the machine architecture level, but may not be realistic at the programming language level.

### Relaxed Memory Models

Recognizing the limitations of sequential consistency, more relaxed or weak memory models have been proposed. Until the late 1990s, most of this work was in the context of hardware instruction set architectures and implementations. These implementation-centric models are mostly driven by specific optimizations desirable for hardware. The most common class relaxes the program order requirement of sequential consistency. For example, the SPARC Total Store Order (TSO) [18] and the x86 models [11] allow reordering a write followed by a read (to a different location) in program order. The IBM PowerPC memory model [9] allows reordering both reads and writes (to different locations). Such models additionally provide various forms of fence instructions to enable programmers to explicitly impose program orders not guaranteed by default. Other optimizations exploited by such models include subtle relaxations of the memory model to allow writes from multiple threads to become visible in inconsistent orders to different observer threads, as well as the ability to make memory references visible to other threads out of order even if a data- or control dependence exists between the references. Although most of the work on this genre of implementation-centric models is in the context of hardware, some programming environments such as OpenMP 3.0 [16] also express their memory models in terms of fence instructions to ensure explicit program ordering.

These relaxed models provide performance benefits over sequential consistency (through both hardware and compiler optimizations), but are inappropriate as programming interfaces. They often require description of subtle interactions and/or are not well-matched to software requirements, making them too complex to reason and/or suboptimal for performance.

## The Data-Race-Free Memory Models

The data-race-free models [1, 4] (also referred to as properly labeled models [8]) take a programmer-centric approach. They observe that good programming practice requires programs to be well synchronized or data-race-free. In such programs, concurrent non-read-only accesses to the same variable are either explicitly marked as synchronization or are separated by explicit synchronization. This ensures that no thread can ever observe another thread between synchronization points, allowing the implementation to aggressively transform code between synchronization points while still maintaining the appearance of sequential consistency.

The data-race-free models formally define the notion of data-race-free programs and guarantee sequential consistency only for those programs. For full implementation flexibility, they do not provide any guarantees for programs that contain data races. This approach allows nearly standard optimizations, while providing a simple model for programmers and was the approach adopted by the Ada programming language [17] and the POSIX standard [10] (albeit informally).

## State-of-the-art in 2010

In the last decade, there has been much work to standardize high-level language programming models, specifically in the context of Java and C++. There is now convergence toward data-race-free as the model of choice. For all practical purposes, the Java memory model [13, 14] and the upcoming C and C++ standards [6, 7] provide the data-race-free model, but with two unfortunate caveats.

First, Java's safety and security properties require that some semantics need to be provided for all programs, including programs with data races. It has been extraordinarily difficult to develop reasonable semantics for racy programs, while still preserving full implementation flexibility. The current formal specification of the Java memory model is therefore quite complex and has an unresolved bug.

Second, although data-race-free maps well to current implementation-centric hardware models for most purposes, there are useful programming idioms where available hardware-level fences are too heavyweight for

language-level synchronization mechanisms. For this reason, Java and C++ provide low-level synchronization constructs that provide potentially better performance on some current hardware, but at the cost of significant complexity to the programming model.

Although most language specifications are converging to the data-race-free model, current implementations often lag. In particular, it is still quite common for C and C++ compilers to effectively introduce copies of an otherwise unmodified variable to itself, for example, by overwriting adjacent structure fields when a small field is modified [5]. These can introduce visible data races, and are hence incompatible with the model.

In summary, although data-race-free provides the best trade-off between performance and ease-of-use today, it falls short for safe languages and current implementations.

## Bibliographic Notes and Further Reading

A more complete overview and retrospective on memory models is provided in [2]. It also surveys some possible approaches for overcoming the deficiencies of current data-race-free models. A tutorial on hardware-oriented relaxed memory models can be found in [3]. A description of the x86 hardware memory model, together with a discussion of some of the issues arising in precisely defining such models, is given in [15].

## Bibliography

1. Adve SV (1993) Designing memory consistency models for shared-memory multiprocessors. Ph.D. thesis, University of Wisconsin-Madison
2. Adve SV, Boehm HJ (2010) Memory models: a case for rethinking parallel languages and hardware. Commun ACM 53(8):90–101
3. Adve SV, Gharachorloo K (1996) Shared memory consistency models: a tutorial. IEEE Comput 29(12):66–76
4. Adve SV, Hill MD (1990) Weak ordering – a new definition. In: Proceedings of 17th international symposium on computer architecture, Seattle, pp 2–14
5. Boehm HJ (2005) Threads cannot be implemented as a library. In: Proceedings of conference on programming language design and implementation, Chicago
6. Boehm HJ, Adve SV (2008) Foundations of the C++ concurrency memory model. In: Proceedings of conference on programming language design and implementation, Tucson, pp 68–78

7. C ++ Standards Committee, Becker P (2010) Programming Languages – C ++ (final committee draft). C ++ standards committee paper WG21/N3092=J16/10-0082, <http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2010/n3092.pdf>, March 2010
8. Gharachorloo K (1995) Memory consistency models for shared memory multiprocessors. Ph.D. thesis, Stanford University
9. IBM Corporation (2010) Power ISA Version 2.06 Revision B. [http://www.power.org/resources/downloads/PowerISA\\_V2.06B\\_V2\\_PUBLIC.pdf](http://www.power.org/resources/downloads/PowerISA_V2.06B_V2_PUBLIC.pdf), 2010
10. IEEE and The Open Group (2001) IEEE Standard 1003.1-2001
11. Intel Corporation (2010) Intel 64 and IA-32 Architectures software developer's manual, vol 3a, system programming guide, Part1. <http://www.intel.com/products/processor/manuals/>, 2010
12. Lamport L (1979) How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans Comput C-28(9):690–691
13. Manson J, Pugh W, Adve SV (2005) The Java memory model. In: Proceedings symposium on principles of programming languages, Long Beach
14. Pugh W, JSR 133 Expert Group (2009) The Java memory model. [http://www.cs.umd.edu/\\_pugh/java/memoryModel/](http://www.cs.umd.edu/_pugh/java/memoryModel/) and referenced pages, July 2009
15. Sewell P, Sarkar S, Owens S, Nardelli FZ, Myreen MO (2010) x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Commun ACM 53(7):89–97
16. The OpenMP ARB (2008) OpenMP application programming interface: Version 3.0. <http://www.openmp.org/mp-documents/spec30.pdf>, May 2008
17. United States Department of Defense (1983). Reference manual for the Ada programming language: ANSI/MILSTD-1815A-1983 standard 1003.1-2001, Springer
18. Weaver DL, Germond T (1994) The SPARC architecture manual, version 9. Prentice Hall, Englewood Cliffs

## Memory Ordering

► [Memory Models](#)

## Memory Wall

SALLY A. MCKEE<sup>1</sup>, ROBERT W. WISNIEWSKI<sup>2</sup>  
<sup>1</sup>Chalmers University of Technology, Goteborg, Sweden  
<sup>2</sup>IBM, Yorktown Heights, NY, USA

## Synonyms

[Data starvation crisis](#)

## Definition

The memory wall describes implications of the processor/memory performance gap that has grown steadily

over the last several decades. If memory latency and bandwidth become insufficient to provide processors with enough instructions and data to continue computation, processors will effectively always be stalled waiting on memory. The trend of placing more and more cores on chip exacerbates the situation, since each core enjoys a relatively narrower channel to shared memory resources. The problem is particularly acute in highly parallel systems, but occurs in platforms ranging from embedded systems to supercomputers, and is not limited to multiprocessors.

## Discussion

### Introduction

The term memory wall was coined in a short, controversial note that William A. Wulf and Sally A. McKee published in a 1995 issue of the *ACM SIGArch Computer Architecture News* (“Hitting the Memory Wall: Implications of the Obvious” [1]). At the time, most computer architects focused entirely on increasing processor speed, believing that improving and creating more latency tolerating techniques would suffice to keep these ever-faster processors fed with instructions and data. The High Performance Computing (HPC) community had already recognized the problem, but 5 years after John Ousterhout published an article entitled “Why Aren't Operating Systems Getting Faster As Fast as Hardware?” [5], the mainstream computing community still believed that out-of-order execution, wider instruction issue widths, non-blocking caches, and increased speculation would continue to deliver performance improvements through increased Instruction Level Parallelism (ILP) and latency tolerance. Even Richard Sites's article in the popular magazine *Microprocessor Report* (“It's the Memory, Stupid!”) [7] failed to change common opinion.

In the memory wall article, the authors make the point that if the entire memory hierarchy, and not just the caches, were not improved in tandem with the microarchitecture, computer performance would cease to improve. They assume that every fifth instruction is a memory reference and that the number of cycles required to fill a cache line grows over time (looking at two different ratios of processor versus DRAM speedups, cache hit of 90% up to 99.8%, and two initial cache miss/hit cost ratios). By graphing the number of cycles it takes to perform a cacheline

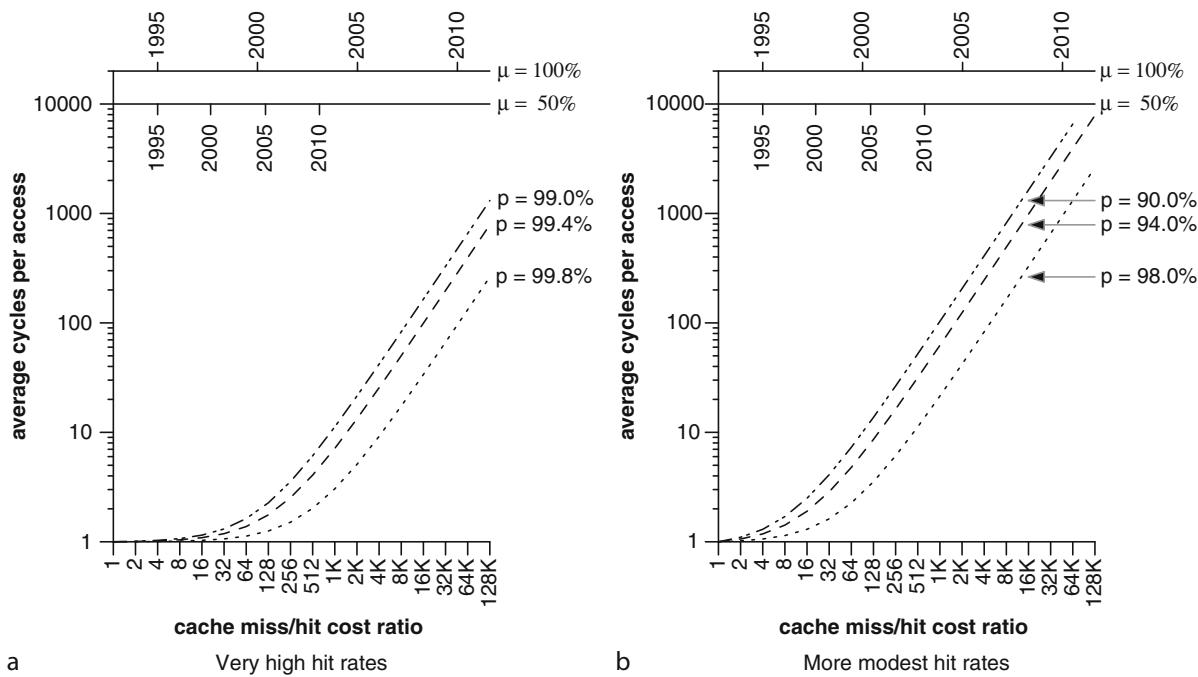
fill, they show that even at impossibly high hit rates and a less aggressive speed growth differential, processor improvements would soon no longer matter, since the cores would find themselves almost always waiting on memory. Their analysis necessarily contains many simplifying assumptions, but the four-page note eventually had impact, and memory wall has become a common term.

Figures 1 and 2 show the average number of cycles required per memory access as cache miss/hit cost ratios rise. The  $x$  axis marks increases in these cost ratios assuming given starting points (i.e., a cache for which fills cost four times as much as hits, and one for which fills cost 16 times as much). The log-based  $y$  axis indicates cycles per access. The curves in each graph indicate average access costs for different (unrealistic) cache hit rates from 90.0% to 99.8%. The lines at the top of each graph indicate where on the cycle-cost curves we would be at different points in time, where the top line marks years for an aggressive improvements (100%) in processor speed every 18 months, and the bottom marks years for a more modest improvement rate (50%). As the original article says, “the specific values won’t change the basic conclusion of this note, namely that we are

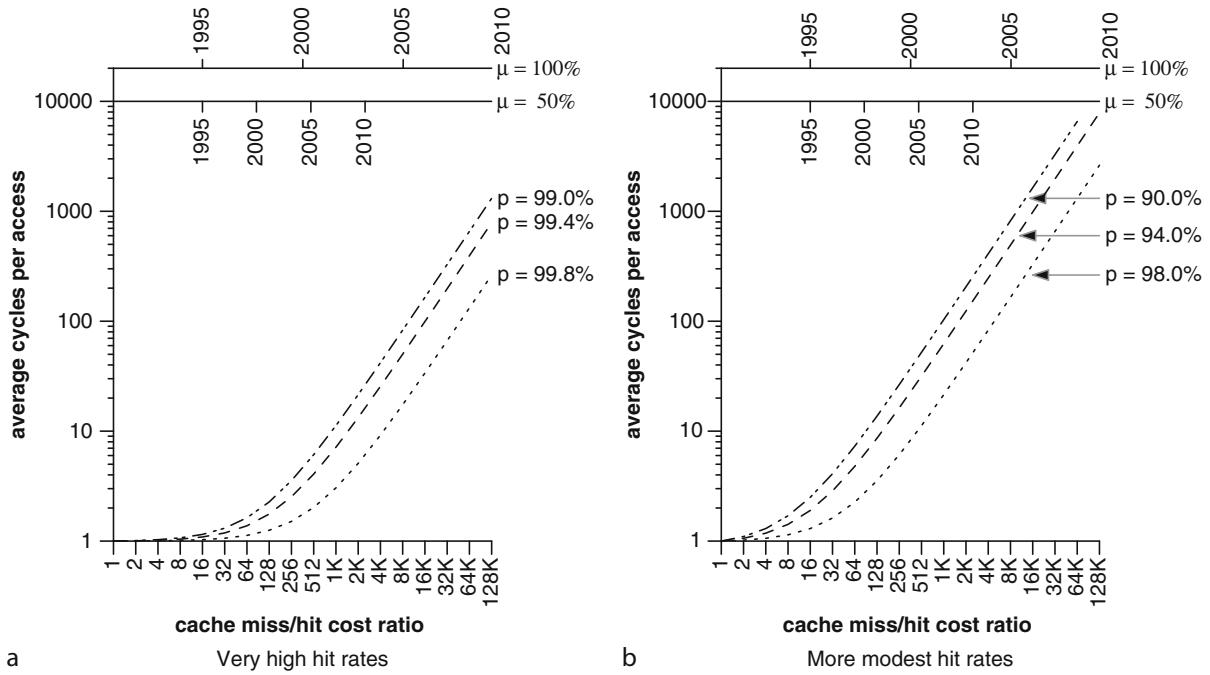
going to hit a wall in the improvement of system performance unless something basic changes” [11].

The memory wall also has a bandwidth component. One aspect of that is the limited number of pins that can be placed on ever-shrinking chips (formerly on the perimeter [1], now on the surface). The 1995 note focuses on latency aspects of the memory wall, but latency and bandwidth are two sides of the same coin, and cannot be teased apart. Denali CEO Sanjay K. Srivastava again stated the obvious in a 2001 interview with Wall Street Reporter: “There has been a growing disparity between the bandwidth needs of the processing elements and the ability of the memory elements to deliver that bandwidth. This ‘data starvation crisis’ is now being recognized as the main bottleneck for all electronic system designs” [8].

Depending on application characteristics and the architectures on which the applications run, current programs may or may not experience a memory wall. Data access locality on many levels is important to good performance. Applications that access sparse data structures, exhibit random (as opposed to structured) access patterns, or have huge working sets that prevent them from enjoying high cache hit rates suffer most.



**Memory Wall.** Fig. 1 Trends for a 1995 cache miss/hit cost ratio of four [11]



**Memory Wall.** Fig. 2 Trends for a 1995 cache miss/hit cost ratio of sixteen [11]

## DRAM Basics

Dynamic Random Access Memory (DRAM) is used in almost all main memories, as opposed to Static Random Access Memory (SRAM), which is used in almost all caches in the memory hierarchy. The term DRAM was coined to indicate that the technology allowed any (random) access to be performed in about the same amount of time, in contrast to tape storage and disks. In spite of the name, some accesses take longer than others. Improvements in DRAMs have largely increased storage capacity, rather than increasing speed, hence the processor/memory performance gap.

DRAMs are organized into internal banks, and each bank, or storage array, is accessed by specifying a row and a column address. Separate wires are used to transmit commands, addresses, and data. When the memory controller issues a row address, the specified line is loaded from the storage array into a bank of sense amplifiers (also called a DRAM *page* or *hot row*) that boosts the signal but also behaves something like a line of cache. The data in the “open” page remains valid long enough to perform many accesses, and consecutive accesses that hit in the same row happen much faster than accesses to different rows. Exploiting reference

locality can thus improve memory performance in these Fast Page Mode (FPM) DRAMs. Reading the row from the storage array is destructive, and so before another row can be accessed, the current row must be written back, which is called “closing” the row. The sense amplifiers must then be recharged before another row can be read. Memory systems can be organized with either open or closed page policies. In the former, data are left in the sense amplifiers in hopes that subsequent accesses will exhibit enough locality to be satisfied by the open row. In the latter, the active page is closed after every access. Using a closed page policy is simpler and provides more predictable access times, and thus many memory controllers close the current page after every cacheline fill.

Memory performance can also be improved by increasing bandwidth and using more chips in parallel. Separate chips are referred to as ranks. One common current packaging format uses Dual Inline Memory Modules (DIMMs), which have separate electrical connections on both sides of the printed circuit boards containing the DRAM circuits. Since the basic DRAM storage technology changes little with time (a bit is simply a capacitance stored at the intersection

of wires in the array), most improvements in DRAM performance come from innovations in the interface through which the memory controller communicates with the chip. DRAM chips were originally asynchronous, requiring a handshake protocol to coordinate data transmission across the memory bus. Eliminating asynchrony (so each action must happen after a specified number of memory clock cycles) and pipelining commands allows multiple accesses to be in flight at a time in Synchronous DRAMs (SDRAMs). Extended Data Out (EDO) DRAMs allow slightly more parallelism: The data output of a read is maintained while the command for the next access is being sent. One of the biggest performance improvements comes from increasing the speed at which data are transmitted. Double Data Rate (DDR) DRAMs transmit data on both the rising and falling edges of the memory clock. DDR2 halves the clock cycle to provide four clock edges per memory clock cycle, and DDR3 divides it further.

Bruce Jacob et al. [2] provide a good reference for all levels of the memory hierarchy. The bottom line is that changing the DRAM interface to incorporate synchrony, pipelining of data and commands, and double data rate transmission have had a significant impact on the design of memory system back ends. In spite of these improvements, DRAMs still cannot deliver the performance required to keep cores from stalling frequently. Given that HPC applications suffer most seriously from the memory wall phenomenon, the rest of this discussion is presented in that context.

## HPC Software Implications of Memory Trends

Many types of applications suffer from inadequate memory performance, but the trends that created conditions leading to the memory wall have historically had the largest impact on HPC applications. These trends cause two broad categories of memory challenges. A confluence of economic and technological factors drives the use of less overall memory for a given amount of computation and the use of less memory bandwidth for the delivered computation *FLOPS* (Floating Point Operations per Second).

## Existing Model for HPC Applications

Shared memory paradigms such as OpenMP, UPC (Universal Parallel C), pthreads, Global Arrays, X10,

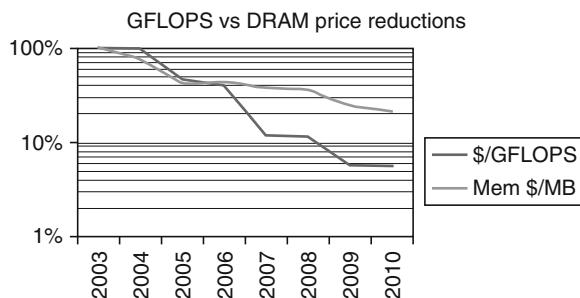
Fortress, and Charm++ are coming into greater use. Nonetheless, the predominant programming model remains single-threaded MPI (Message Passing Interface). Reduced amounts of memory combined with increased core counts cause the standard model of a single thread per MPI task to require re-examination, and the reduced bandwidth will in turn cause application programmers to redesign the algorithm decomposition of their problems. A common program structure used by MPI programs alternates between computation and communication phases. All tasks in the application synchronize via either reduction operations or explicit barriers. The tasks proceed through each of these phases in lock step. Two consequences of this model are pertinent to memory effects. These are highlighted here and explored in detail below.

The first consequence is that as memory per computation core is reduced, then for a given single-threaded MPI task, the amount of data on which that task can compute is reduced. This, in turn, can potentially imply that either the data needed by the algorithm decomposition no longer fit in memory, or that the amount of computation per phase is reduced. The reduction of computation per phase causes additional load balancing challenges: Tasks spend a greater percentage of time communicating instead of doing useful computation, thereby reducing the overall efficiency of the algorithm.

The second consequence is that as the ratio of memory bandwidth per FLOP shrinks, the percentage of time the application spends moving data to and from memory grows. Though communication/synchronization time is inevitable with decreasing bandwidth ratios, this limitation is imposed in part by the MPI model, which prevents the application from leveraging more efficient intra-node optimizations.

## Amount of Memory

Figure 3 shows the increasing cost of memory relative to the cost of computing. This trend has led to less available memory for a given amount of computation on large supercomputers. Although there are some technologies on the horizon that may provide a different cost point — for example, PCM (Phase Change Memory) — there will be challenges in effectively utilizing them. More importantly, even if designers can incorporate them, they will only provide a one-time improvement in the cost ratio. Overall, the community consensus holds that



**Memory Wall.** Fig. 3 Memory cost trends

even in the presence of occasional improvements in the memory/computation cost ratio, the general trend will continue, and the gap will widen.

### Memory Bandwidth

The trends in memory bandwidth are at least as problematic. Though improvements have been made in terms of the number of pins and later balls, there is still a physical limitation to the size and thus number of off-chip connections. As with the amount of memory there are a couple technologies on the horizon, such as chip stacking, that has a potential to provide a one time gain. However, as with the amount of memory, consensus among experts indicate that trend will continue. Though not memory bandwidth related, there is an increasing gap in communication bandwidth versus computation power. This trend leads to similar challenges. The effect of decreasing memory bandwidth per computation is examined in the next section.

### Future Directions

U.S. Department of Energy roadmaps project that the next generation of supercomputers will see  $O(1M)$  cores with the count climbing to a projected  $O(100M)$  cores in the exascale era near the end of the decade (2018–2019). One implication of all these memory trends is that the core algorithms in HPC applications must be redesigned to handle not just the challenges introduced by memory trends but also the scale of the supercomputers of coming generations. Although researchers recognize the necessity of doing this, it is a daunting task, and in many areas there has not been

significant progress. More work has been applied on the programming model front.

The trends provide a strong indication that most applications written in a flat (only MPI), single-threaded programming model will not be able to take advantage of the computing power that next-generation and coming exascale machines will offer. Many programmers have recognized this, and some are in the process of modifying applications. Many different avenues are being pursued, but most are based on executing more than a single software thread of execution within a node. This strategy directly attacks the challenge of amount of memory by providing many threads of computation per given memory. The most prevalent option is combining OpenMP with MPI. While this is a readily available model, researchers have already identified difficulties with the existing specification.

The ability to run multiple threads within a given MPI task indirectly addresses the memory bandwidth and communication bandwidth challenges. Such a strategy releases the threads running within the MPI task from being held to MPI algorithm limitations of interaction. For example, threads can take advantage of shared memory, and (more importantly) shared L2 or even L1 caches, or other intra-chip optimizations such as atomic operations. The disadvantage of this two-tiered or hybrid programming model is its increased complexity. Programming languages such as UPC, X10, Chapel, and Fortress have been introduced to address this complexity, providing an integrated programming model across the entire system. They explicitly support multiple levels of parallel execution, and recognize that memory access is non-uniform: Required data will be stored physically closer to some threads than others (and thus data locality becomes important on yet another level).

It seems likely that truly addressing all these trends and the challenges they impose will benefit from a new programming paradigm. The challenge is analogous to the transition to MPI: Changes must be embraced by the community at large. Porting applications to new programming models is extremely expensive, and will only be undertaken once sufficient confidence in an emergent programming model exists. Memory trends must be addressed in both hardware and software by a unified community.

## Related Entries

- ▶ [Buses and Crossbars](#)
- ▶ [Cache Coherence](#)
- ▶ [Chapel \(Cray Inc. HPCS Language\)](#)
- ▶ [CHARM++](#)
- ▶ [Cilk](#)
- ▶ [Coarray Fortran](#)
- ▶ [Cray T3E](#)
- ▶ [Cray Vector Computers](#)
- ▶ [Cray XMT](#)
- ▶ [Cray XT3 and Cray XT Series of Supercomputers](#)
- ▶ [Cray XT4 and Seastar 3-D Torus Interconnect](#)
- ▶ [Denelcor HEP](#)
- ▶ [Earth Simulator](#)
- ▶ [Exascale Computing](#)
- ▶ [Fortran 90 and Its Successors](#)
- ▶ [Fortress \(Sun HPCS Language\)](#)
- ▶ [Functional Languages](#)
- ▶ [Green Flash: Climate Machine \(LBNL\)](#)
- ▶ [IBM Blue Gene Supercomputer](#)
- ▶ [Instruction-Level Parallelism](#)
- ▶ [Vector Extensions, Instruction-Set Architecture \(ISA\)](#)
- ▶ [Intel Core Microarchitecture, x86 Processor Family](#)
- ▶ [Latency Hiding](#)
- ▶ [MPI \(Message Passing Interface\)](#)
- ▶ [Nonuniform Memory Access \(NUMA\) Machines](#)
- ▶ [OpenMP](#)
- ▶ [Power Wall](#)
- ▶ [Processors-in-Memory](#)
- ▶ [PVM \(Parallel Virtual Machine\)](#)
- ▶ [Reconfigurable Computers](#)
- ▶ [Roadrunner Project, Los Alamos](#)
- ▶ [Shared-Memory Multiprocessors](#)
- ▶ [SoC \(System on Chip\)](#)
- ▶ [Top500](#)
- ▶ [Transactional Memories](#)
- ▶ [UPC](#)
- ▶ [Vector Extensions, Instruction-Set Architecture \(ISA\)](#)

## Bibliographic Notes and Further Reading

Publication of “Hitting the Memory Wall: Implications of the Obvious” inspired a flurry of responses, both in private email and in print. Most of these put forth arguments for why the memory wall did not and would

not exist (for example, see Maurice Wilkes’s “The Memory Wall and the CMOS End-Point” [10], also in *ACM SIGArch Computer Architecture News*). A simple online search yields many such articles, and it is interesting to review the numerous then-popular assumptions and to examine which ones have held over time.

Chun Liu et al. [3] discuss the problem in the context of the chip multiprocessors (CMPs) that began to emerge over the past decade. Thus, in spite of the community’s original skepticism about the very existence of the problem, a few researchers have continued to address different aspects of the memory wall, keeping the term in common computer parlance.

More recently, Richard Murphy [4] examines memory performance (both latency and bandwidth) for traditional and emerging supercomputer applications, finding that performance for both kinds of workloads is dominated by the memory system, but that the emerging applications are even more sensitive to memory latency and bandwidth limitations than their traditional counterparts. He notes that for massively parallel processors, the memory wall manifests itself due to the limited number of independent channels to memory compared to increases in numbers of cores. Thus, the problem predicted by Doug Burger et al. [1] in their 1996 article, “Memory Bandwidth Limitations of Future Microprocessors,” has been realized for this class of computers. Samuel K. Moore emphasizes Murphy’s findings with respect to informatics codes in a short *IEEE Spectrum* article, “Multicore Is Bad News for Supercomputers.” He quotes Sandia National Laboratory’s director of computation, James Peery: “After about eight cores, there’s no improvement.” Nonetheless, the debate regarding the conclusions of both articles continues: For example, see Ars Technica [9] for lengthy conversations on the memory wall for all types of CMP systems.

Brian Rogers et al. [6] developed an analytic model of CMP memory bandwidth to analyze constraints imposed by gap between increasing numbers of cores and limited memory bandwidth, and they used this model to assess the likely effectiveness of combinations of known techniques such as link compression, cache compression, DRAM caches, and stacked (3D) caches, concluding that by using appropriate combinations of approaches to reduce main memory bandwidth

requirements, the number of cores on chip is likely to reach 24–40 or higher (for future technology generations). The philosophy behind this work is the classic one that has allowed us to address the memory wall problem as well as we have thus far: We must use every technique available, and every combination thereof, to overcome the memory wall. Rogers et al. project that in this way, CMPs four technology generations from now may support over 180 cores. At that point, cost, power, and thermal considerations are likely to represent even larger walls. The interdependences of this confluence of problems place even greater constraints on how any individual component can be addressed: The memory wall is no longer just about latency and bandwidth, but about cost, power, area, and temperature, and thus will likely remain a prominent problem in computer system design for generations to come.

## Bibliography

1. Burger D, Goodman J, Kägi A (1996) Memory bandwidth limitations of future microprocessors. In: Proceedings of the 23rd International Symposium on Computer Architecture, Philadelphia, 22–24 May 1996. IEEE/ACM, Los Alamitos/New York, pp 78–89
  2. Jacob B, Ng S, Wang D (2007) Memory systems: cache, DRAM, disk, 1st edn. Elsevier/Morgan Kaufmann, Burlington/San Francisco
  3. Liu C, Sivasubramaniam A, Kandemir M (2004) Organizing the last line of defense before hitting the memory wall for CMPs. In: Proceedings of the 10th IEEE Symposium on High Performance Computer Architecture, Madrid, 14–18 Feb 2004. IEEE, Los Alamitos, pp 176–185
  4. Murphy R (2007) On the effects of memory latency and bandwidth on supercomputer application performance. In: Proceedings of the IEEE International Symposium on Workload Characterization, Boston, 27–29 Sept 2007. IEEE, Piscataway, pp 35–43
  5. Ousterhout J (1990) Why aren't operating systems getting faster as fast as hardware? In: Proceedings of the Summer USENIX Technical Conference, June 1990, pp 247–256
  6. Rogers B, Krishna A, Bell G, Vu K, Jiang X, Solihin Y (2009) Scaling the bandwidth wall: challenges in and avenues for CMP scaling. In: Proceedings of the 36th International Symposium on Computer Architecture, Austin, 20–24 June 2009. IEEE/ACM, Los Alamitos/New York, pp 371–382
  7. Sites R (1996) It's the memory, stupid. Microprocessor Rep 10(10):2–3
  8. Srivastava S (2001) CEO interview. Wall Street Reporter, Aug 2001
  9. Stokes J (2008) Analysis: more than 16 cores may well be pointless. In: Ars Technica. Condé Nast Digital. Available <http://arstechnica.com/hardware/news/2008/12/analysis-more-than-16-cores-may-well-be-pointless.ars>, Dec. 2008
  10. Wilkes M (1995) The memory wall and the CMOS end-point. Comput Archit News 23(4):4–6
  11. Wulf W, McKee S (1995) Hitting the memory wall: implications of the obvious. Comput Archit News 23(1):20–24
- 

## MEMSY

- Erlangen General Purpose Array (EGPA)

## Mesh

- Hypercubes and Meshes
- Networks, Direct

## Mesh Partitioning

- METIS and ParMETIS

## Message Passing

- MPI (Message Passing Interface)
- PVM (Parallel Virtual Machine)

## Message Passing Interface (MPI)

- MPI (Message Passing Interface)

## Message-Passing Performance Models

- Bandwidth-Latency Models (BSP, LogP)

## METIS and ParMETIS

GEORGE KARYPIS

University of Minnesota, Minneapolis,  
MN, USA

### Synonyms

Decomposition; Fill-reducing orderings; Graph partitioning; Load balancing; Mesh partitioning

### Definition

METIS and ParMETIS are serial and parallel software packages for partitioning and repartitioning large irregular graphs and unstructured meshes and for computing fill-reducing orderings of sparse matrices.

### Introduction

Algorithms that find a good partitioning of highly irregular graphs and unstructured meshes are critical for developing efficient solutions for a wide range of problems in many application areas on both serial and parallel computers. For example, large-scale numerical simulations on parallel computers, such as those based on finite element methods, require the distribution of the finite element mesh to the processors. This distribution must be done so that the number of elements assigned to each processor is the same, and the number of adjacent elements assigned on different processors is minimized. The goal of the first condition is to balance the computations among the processors. The goal of the second condition is to minimize the communication resulting from the placement of adjacent elements to different processors. Graph partitioning can be used to successfully satisfy these conditions by first modeling the finite element mesh by a graph, and then partitioning it into equal parts.

Graph partitioning algorithms are also used to compute fill-reducing orderings of sparse matrices. These fill-reducing orderings are useful when direct methods are used to solve sparse systems of linear equations. A good ordering of a sparse matrix dramatically reduces both the amount of memory and the time required to solve the system of equations. Furthermore, the fill-reducing orderings produced by graph partitioning

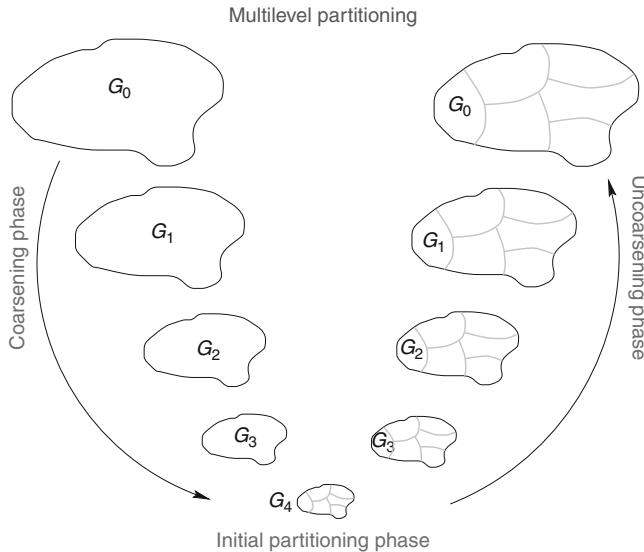
algorithms are particularly suited for parallel direct factorization as they lead to a high degree of concurrency during the factorization phase.

### METIS

METIS is a serial software package for partitioning large irregular graphs, partitioning large meshes, and computing fill-reducing orderings of sparse matrices. METIS has been developed at the Department of Computer Science and Engineering at the University of Minnesota and is freely distributed. Its source code can be downloaded directly from <http://www.cs.umn.edu/~metis>, and is also included in numerous software distributions for Unix-like operating systems such as Linux and FreeBSD.

The algorithms implemented in METIS are based on the multilevel graph partitioning paradigm [3, 4, 6], which has been shown to quickly produce high-quality partitions and fill-reducing orderings. The multilevel paradigm, illustrated in Fig. 1, consists of three phases: graph coarsening, initial partitioning, and uncoarsening. In the graph coarsening phase, a series of successively smaller graphs is derived from the input graph. Each successive graph is constructed from the previous graph by collapsing together a maximal size set of adjacent pairs of vertices. This process continues until the size of the graph has been reduced to just a few hundred vertices. In the initial partitioning phase, a partitioning of the coarsest and hence, smallest, graph is computed using relatively simple approaches such as the algorithm developed by Kernighan-Lin [8]. Since the coarsest graph is usually very small, this step is very fast. Finally, in the uncoarsening phase, the partitioning of the smallest graph is *projected* to the successively larger graphs by assigning the pairs of vertices that were collapsed together to the same partition as that of their corresponding collapsed vertex. After each projection step, the partitioning is refined using various heuristic methods to iteratively move vertices between partitions as long as such moves improve the quality of the partitioning solution. The uncoarsening phase ends when the partitioning solution has been projected all the way to the original graph.

METIS uses novel approaches to successively reduce the size of the graph as well as to refine the partition during the uncoarsening phase. During coarsening, METIS



**METIS and ParMETIS.** Fig. 1 The three phases of multilevel  $k$ -way graph partitioning. During the coarsening phase, the size of the graph is successively decreased. During the initial partitioning phase, a  $k$ -way partitioning is computed. During the multilevel refinement (or uncoarsening) phase, the partitioning is successively refined as it is projected to the larger graphs.  $G_0$  is the input graph, which is the finest graph.  $G_{i+1}$  is the next level coarser graph of  $G_i$ .  $G_4$  is the coarsest graph

employs algorithms that make it easier to find a high-quality partition at the coarsest graph. During refinement, METIS focuses primarily on the portion of the graph that is close to the partition boundary. These highly tuned algorithms allow METIS to quickly produce high-quality partitions and fill-reducing orderings for a wide variety of irregular graphs, unstructured meshes, and sparse matrices.

METIS provides a set of stand-alone command-line programs for computing partitionings and fill-reducing orderings as well as an application programming interface (API) that can be used to invoke its various algorithms from C/C++ or Fortran programs. The list of stand-alone programs and API routines of the current version of METIS (version 4.x) is shown in Table 1. The API routines allow the user to alter the behavior of the various algorithms and provide additional routines that partition graphs into unequal-size partitions and compute partitionings that directly minimize the total communication volume.

### Partitioning a Graph

METIS provides the pmetis and kmetis programs for partitioning an unstructured graph into a user-specified number of  $k$  equal-size parts. The partitioning algorithm used by pmetis is based on multilevel

recursive bisection described in [6], whereas the partitioning algorithm used by kmetis is based on multi-level  $k$ -way partitioning described in [4]. Both of these methods are able to produce high-quality partitions. However, depending on the application, one program may be preferable than the other. In general, kmetis is preferred when it is necessary to partition graphs into more than just a small number of partitions. For such cases, kmetis is considerably faster than pmetis. On the other hand, pmetis is preferable for partitioning a graph into a small number of partitions. The API routines corresponding to the pmetis and kmetis programs are METIS\_PartGraphRecursive and METIS\_PartGraphKway, respectively.

### Alternate Partitioning Objectives

The objective of the traditional graph partitioning problem is to compute a balanced  $k$ -way partitioning such that the number of edges (or in the case of weighted graphs the sum of their weights) that straddle different partitions is minimized. This objective is commonly referred to as the *edge-cut*. When partitioning is used to distribute a graph or a mesh among the processors of a parallel computer, the objective of minimizing the edge-cut is only an approximation of the true communication cost resulting from the partitioning [2].

**METIS and ParMETIS. Table 1** An overview of METIS' command-line and library interfaces

| Operation                   | Stand-alone program | API routine                |
|-----------------------------|---------------------|----------------------------|
| Partition a graph           | pmetis              | METIS_PartGraphRecursive   |
|                             | kmetis              | METIS_PartGraphKway        |
|                             |                     | METIS_PartGraphVKway       |
|                             |                     | METIS_mCPartGraphRecursive |
|                             |                     | METIS_mcPartGraphKway      |
|                             |                     | METIS_WPartGraphRecursive  |
|                             |                     | METIS_WPartGraphKway       |
|                             |                     | METIS_WPartGraphVKway      |
| Partition a mesh            | partnmesh           | METIS_PartMeshNodal        |
|                             | partdmesh           | METIS_PartMeshDual         |
| Compute a fill-reducing     | oemmetis            | METIS_EdgeND               |
| Ordering of a sparse matrix | onmetis             | METIS_NodeND               |
|                             |                     | METIS_NodeWND              |
| Convert a mesh into a graph | mesh2nodal          | METIS_MeshToNodal          |
|                             | mesh2dual           | METIS_MeshToDual           |

The communication cost resulting from a  $k$ -way partitioning generally depends on the following factors: (a) the total communication volume, (b) the maximum amount of data that any particular processor needs to send and receive; and (c) the number of messages a processor needs to send and receive. METIS' API provides the METIS\_PartGraphVKway and METIS\_WPartGraphVKway routines that can be used to directly minimize the total communication volume resulting from the partitioning (first factor). In addition, METIS also provides support for minimizing the third factor (which essentially reduces the number of startups) and indirectly (up to a point) reduces the second factor. This enhancement is provided as a refinement option for both the METIS\_PartGraphKway and METIS\_PartGraphVKway routines, and is the default option of kmetis and METIS\_PartGraphs Kway.

### Support for Multiphase and Multi-physics Computations

The traditional graph partitioning problem formulation is limited in the types of applications that it can effectively load balance because it specifies that only a single quantity be balanced. Many important types of multiphase and multi-physics computations require that multiple quantities be balanced simultaneously. This is because synchronization steps exist between the

different phases of the computations, and so, each phase must be individually load balanced. That is, it is not sufficient to simply sum up the relative times required for each phase and to compute a partitioning based on this sum. Doing so may lead to some processors having too much work during one phase of the computation (and so, these may still be working after other processors are idle), and not enough work during another. Instead, it is critical that every processor must have an equal amount of work from each phase of the computation.

METIS includes partitioning routines that can be used to partition a graph in the presence of such multiple balancing constraints. Each vertex is now assigned a vector of  $m$  weights and the objective of the partitioning routines is to minimize the edge-cut subject to the constraints that each one of the  $m$  weights is equally distributed among the domains. For example, if the first weight corresponds to the amount of computation and the second weight corresponds to the amount of storage required for each element, then the partitioning computed by the new algorithms will balance both the computation performed in each domain and the amount of memory that it requires. The multi-constraint partitioning algorithms and their applications are further described in [3]. The pmetis and kmetis programs invoke the multi-constraint partitioning routines whenever the input graph specifies more than one set of vertex weights. Support for multi-constraint

partitioning via METIS' API is provided by the `METIS_mCPartGraphRecursive` and `METIS_METIS_mCPartGraphKway` routines that are based on the multilevel recursive bisection and the multilevel  $k$ -way partitioning paradigms, respectively.

## Partitioning for Heterogeneous Parallel Computing Architectures

Heterogeneous computing platforms containing processing nodes with different computational and memory capabilities are becoming increasingly more common. METIS' API provides the `METIS_WPartGraphRecursive`, `METIS_WPartGraphKway`, and `METIS_WPartGraphVKway` routines, which are designed to partition a graph into  $k$  parts such that each part contains a prespecified fraction of the total number of vertices (or vertex weight). By matching the weights specified for each partition to the relative computational and memory capabilities of the various processors, these routines can be used to compute partitions that balance the computations on heterogeneous architectures.

## Partitioning a Mesh

METIS provides the `partnmesh` and `partdmesh` programs for partitioning meshes arising in finite element or finite volume methods. These programs take as input the element node array of the mesh and compute a  $k$ -way partitioning for both its elements and its nodes. METIS currently supports four different types of mesh elements that are triangles, tetrahedra, hexahedra (bricks), and quadrilaterals. These programs first convert the mesh into a graph, and then use `kmetis` to partition this graph. The difference between these two programs is that `partnmesh` converts the mesh into a nodal graph (i.e., each node of the mesh becomes a vertex of the graph), whereas `partdmesh` converts the mesh into a dual graph (i.e., each element becomes a vertex of the graph). In the case of `partnmesh`, the partitioning of the nodal graph is used to derive a partitioning of the elements. In the case of `partdmesh`, the partitioning of the dual graph is used to derive a partitioning of the nodes. Both of these programs produce partitioning of comparable quality, with `partnmesh` being considerably faster than `partdmesh`. However, in some cases, `partnmesh` may produce partitions that have higher load imbalance

than `partdmesh`. The API routines corresponding to the `partnmesh` and `partdmesh` programs are `METIS_PartMeshNodal` and `METIS_PartMeshDual`, respectively.

## Computing a Fill-Reducing Ordering of a Sparse Matrix

METIS provides two programs `ometis` and `onmetis` for computing fill-reducing orderings of sparse matrices. Both programs use multilevel nested dissection to compute a fill-reducing ordering [6]. The nested dissection paradigm is based on computing a vertex separator for the graph corresponding to the matrix. The nodes in the separator are moved to the end of the matrix, and a similar process is applied recursively for each one of the other two parts.

These programs differ on how they compute the vertex separators. The `ometis` program finds a vertex separator by first computing an edge separator using a multilevel algorithm, whereas the `onmetis` program uses the multilevel paradigm to directly find a vertex separator. The orderings produced by `onmetis` generally incur less fill than those produced by `ometis`. In particular, for matrices arising in linear programming problems the orderings computed by `onmetis` are significantly better than those produced by `ometis`. Furthermore, `onmetis` utilizes compression techniques to reduce the size of the graph prior to computing the ordering. Sparse matrices arising in many application domains are such that certain rows of the matrix have the same sparsity patterns. Such matrices can be represented by a much smaller graph in which all rows with identical sparsity pattern are represented by just a single vertex whose weight is equal to the number of rows. Such compression techniques can significantly reduce the size of the graph, whenever applicable, and substantially reduce the amount of time required by `onmetis`. However, when there is no reduction in graph size, `ometis` is about 20–30% faster than `onmetis`. Furthermore, for large matrices arising in three-dimensional problems, the quality of orderings produced by the two algorithms is quite similar.

METIS' API provides the `METIS_EdgeND`, `METIS_NodeND`, and `METIS_NodeWND` routines for computing fill-reducing orderings for sparse matrices. The first two routines provide the functionality

of the `oemetis` and `onmetis` programs, respectively; whereas, the third routine is designed to compute fill-reducing orderings of graphs whose vertices have weights corresponding to the number of rows in the original sparse matrix with identical sparsity structure.

### Converting a Mesh into a Graph

METIS provides two programs: `mesh2nodal` and `mesh2dual` for converting a mesh into the graph format used by METIS. In particular, `mesh2nodal` converts the element node array of a mesh into a nodal graph; that is, each node of the mesh corresponds to a vertex in the graph and two vertices are connected by an edge if the corresponding nodes are connected by lines in the mesh. Similarly, `mesh2dual` converts the element node array of a mesh into a dual graph; that is, each element of the mesh corresponds to a vertex in the graph and two vertices are connected if the corresponding elements in the mesh share a face. The API routines corresponding to these two programs are `METIS_MeshToNodal` and `METIS_MeshToDual`, respectively. Since METIS does not provide API routines that can directly compute a multi-constraint partitioning of a mesh or compute a partitioning that minimizes its overall communication cost, these routines can be used to first convert the mesh into a graph, which can then be used as input to METIS' graph partitioning routines to obtain such partitionings.

### ParMETIS

ParMETIS is an MPI-based parallel library that implements a variety of algorithms for partitioning and repartitioning unstructured graphs and meshes and for computing fill-reducing orderings of sparse matrices.

ParMETIS has been developed at the Department of Computer Science and Engineering at the University of Minnesota and is freely distributed. Its source code can be downloaded directly from <http://www.cs.umn.edu/~metis>. The algorithms in ParMETIS are based on the corresponding algorithms implemented in METIS. However, ParMETIS extends METIS' functionality by including routines that are especially suited for parallel computations and large-scale numerical simulations.

The complete list of operations that ParMETIS' routines can perform and the associated API routines of the current version of ParMETIS (version 3.x) is shown in Table 2. ParMETIS' partitioning and repartitioning routines provide support for single- and multi-constraint partitioning and for parallel computational platforms with heterogeneous computing capabilities. Note that unlike METIS, ParMETIS does not provide any stand-alone parallel programs for parallel partitioning and ordering.

### Partitioning a Graph

`ParMETIS_V3_PartKway` is the routine in ParMETIS that is used to partition unstructured graphs. This routine takes a graph and computes a  $k$ -way partitioning that minimizes the edge-cut. `ParMETIS_V3_PartKway` makes no assumptions on how the graph is initially distributed among the processors. It can effectively partition a graph that is randomly distributed as well as a graph that is well distributed. The parallel graph partitioning algorithm used in `ParMETIS_V3_PartKway` is based on the serial multilevel  $k$ -way partitioning algorithm (implemented by METIS' `kmetis` program and `METIS_Part`

**METIS and ParMETIS. Table 2** An overview of ParMETIS' API routine

| Partition                                                       | API routine                             |
|-----------------------------------------------------------------|-----------------------------------------|
| Partition a graph                                               | <code>ParMETIS_V3_PartKway</code>       |
|                                                                 | <code>ParMETIS_V3_PartGeom</code>       |
|                                                                 | <code>ParMETIS_V3_PartGeomKway</code>   |
| Partition a mesh                                                | <code>ParMETIS_V3_PartMeshKway</code>   |
| Repartition a graph corresponding to an adaptively refined mesh | <code>ParMETIS_V3_AdaptiveRepart</code> |
| Refine the quality of an existing partitioning                  | <code>ParMETIS_V3_RefineKway</code>     |
| Compute a fill-reducing ordering of a sparse matrix             | <code>ParMETIS_V3_NodeND</code>         |
| Convert a mesh into a graph                                     | <code>ParMETIS_V3_MeshToDual</code>     |

GraphKway API routine) and parallelized in [5, 7, 9, 10].

When coordinate information is available about the vertices of the graph (e.g., when the vertices correspond to mesh nodes or elements), ParMETIS provides two additional routines that can be used to compute the partitioning. The first is `ParMETIS_V3_PartGeom` that uses an approach based on space-filling curves. This routine is very fast but produces relatively low-quality solutions. The second is `ParMETIS_V3_PartGeomKway` that first computes an initial partitioning using the method based on space-filling curves, redistributes the graph according to this partitioning, and then calls `ParMETIS_V3_PartKway` to compute the final high-quality partitioning of the graph. Even though this approach computes two different partitionings, because the second partitioning is computed using a graph that is already well-distributed among the processors, it actually takes less time. Also, the quality of the final partitioning produced by `ParMETIS_V3_PartGeomKway` is comparable to that computed by `ParMETIS_V3_PartKway`.

Note that in most of ParMETIS' partitioning routines the number of sub-domains is decoupled from the number of processors. Hence, it is possible to use these routines to compute a  $k$ -way partitioning independent of the number of processors that are used.

### Partitioning Adaptively Refined Meshes

For large-scale scientific simulations, the computational requirements of techniques relying on globally refined meshes become very high, especially as the complexity and size of the problems increase. By locally refining and de-refining the mesh either to capture flow-field phenomena of interest or to account for variations in errors, adaptive methods make standard computational methods more cost effective. The efficient execution of such adaptive scientific simulations on parallel computers requires a periodic repartitioning of the underlying computational mesh. These repartitionings should minimize both the inter-processor communications incurred in the iterative mesh-based computation and the data redistribution costs required to balance the load.

Repartitioning algorithms fall into two general categories. The first category balances the computation by incrementally diffusing load from those sub-domains

that have more work to adjacent sub-domains that have less work. These schemes are referred to as *diffusive schemes*. The second category balances the load by computing an entirely new partitioning, and then intelligently mapping the sub-domains of the new partitioning to the processors such that the redistribution cost is minimized. These schemes are generally referred to as *remapping schemes*. Remapping schemes typically lead to repartitionings that have smaller edge-cuts, while diffusive schemes lead to repartitionings that incur smaller redistribution costs. However, since these results can vary significantly among different types of applications, it can be difficult to select the best repartitioning scheme for the job.

ParMETIS provides the `ParMETIS_V3_AdaptiveRepart` routine for repartitioning adaptively refined meshes. This routine assumes that the mesh is well distributed among the processors, but that (due to mesh refinement and de-refinement) this distribution is poorly load balanced. `ParMETIS_V3_AdaptiveRepart` is based on the Unified Repartitioning Algorithm [10, 12] and combines the best characteristics of remapping and diffusion-based repartitioning schemes. A key parameter used by this algorithm is the *ITR Factor* that describes the ratio between the time required for performing the inter-processor communications incurred during parallel processing compared to the time to perform the data redistribution associated with balancing the load. As such, it allows for the computation of a single metric that describes the quality of the repartitioning.

The underlying algorithm of `ParMETIS_V3_AdaptiveRepart` follows the multilevel partitioning paradigm but uses a technique known as *local coarsening* that only collapses together vertices that are located on the same processor. On the coarsest graph, an initial partitioning need not be computed, as one can either be derived from the initial graph distribution (in the case when sub-domains are coupled to processors), or else one needs to be supplied as an input to the routine (in the case when sub-domains are decoupled from processors). However, this partitioning does need to be balanced. The balancing phase is performed on the coarsest graph by using both remapping- and diffusion-based algorithms [11]. A quality metric for each of these partitionings is then computed (using the ITR Factor) and the partitioning with the highest quality is selected.

This technique tends to give very good points from which to start multilevel refinement, regardless of the type of repartitioning problem or the value of the ITR Factor. Note that the fact that the algorithm computes two initial partitionings does not impact its scalability as long as the size of the coarsest graph is suitably small. Finally, multilevel refinement is performed on the balanced partitioning in order to further improve its quality. Since `ParMETIS_V3_AdaptiveRepart` starts from a graph that is already well distributed, it is very fast.

Appropriate values for the ITR Factor can easily be determined depending on the times required to perform (a) all inter-processor communications that have occurred since the last repartitioning, and (b) the data redistribution associated with the last repartitioning/load balancing phase. Simply divide the first time by the second. The result is the correct ITR Factor. In case these times cannot be ascertained (e.g., for the first repartitioning/load balancing phase), our experiments have shown that values between 100 and 1,000 work well for a variety of situations.

`ParMETIS_V3_AdaptiveRepart` can be used to load balance the mesh either before or after mesh adaptation. In the latter case, each processor first locally adapts its mesh, leading to different processors having different numbers of elements. `ParMETIS_V3_AdaptiveRepart` can then compute a partitioning in which the load is balanced. However, load balancing can also be done before adaptation if the degree of refinement for each element can be estimated *a priori*. That is, if we know ahead of time into how many new elements each old element will subdivide, we can use these estimations as the weights of the vertices for the graph that corresponds to the dual of the mesh. In this case, the mesh can be redistributed before adaption takes place. This technique can significantly reduce data redistribution times.

### Improving the Quality of a Partitioning

ParMETIS provides the `ParMETIS_V3_RefineKway` routine that can be used to improve the quality of an existing partitioning. `ParMETIS_V3_RefineKway` can be used to improve the quality of partitionings that are produced by other partitioning algorithms. `ParMETIS_V3_RefineKway` can also be used repeatedly to further improve the quality

of a partitioning. However, each successive call to `ParMETIS_V3_RefineKway` will tend to produce smaller improvements in quality.

### Partitioning Meshes

ParMETIS provides the `ParMETIS_V3_PartMeshKway` routine for directly partitioning the elements of an unstructured (or structured) mesh. This partitioning is computed by first converting the mesh into a graph whose vertices are the elements of the mesh and then using `ParMETIS_V3_PartKway` to compute the actual partitioning. Note that in the case of adaptive computations, ParMETIS does not provide routines that can directly repartition adaptively refined meshes. However, an application can use the `ParMETIS_V3_MeshToDual` routine to construct in parallel the dual graph of the mesh, which can then be provided as input to the `ParMETIS_V3_AdaptiveRepart` routine.

### Computing Fill-reducing Orderings of Sparse Matrices

`ParMETIS_V3_NodeND` is the routine provided by ParMETIS for computing fill-reducing orderings of sparse matrices. The algorithm implemented by `ParMETIS_V3_NodeND` is similar to that implemented by METIS' `onmetis` stand-alone program and `METIS_NodeND` routine. To achieve high performance, `ParMETIS_V3_NodeND` first uses `ParMETIS_V3_PartKway` to compute a high-quality partitioning and redistributes the graph accordingly. Next it proceeds to compute the  $\log p$  levels of the elimination tree concurrently. When the graph has been separated into  $p$  parts (where  $p$  is the number of processors), the graph is redistributed among the processor so that each processor receives a single subgraph, and `METIS_NodeND` is used to order these smaller subgraphs.

### Related Entries

- ▶ [Chaco](#)
- ▶ [Dense Linear System Solvers](#)
- ▶ [Domain Decomposition](#)
- ▶ [Graph Partitioning](#)
- ▶ [Hypergraph Partitioning](#)
- ▶ [Load Balancing, Distributed Memory](#)
- ▶ [PaToH \(Partitioning Tool for Hypergraphs\)](#)

- [Reordering](#)
- [Sparse Direct Methods](#)

## Bibliographic Notes and Further Readings

There is an extensive body of research related to the topic of graph and mesh partitioning and its applications to parallel and scientific computing. Schloegel et al. [13] and Devine et al. [1] provide comprehensive overviews of the various methods that have been developed and their applications to high performance scientific computing. In addition, besides METIS and ParMETIS, there are a number of other tools that have been developed and are actively maintained for serial and parallel graph partitioning.

## Bibliography

1. Devine K, Boman EG, Karypis G (2006) Partitioning and load balancing for emerging parallel applications and architectures. In: Heroux M, Raghavan P, Simon HD (eds) Parallel Processing for Scientific Computing. SIAM
2. Hendrickson B (1998) Graph partitioning and parallel solvers: Has the emperor no clothes? In: Proceedings irregular 1998, Berlin
3. Karypis G, Kumar V (1998) Multilevel algorithms for multi-constraint graph partitioning. In: Proceedings of supercomputing 1998, New York
4. Karypis G, Kumar V (1998) Multilevel k-way partitioning scheme for irregular graphs. J Parallel Distrib Comput 48(1):96–129
5. Karypis G, Kumar V (1998) A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. J Parallel Distrib Comput 48(1):71–95
6. Karypis G, Kumar V (1999) A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J Scientific Comput 20(1):359–392
7. Karypis G, Kumar V (1997) A coarse-grain parallel multilevel k-way partitioning algorithm. In: Proceedings of the eighth SIAM conference on parallel processing for scientific computing 1997, Minneapolis, MN
8. Kernighan BW, Lin S (1970) An efficient heuristic procedure for partitioning graphs. Bell Syst Tech J 49(2):291–307
9. Schloegel K, Karypis G, Kumar V (2000) Parallel multilevel algorithms for multi-constraint graph partitioning. In: Proceedings of Europar 2000, 2000. Distinguished Paper Award
10. Schloegel K, Karypis G, Kumar V (2000) A unified algorithm for load-balancing adaptive scientific simulations. In: Proceedings supercomputing '00, Santa Fe, New Mexico
11. Schloegel K, Karypis G, Kumar V (2001) Wavefront diffusion and lmsr: algorithms for dynamic repartitioning of adaptive meshes. IEEE Trans Parallel Distrib Syst 12(5):451–466
12. Schloegel K, Karypis G, Kumar V (2002) Parallel static and dynamic multi-constraint partitioning. Concurrency Comput Pract Ex 14:219–240
13. Schloegel K, George Karypis, and Vipin Kumar (2002) Graph partitioning for high performance scientific simulations. In: Dongarra J, Foster I, Fox G, Kennedy K, White A, Torczon L, Gropp W (eds) Sourcebook of parallel computing. Morgan Kaufmann

## Metrics

ALLEN D. MALONY

University of Oregon, Eugene, OR, USA

## Synonyms

[Performance metrics](#)

## Definition

A metric is a quantitative measure used to characterize, evaluate, and compare parallel performance. Many performance metrics are defined based on the variety of performance data and other information that can be obtained from a measurement of a parallel execution. Metrics can be absolute as well as derived from other metrics. Parallel metrics are used to understand the relationship of scalability and performance.

## Discussion

### Introduction

Performance is at the heart of parallel computing. Qualitatively, performance embodies those aspects that motivate parallel computing research and development, such as delivering greater speed and solving larger problems. However, for purposes of scientific and engineering discussion, it is necessary to describe performance in more quantitative terms. Metrics make it possible to characterize, evaluate, and compare performance based on methods for performance data measurement and analysis. Metrics are determined from performance measures. Where a *measure* constitutes a quantity or degree of performance, and implies a means for ascertaining (measuring) it, a *metric* is a “standard of measure,” a semantic representation of performance. Metrics are the language of parallel performance science.

Metrics are mostly determined by the variety of performance data that can be obtained in measurements,

reflecting different features of interest in a parallel execution. Metrics can be of different types, depending on the purpose for their use. Where do metrics come from? What are the common metrics used in the parallel computing community? How are metrics applied? These are some of the questions discussed below.

## Standard Metrics

Performance metrics are based on measures of parallel operation. Two questions naturally follow. What are the features of parallel operation to quantify? How are those features measured in a real parallel program? The term “standard metric” is used here to reflect common performance data from measurement of parallel program execution.

## Time Metrics

Parallel computing is generally concerned with reducing the amount of time necessary to perform a computational task, whether that be the execution of single parallel application or the running of many jobs together on a parallel machine. Thus, the most fundamental performance metric of interest to parallel computing is *execution time*. A time metric is calculated in different ways depending on what is available for time measurement, and there can be different types of execution time to observe.

Time measurement requires a clock source (a clock reference), and different sources have different resolution and accuracy characteristics. *Wallclock time* regards the passage of real time as a clock basis, as one would use a clock on the wall. It is used often to quantify the actual time taken for a parallel execution to complete, from start to finish. One could use a watch to time a parallel program, but all operating systems provide some form of real time clock to keep track of time. The real time clock is created from a hardware source and can have a high degree of resolution.

However, there are different components of time reflecting different states of execution. Operating systems normally keep track of *CPU time*, *I/O time*, and the time spent *idle* for a process. CPU time is typically broken down into *user* and *system* time. User time is the time the process spends in execution of the program code and system time is the time the operating system spends executing on behalf of the process. The CPU time measure (with user and system components)

is commonly referred to as *process virtual time* since it is maintained on a process basis and the virtual clock advances only when the process is executing in either user or system mode.

Whereas it is helpful to see the breakdown of time for each individual process or thread of a parallel program, process virtual time cannot serve as a uniform time reference for the whole parallel execution, primarily because it ignores periods when the process is idle. In general, it is important to have a single, real time clock reference for measuring time on all processes to determine accurately the time from the beginning of the parallel computation to the end. The real time clock is also necessary to allow performance measurements of interactions between processes (threads). In practice, synchronized clocks are hard to achieve in a parallel machine, requiring local real time clocks to be used with software-based time correlation.

In summary, time metrics can apply to the parallel program as a whole, in which case it is necessary to have a common time basis. *Total execution time* is the metric used most often for reporting the performance of a parallel execution. It represents the elapsed time from the beginning of the parallel computation to the end. However, further breakdown of execution time across the processes / threads of the parallel program can be important for performance analysis, especially of synchronization and communication operations. Multiple methods of measuring time are used to compute time metrics of concern. Note, it is also useful to sum time metrics from individual processes to get a single metric representing total time consumed overall. For instance, adding the time processes spend doing real computation could represent total *work*. Similarly, adding the time processes spend in synchronization and communication could represent total *overhead*.

## OS and Hardware Metrics

Time is not the only metric used to characterize parallel execution. The operating system and processor hardware provide performance data that reflects how a program interacts with the parallel machine. From the operating system standpoint, it is possible to see information about virtual memory, paging, I/O, and networking. OS metrics typically captures performance data at the process level as counts of interesting events and other quantitative measures.

The availability of counters in modern computing hardware has made it possible to generate performance metrics about the operation of the processor, memory system, and other hardware components. All CPUs today have counters available covering a range of functionality from tracking instructions executed, to measuring cache operation, to basic timing. Access to the counters are made possible through the instruction set architecture, at the low level, and libraries such as the Performance API (PAPI) [3], at higher levels. Using the counter interfaces, it is possible to calculate metrics such as the number of floating point operations (*flops*), level 1 cache misses (data, instruction), translation lookaside buffer misses, and so on.

OS and hardware metrics are measured locally in a parallel system. For some, it makes sense to accumulate the performance data to compute a total value, as a way to convey an amount of work, such as in the case of total flops, or an average value, to elucidate an overall parallel inefficiency, as in the case of average cache misses per processor. Clearly, flops (typically reported as millions or billions of flops, *Mflops* or *Gflops*) is an important metric since it is used commonly to represent the amount of computational work in scientific applications.

### Derived Metrics

The types of metrics above one might regard as *raw metrics* since they come directly from measurement. New metrics can be calculated from other metrics to better characterize performance. A *derived metric* is the result of an arithmetic metric expression. The best known derived metric is flops per second rate (also referred to by the acronym *flops*), determined by dividing the total number of floating point operations performance by the execution (summed over all processes) by the total execution time. Flops rate is the primary derived metric reported for high-performance scientific computing.

Rate-based derived metrics are easily calculated from any raw metric. Any OS or hardware counter can be used to determine informative rate metrics, such as instructions per second, cache misses per second, or page faults per second. Sometimes it is necessary to instrument the program code to determine certain values. For instance, sustainable *memory bandwidth* and *communication bandwidth* in gigabytes per second

depends on knowing how much data is being referenced or transferred.

However, the derived metric expression can be anything. One important derived metric is the *computation-to-communication ratio*, computed in two ways:

1. The number of calculations (determined by some computation metric) divided by the number of communication messages (measured as number or bytes)
2. The time spent calculating divided by the time spent communicating

The computation-to-communication ratio is helpful in gauging the *efficiency* of a parallel execution. Lower ratios suggest more communication overhead in the parallel execution. The ratio could also reflect the differences in performance of the processing and communications infrastructure. Improving communication network bandwidth and latency would result in an increase in the ratio. This type of derived metric could be called a *relative metric* since it is describing one metric with respect to another, generally as a ratio. If time is the denominator, the relative metric becomes a rate.

Other expressions for deriving metrics might help in the aggregation or categorization of more specific performance, such as in summing the time spent in all MPI routines into a single MPI class.

### Parallelism Metrics

The metrics discussed above can be thought of as describing aspects of parallel execution performance where the parallelism degree, problem size, and other parameters are fixed. Such an execution instance is typically called a *performance experiment*. Obviously, standard metrics from multiple performance experiments can be captured and compared. In this regard, new *comparative metrics* can be created to highlight certain cross-experiment performance aspects. By definition, each performance experiment reflects differences in how the experiment was constructed, on what platform it executed, the input data sets, and many other parameters of performance interest. A comparative metric can analyze performance along one or several parameter dimensions.

A simple comparative metric would consider how a standard or derived performance metric varies for

changes in a single parameter. Consider varying the amount of calculations performed by a parallel program by increasing the size of the problem. The number of parallel processes used will remain fixed. This is a type of performance study called *problem size scaling*. After running the performance experiment for each problem size, the standard performance metrics can be compared with respect to the problem size choices. One problem size could be chosen as the base, for instance, and performance metrics for the other sizes normalized to the base metrics. Since the parallelism degree is fixed, the result of the comparative analysis gives insight to effects on performance due to the amount of computation, memory allocation, and other factors dependent on problem size.

The term “parallelism metrics” will be used generally to characterize performance effects as a result of changes in parallel execution parameters. As comparative metrics, parallelism metrics have implicit and explicit relevance to how parallelism is manifesting in performance variation.

## Speedup

A fundamental parallelism metric is *speedup*. Simply put, speedup expresses the performance improvement as parallelism increases. If  $T_s$  is the sequential execution time of a program and  $T_p$  is the execution time when the program is run in parallel on  $p$  processors, speedup is the simple expression:  $S = S_p = T_s/T_p$ . The speedup metric conveys performance improvement when  $S > 1$  and degradation when  $S < 1$ . *Ideal speedup* (also known as *ideal scaling*) is defined by  $S_p = p$ . *Superlinear speedup* is obtained when  $S_p > p$ .

However, there are two well-known speedup formulations, determined by how execution time is modeled. The first is taken from Amdahl's law [1] which is a method to find the maximum expected improvement to an overall system when only part of the system is improved. If the number of processors is increased for use in the parallel portion of an application's computation, Amdahl's law is a speedup expression that identifies the theoretical maximum performance improvement possible.

Let  $T_s$  be the execution time of a sequential computation. Let  $\alpha$  be the fraction of the computation that cannot be parallelized. If there are  $p$  processors, then the time of the parallel computation,  $T_p$ , can be written

as  $T_p = \alpha * T_s + (1 - \alpha) * T_s/p$ , assuming the parallel execution is ideal and does not result in any overheads. Now the speedup equation can be written as:  $S = T_s/T_p = \frac{1}{\alpha + (1-\alpha)/p}$ . Notice as  $p \rightarrow \infty$ ,  $S$  is bounded by  $\frac{1}{\alpha}$ . In other words, the sequential portion of the computation limits the speedup that can be achieved.

Speedup formulated in this manner is also known as *strong speedup* (or *strong scaling*), since the size of the computational work to be performed is defined by the sequential execution and does not change. The *sequential fraction* becomes a dominant, strong factor in the speedup metric. There was concern in the parallel computing community that large-scale parallel performance would be hard to achieve since even tiny portions of sequential execution place severe constraints on the actual speedup that can be obtained. For example, a 1% sequential fraction limits speedup to 100 maximum.

Suppose instead that the computational work is allowed to increase with increasing parallelism degree. This is the basis of Gustafson's Law [6]. Let  $n$  be the problem size,  $p$  be the number of processors, and  $T_p$  be the execution time on a parallel computer. We can write  $T_p = a(n) + b(n)$  where  $a(n)$  is the part that executes sequentially and  $b(n)$  is the part that executes in parallel. To determine execution time on a single processor, we just multiply the parallel component by the number of processors used:  $T_s = a(n) + p * b(n)$ . (Again, overheads are assumed to be zero.) If  $T_p$  is normalized to 1,  $a(n)$  and  $b(n)$  then represent sequential and parallel fractions of the total computation. The speedup equation can be written as:  $S = T_s/T_p = T_s = a(n) + p * (1 - a(n))$ .

Now assume increasing problem size. If the serial portion,  $a(n)$ , diminishes (as a fraction of the total computation) as  $n$  increases, speedup will be bounded by the number of processors,  $p$ :  $S \rightarrow p$  as  $n \rightarrow \infty$ . The term “weak speedup” (or “weak scaling”) is used to describe this speedup metric based on Gustafson's Law. The idea is that good speedups can be achieved with greater numbers of processors, as long as the amount of (relative) parallel work being done can be increased. This tends to be not difficult to do with many applications. As a result, the community became more confident that massively parallel computer systems could be productively utilized.

## Efficiency

Speedup metrics are calculations based on time metrics. How does speedup itself change as a result of parallelism degree? An *efficiency metric* can be defined as the ratio of speedup to the number of processors used to obtain it:  $E_p = \frac{S_p}{p}$ . Efficiency is an example of a *utilization metric*. In essence, it represents a return on investment in parallelism. For instance, a parallel program might achieve a speedup of 100, but it takes 1,000 processors to do it. Efficiency in this case is less than 10%.

Efficiency can change as a result of any performance factor that is influenced by parallelism degree. Returning to problem size, suppose performance experiments are conducted with varying problem size and parallelism. When all experiments are completed, efficiency can be computed for each. To quantify the effect of change problem size on parallel program efficiency, an *isoefficiency function* can be computed. Given a chosen level of efficiency, its purpose is to specify what size of problem must be solved on a given number of processors. Isoefficiency was created as a technique to analyze the scalability of parallel algorithms [5].

## Metric Use

Metrics form the basis of performance evaluation methodology and are used for purposes of benchmarking, characterization, performance problem diagnosis, and optimization. The particular performance evaluation problem will determine which metrics are best to use. Sometimes convention defines the metric(s) of choice.

## Peak Metrics

It has been standard practice to rate parallel machine performance by *peak speed*, usually defined for scientific computing in terms of maximum flops per second a parallel machine can possibly deliver. It is computed by summing up the maximum performance of every processor that can be executing in parallel. Of course, peak speed is never attained by a parallel application, since there are overheads involved and peak speed assumes only certain instructions are used. However, it has been applied as an upper-bound threshold for evaluating parallel execution efficiency. The term “percentage of peak” represents an efficiency metric where the total attained flops per second from a parallel application is divided

by peak flops. Because of the strong influence of processor components and system architecture, percentage of peak can differ from machine to machine for the same application. Clearly, different applications have different parallelism behavior and thus will vary in percentage of peak on the same machine. In general, it is not uncommon to see low percentages of peak for many parallel applications.

## Benchmark Metrics

Benchmarking is the process of conducting controlled experiments using a suite of programs to test performance factors associated with machine platform, compilers, libraries, and other parameters, including parallelism degree and problem size. In contrast to just comparing performance to some absolute measure, benchmarking is intended to allow more informed evaluation with respect to program properties. Benchmark codes are designed to exercise certain execution aspects, whether it be processor, memory, or networking performance, as well as to be representative of application code behavior. There is a long history to benchmarking of parallel machines, particularly for high-performance computing.

The most common metric used in scientific benchmarking is again *flops per second* or just *flops* for short. It is the primary metric in the LINPACK benchmark suite [4] and the basis for the High-Performance LINPACK (HPL) benchmark [10] version used to rank supercomputers in the TOP500 [9] list. The LINPACK benchmarks measure a system’s floating point computing power when solving a dense  $N \times N$  system of linear equations,  $Ax = b$ , using Gaussian elimination with partial pivoting. The number of floating point operations is known,  $\frac{2}{3} * N^3 + 2 * N^2$ , needing only execution time to be measured to determine flops.

To rank systems for the TOP500, HPL is run for different matrix sizes to find the maximal performance,  $R_{\max}$  at  $N_{\max} \times N_{\max}$  matrix size. The peak performance of the system,  $R_{\text{peak}}$ , is also reported. In addition to these, the metric  $N_{1/2}$  is calculated as the matrix dimension where half of the performance ( $R_{\max}/2$ ) is achieved. This metric is reminiscent of the famous  $N_{1/2}$  metric created by Hockney and Jesshope [7] to convey the efficiency of vector pipelined architecture. Here  $N_{1/2}$  refers to the vector length at which half of the theoretical peak vector pipeline performance is achieved.

Larger values mean the vector hardware has a harder time amortizing the pipeline startup costs.

There has been controversy over the years in the use of flops as a single overall parallel performance metric. Some argue that it overemphasizes the importance of floating point operations and obscures other factors more relevant to the resulting performance. There have also been questions about the use of LINPACK as a benchmark for ranking HPC machines because it represents only a particular type of parallel code. Such is the nature of benchmarking.

Other benchmarks have been created to address these concerns. The NAS parallel benchmarks [2] and the SPEC HPC benchmarks [11], among others, provide a variety of parallel codes for evaluation. They support *base* experiments and *tuned* experiments where different optimizations are allowed. The focus is on scientific computing and the primary metric is still flops. However, they also calculate what might be called *workload metrics* as a formula of the different benchmark results, for example, a geometric mean in the case of SPEC HPC.

The HPC Challenge benchmark [8] is different in that it was created to measure a broader spectrum of factors for performance evaluation. Certainly, flops is a metric for certain HPCC codes, but memory and communication bandwidth metrics are targeted with others, to gain a better sense of the capacity of the parallel system. To drill down more on memory performance, one benchmark code calculates billions of integer random updates to memory per second, called *GUPS*. Random memory performance is often a limiter to overall application performance because locality of reference is poor.

## Tuning Metrics

Metrics provide a means to characterize what, how, and where performance is manifested in parallel execution. Derivative and parallel metrics further quantify performance factors and give means to evaluate parallelism effects. As the goal of parallel computing is to deliver performance returns, there is strong interest optimizing applications. Tuning the performance of a parallel application is an iterative process consisting of several steps involving the identification and localization of inefficiencies, instigating their repair, and validating performance improvement.

The idea of *tuning metrics* is to represent the inefficiencies that result in performance degradation so that the critical ones can be targeted and tuning methods can be selected. It is typical to speak of these inefficiencies as *performance bottlenecks* and the process of characterizing and prioritizing bottlenecks as *bottleneck analysis*. Depending on the parallel computational model and parallel programming approach (e.g., shared memory multi-threading or distributed memory message passing), different performance bottlenecks will result. The goal is to find tuning metrics that help in bottleneck analysis.

For instance, tuning methods for task-based parallel programs might apply *critical path analysis* techniques to find the path of parallel execution events that takes the longest time to execute. If optimizations can be made along this critical path to reduce its overall execution time, the parallel performance overall will be improved. However, the critical path may change as a result, meaning that any further improvements with respect to the original path will be ineffective. Critical path analysis can be applied to multi-threaded and message passing programs [12].

Tuning metrics include synchronization overheads, communication delays, and other costs associated with parallel execution. Ordering of severity of these metrics can guide where optimization focus should be applied. Optimal large-scale parallel performance depends strongly on minimizing waiting time across the parallel system. Here, *load balance* is an important metric to consider, but it is a difficult one to actually measure directly.

It should be kept in mind that performance metrics used for tuning are representing complex interactions between multiple performance factors at the algorithm, software, hardware, and system levels. Diagnosing performance problems requires metrics to identify source of inefficiencies, and applying optimizations can change performance effects, requiring further tuning iterations.

## Summary

Parallel computing is an endeavor to improving performance through the use of parallel resources. To evaluate, characterize, diagnose, and tune parallel performance, it is necessary to quantify performance information in forms for critical review. This is the role of metrics.

## Related Entries

- [Amdahl's Law](#)
- [Benchmarks](#)
- [Gustafson's Law](#)
- [Performance Analysis Tools](#)

## Bibliography

1. Amdahl GM (1967) Validity of the single-processor approach to achieving large-scale computing capabilities. In: Proceedings of the American Federation of Information Processing Societies Conference. AFIPS Press, New Jersey, pp 483–485
2. Bailey D et al (1994) The nas parallel benchmarks. Technical Report NAS Technical Report RNR-94-007, NASA Ames Research Center, Moffett Field, March 1994
3. Browne S, Dongarra J, Garner N, Ho G, Mucci P (2000) A portable programming interface for performance evaluation on modern processors. *Int J High Perform Comput Appl* 14(3): 189–204
4. Dongarra J, Bunch J, Moler C, Stewart P (1984) Linpack benchmark. <http://www.netlib.org/lipack/>
5. Grama AY, Gupta A, Kumar V (1993) Isoefficiency: measuring the scalability of parallel algorithms and architectures. *IEEE Concurr* 1(3):12–21
6. Gustafson JL (1988) Reevaluating Amdahl's Law. *Commun ACM* 31(5):532–533
7. Hockney (1995) The science of computer benchmarking. SIAM, Philadelphia
8. HPCS HPC Challenge (2011) Hpc challenge benchmark. <http://icl.cs.utk.edu/hpcc/>
9. Meuer H, Dongarra J, Strohmaier E, Simon H (2005) Top500. <http://www.top500.org/>
10. Petitet A, Whaley R, Dongarra J, Cleary A (2008) HPL – A portable implementation of the high-performance linpack benchmark for distributed-memory computers. <http://www.netlib.org/benchmark/hpl/>
11. Standard Performance Evaluation Corporation (2007) Spec's benchmarks and published results: high performance computing, openmp, MPI. <http://www.spec.org/benchmarks.html>
12. Yang CQ, Miller B (1988) Critical path analysis for the execution of parallel and distributed programs. In: 8th International conference on distributed computing systems. IEEE computer society press, San Jose, California, pp 366–375

## Microprocessors

- [AMD Opteron Processor Barcelona](#)
- [DEC Alpha](#)
- [EPIC Processors](#)
- [IBM Power Architecture](#)

- [Intel Core Microarchitecture, x86 Processor Family](#)
- [Superscalar Processors](#)

## MILC

STEVEN GOTTLIEB

Indiana University, Bloomington, IN, USA

## Synonyms

[MIMD lattice computation](#)

## Definition

MILC is an acronym for MIMD Lattice Computation and is both the name of a long enduring physics collaboration, that is, the MILC Collaboration, and the code and the lattice gauge configurations that the MILC collaboration makes publicly available, that is, the MILC code and the MILC configurations, respectively.

## Discussion

### History of the Collaboration

The MILC collaboration was established in 1990. The original members of the collaboration were Claude Bernard, Thomas DeGrand, Carleton DeTar, Steven Gottlieb, Alex Krasnitz, Michael Ogilvie, Robert Sugar, and Doug Toussaint. Many postdocs and students have joined the collaboration, and more senior members joining later include Urs Heller, Jim Hetrick, and James Osborn. Additional members can be found at the collaboration Web site [1] or by consulting the list of publications [2].

### Physics Goals

Physicists know of four fundamental forces in Nature. They are gravitation, the electromagnetic force, the weak force, and the strong force. The Standard Model of Elementary particle physics describes the latter three forces as gauge theories [3]. For a gauge theory, the forces are determined by a symmetry group. The electromagnetic symmetry group is called  $U(1)$  and the theory is called quantum electrodynamics (QED). The charge of the electron is small and allows an expansion in powers of the charge. This expansion is called perturbation theory. The electromagnetic and weak forces are closely related and the combined theory, called the

electroweak theory, is described by the symmetry group  $SU(2) \times U(1)$ . The combined theory can also be treated by perturbation theory. Applications of perturbation theory to QED date to the 1940s.

For the strong force, the symmetry group is  $SU(3)$  and the theory is called quantum chromodynamics (QCD). Because this force is so strong, perturbation theory is not able to treat many of the interesting phenomena. The MILC Collaboration studies quantum chromodynamics (QCD) using the technique of lattice gauge theory. Lattice gauge theory was formulated by Kenneth Wilson in 1974 [4]. The lattice formulation of a quantum field theory allows a nonperturbative treatment of the theory, which is necessary for QCD and other theories with a strong coupling. According to QCD, matter consists of spin-1/2 particles that are called quarks. There are six different types of quarks and they bind together in combinations of three quarks (called baryons) or as quark-antiquark pairs (called mesons). The MILC collaboration studies the spectrum and decay properties of such bound states [5]. MILC also studies the properties of strongly interacting matter at very high temperature as existed in the early universe very shortly after the Big Bang.

### Characteristics of Lattice QCD as a Computational Problem

Lattice gauge theories are generally formulated on a regular hypercubic grid of points in space-time. Denote the grid size as  $N_s^3 \times N_t$ , where there are  $N_s$  grid points for each spatial direction and  $N_t$  grid points in the time direction. The spacing between grid points is denoted  $a$ , using units in which the speed of light  $c = 1$ . To get accurate results, the limit  $a \rightarrow 0$ , called the continuum limit, must be taken.

The variables that describe the quarks are multicomponent fields located at the grid points. There are several different ways to formulate the theory for quarks. The most compact method, called staggered or Kogut-Susskind quarks [6], has three complex components:  $\chi^i$ , where  $i = 1, 2, 3$ . The three components correspond to the symmetry group  $SU(3)$  of QCD. Another formulation, used in the original paper of Wilson [4], more closely mimics the original Dirac formulation of fermions used in QED and has a second spinor index on the quark field:  $\psi^{\alpha i}$ , where  $\alpha = 1, 2, 3, 4$  and  $i$  is as before. The MILC code supports calculations with either type of field.

The variables that describe the gauge fields are called gluons in QCD and are the analog of the photon in QED. In the continuum theory, these variables are elements of the Lie algebra of the symmetry group; however, in the lattice theory, elements of the symmetry group itself are used. In the case of the QCD, the gauge fields are  $3 \times 3$  complex, unitary matrices with unit determinant, that is, elements of  $SU(3)$ . There is one such element for each link connecting nearest neighbor grid points. If  $x$  is a grid point in space-time and  $\hat{\mu}$  is a unit vector in one of the four directions, then denote the matrix as  $U(x, x + \hat{\mu})$  or  $U_\mu(x)$ . There is a very important property of these variables that  $U(x, y) = U^\dagger(y, x)$ , where  $\dagger$  denotes the adjoint or transpose conjugate. Thus, links where  $y$  is a neighbor in a negative direction from  $x$  can be expressed in terms of a matrix based at  $y$ . That is,

$$U(x, y = x - \hat{\mu}) = U^\dagger(y, x) = U^\dagger(y, y + \hat{\mu}) = U_\mu^\dagger(y).$$

Thus, only the matrices corresponding to links oriented in the positive directions are stored.

To summarize, at each grid point, four  $3 \times 3$  complex matrices that describe the gluon or gauge fields of QCD are stored. The quarks are described by 3- or  $3 \times 4$ -component complex fields. In Nature, there are six different types of quarks labeled  $u, d, c, s, t$ , and  $b$ . This label has been given the fanciful name flavor. The lightest three quarks  $u, d$ , and  $s$  are most important in the calculations because they are light enough that they can spontaneously appear and disappear from the vacuum as quantum fluctuations. Quark flavors whose vacuum fluctuations are included in the calculation are said to be dynamical quarks. The  $t$  and  $b$  quarks are much too heavy for this effect to be significant for them. Calculations including a dynamical  $c$  quark are just starting to be explored. A quark or antiquark that appears in a bound state whose properties are studied is called a valence quark. All the quarks except for  $t$  have been treated as valence quarks. (The  $t$  quark decays too quickly to form bound states.) Special techniques are needed to treat the valence  $c$  and  $b$  quarks because they are so much heavier than the  $u, d$ , and  $s$  quarks [7].

In lattice QCD the partial differential equations of the continuous theory become finite difference equations on the grid of space-time points. The computations are very easy to vectorize or to parallelize. The MILC code is designed for parallelization via domain

decomposition [8, 9]. Contiguous blocks of grid points are assigned to each node or core of a parallel computer. The MILC code has only one level of hierarchy at present. Below, the word node is used for what is generally each MPI process or core of a job. Each node will be responsible for updating the variables that are stored at all the grid points “owned” by that node. Because of the finite difference equations, updating a variable stored at a site will require knowledge of variables stored at neighboring sites. However, there is one important feature of the gauge symmetry that is essential to QCD. In the finite differences that appear in place of derivatives, one cannot just subtract the values of fields at neighboring grid points; one must first “parallel transport” a field to a neighboring site by multiplying by the gauge matrix  $U$  corresponding to the link joining the sites. Thus,  $\chi(x) - \chi(y)$  is not valid and one would need to compute  $\chi(x) - U_\mu(x)\chi(y)$ , if  $y = x + \hat{\mu}$ , in order to maintain the gauge symmetry of the theory. In practice, this means there are many instances of multiplication of 3- or  $3 \times 4$ -component vectors by  $3 \times 3$  matrices. This allows for some data reuse in what is otherwise a sparse problem. It also encourages the programmer to develop a library of routines that operate on matrices and vectors.

The other issue of dealing with finite differences and domain decomposition is that it is necessary to take into account the possibility of a neighboring value possibly being off-node. In the MILC code, this is dealt with via messaging and setting of pointers. The programmer is shielded from the bother of distinguishing between on-node and off-node neighbors. Three routines are used when accessing values from neighboring grid points. They are start\_gather, wait\_gather, and cleanup\_gather. The initial development of the MILC code was done on the Intel IPSC/860 and Ncube 6400. These machines had long latencies compared with some of the other parallel machines available then. Thus, it was impractical to fetch a neighboring value for a single grid point. These calls are designed to fetch all of the off-node values in a particular direction for a needed field. Actually, because of red-black checkerboarding for many operations, there is also a choice of fetching only from red sites, black sites, or all sites. The code uses the notion of parity rather than checkboard color, and that notation is used below. An even site is one for which the coordinates of the grid point obey  $(x + y + z + t) \bmod 2 = 0$ .

The other sites are odd. Since the same operations are going to be carried out on each site of the grid (or each site of one parity), it is efficient to get all off-node values with a single message from any other node that “owns” (some of) the required data. The start\_gather routine checks to see which data is needed from off-node, and which data this node owns that will be required by other nodes. It allocates buffers for each of the messages that are expected from other nodes and posts the receives. It gathers up the data that will be required by each neighboring node and sends it off. The routine also sets up a pointer for each grid point that either points to the normal on-node location of an on-node neighbor value, or to a location in the message buffer that will contain a message with the data from a neighboring node. Before the neighboring values are accessed through the pointer, the programmer must wait for the wait\_gather routine to return. If possible, other useful work can be done before the call to wait\_gather in order to overlap communication and computation. The routine cleanup\_gather frees the buffers allocated for the messages. Early on it was recognized that a significant amount of time was spent setting up the pointers, so restart\_gather was added to repeat a previous gather reusing the same buffers thus eliminating the need to recalculate the pointers or allocate space for the buffers. (This call is now obsolete, and the feature is subsumed by another routine that hides this from the application programmer.)

Through the use of the routines described above, the application programmer is freed from the burden of dealing with calls to MPI or knowing which grid points have off-node neighbors or which node contains the off-node neighbor. All the information about the domain decomposition is contained in a set of layout routines used in the initial setup routine that allocates the space for the grid of sites. The number of nodes on which a job will run is determined at run time rather than being fixed at compile time. In the original code, a site structure for each grid point contained all of the needed physical variables and some extra information about the layout. The setup routine allocates an array of sites called lattice.

## Coding Style and Examples

A brief introduction to the code with only enough detail to show some snippets of code that loop over lattice sites

and gather values from neighbor sites follows. More complete documentation is available online [10] or in the doc subdirectory of the code distribution. Topics to be discussed include: global variables, data types, site structure, moving around on the lattice, macros, and gathers.

## Global Variables

A number of variables are used so often that they are declared as globals. Each application has an include file lattice.h part of which is below.

```
/* Definition of globals */

#ifndef CONTROL
#define EXTERN
#else
#define EXTERN extern
#endif

/* The following are global scalars */
EXTERN int nx,ny,nz,nt;
/* lattice dimensions */
EXTERN int volume;
/* volume of lattice = nx*ny*nz*nt */

/* Some of these global variables
 are node dependent */
/* They are set in "make_lattice()" */
EXTERN int sites_on_node;
/* number of sites on this node */
EXTERN int even_sites_on_node;
/* number of even sites on this node */
EXTERN int odd_sites_on_node;
/* number of odd sites on this node */
EXTERN int number_of_nodes;
/* number of nodes in use */
EXTERN int this_node;
/* node number of this node */
```

In practice, only the file that contains the main function, always called control.c, defines the variable CONTROL, in which case EXTERN is null and the storage for the global variables is allocated there. In all other files, EXTERN becomes extern, and it is possible to access the global variables. The comments should clarify the meaning of these variables.

## Data Types

In order to be able to conveniently compile for either single or double precision, complex variables are defined in include/complex.h as follows:

```
/* generic precision complex number
 definition */
/* specific for float complex */
typedef struct {
 float real;
 float imag;
} fcomplex;

/* specific for double complex */
typedef struct {
 double real;
 double imag;
} dcomplex;

#if (PRECISION==1)
#define complex fcomplex
#else
#define complex dcomplex
#endif
```

In include/su3.h, these definitions are used to define complex matrices and vectors.

```
typedef struct { fcomplex e[3][3]; } fsu3_matrix;
typedef struct { fcomplex c[3]; } fsu3_vector;
typedef struct { dcomplex e[3][3]; } dsu3_matrix;
typedef struct { dcomplex c[3]; } dsu3_vector;
#if (PRECISION==1)
#define su3_matrix fsu3_matrix
#define su3_vector fsu3_vector
#else
#define su3_matrix dsu3_matrix
#define su3_vector dsu3_vector
#endif
```

In this way, the application programmer can use complex, su3\_matrix, and su3\_vector, and at compile time the value of PRECISION will determine whether single or double precision is used.

There are additional datatypes to deal with Wilson (4-component) quarks. Details can be found in the documentation. The include files complex.h and su3.h also contain prototypes of many functions that use the datatypes just defined.

## Site Structure

The file lattice.h also defines a “site” structure that contains all the physical variables needed at each grid point. This is convenient in that the programmer only has to add another structure to the list, and then all variables are allocated with one call to malloc in a fairly standard setup routine. The section on “Evolution of the Code” describes a disadvantage of this approach. Here is a simple example of a site structure that only contains the gauge matrices and two other matrices and two su3 vectors.

```
typedef struct {
 /* The first part is standard to all
 programs */
 /* coordinates of this site */
 short x,y,z,t;
 /* is it even or odd? */
 char parity;
 /* my index in the array */
 int index;
 /* Now come the physical fields,
 program dependent */
 /* gauge field */
 su3_matrix link[4];
 su3_matrix tempmat1,staple;
 su3_vector phil, phi2;
} site;

/* The lattice is a single global
 variable - (actually this is the
 part of the lattice on this node) */
EXTERN site *lattice;
```

In the file generic/make\_lattice.c there is a function make\_lattice that allocates space for the lattice and for pointers that are used by the gather routines. The variables such as coordinates, parity, and index defined in the site structure are also set up there.

```
void make_lattice() {
 register int i;
 /* scratch */
 short x,y,z,t;
 /* coordinates */
 /* allocate space for lattice, fill
 in parity, coordinates and index */
 node0_printf("Mallocing %.1f MBytes
per node for lattice\n",
 (double)sites_on_node
 * sizeof(site)/1e6);
 lattice = (site *)malloc
(sites_on_node
 * sizeof(site));
 if(lattice==NULL) {
 printf("NODE %d: no room for
lattice\n",this_node);
 terminate(1);
 }

 /* Allocate address vectors */
 for(i=0;i<N_POINTERS;i++) {
 gen_pt[i] = (char **)malloc
(sites_on_node*sizeof(char *));
 if(gen_pt[i]==NULL) {
 printf("NODE %d: no room for
pointer vector\n",this_node);
 terminate(1);
 }
 }

 for(t=0;t<nt;t++) for(z=0;z<nz;z++)
 for(y=0;y<ny;y++) for(x=0;x<nx;x++) {
 if(node_number(x,y,z,t)==mynode()) {
 i=node_index(x,y,z,t);
 lattice[i].x=x; lattice[i].y=y;
 lattice[i].z=z; lattice[i].t=t;
 lattice[i].index =
x+nx*(y+ny*(z+nz*t));
 if((x+y+z+t)%2 == 0)
 lattice[i].parity=EVEN;
 else
 lattice[i].parity=ODD;
 }
 }
}
```

## Moving Around on the Lattice

In the code above, each node is performing a loop over the entire lattice volume, not just over the sites on the node. Two utility functions defined in the layout routines `node_number(x,y,z,t)` and `node_index(x,y,z,t)` return which node a grid point is assigned to and which index it has on the node to which it is assigned. The value of `lattice[i].index` is this site's position in the entire grid, not just the part of the grid assigned to this node. The code shows how to access any variable that is part of the site structure. An alternative access scheme is based on pointers.

```
site *s;
s=&(lattice[i]);
s->x /* is now a pointer to variable
lattice[i].x whose value is the
x coordinate of that grid point */
```

## Macros

Loops should not normally iterate over the whole lattice volume. They should be restricted to the sites on a node, or the sites of one parity on that node. Also, a function that operates on a vector field may sometimes need to use `phi1` as input and other times `phi2`. However, C does not allow the name of a structure member to be passed as an argument. There are macros defined in `include/macros.h` to simplify the coding.

To simplify the loop over the sites on each node, four macros are defined.

```
/* Standard red-black checkerboard */

/* macros to loop over sites of a
 given parity.
Usage:
 int i;
 site *s;
FOREVENSITES(i,s) {
 commands, where s is a
 pointer to the current site
 and i is the index of the
 site on the node
}
*/
#define FOREVENSITES(i,s) \
 for(i=0,s=lattice;
```

```
i<even_sites_on_node;i++,s++)
#define FORODDSITES(i,s) \
 for(i=even_sites_on_node,
 s = &(lattice[i]);
 i<sites_on_node;i++,s++)
#define FORSOMEPARITY(i,s,choice) \
 for(i=((choice)==ODD ?
 even_sites_on_node : 0), \
 s= &(lattice[i]); \
 i< ((choice)==EVEN ?
 even_sites_on_node :
 sites_on_node); \
 i++,s++)
#define FORALLSITES(i,s) \
 for(i=0,s=lattice;
 i<sites_on_node;i++,s++)
```

In each case, loop is restricted to the required sites, and either the integer `i` or the pointer `s` can be used to navigate as described above.

To deal with the issue of not being able to pass the identifier of a structure member, two new concepts “field offset” and “field pointer” are defined. The macros to implement them are also in `macros.h`.

```
/* "field offset" and "field pointer" */

/* used when fields are arguments
 to subroutines */
/* Usage: fo = F_OFFSET(field),
 where "field" is the name of a field
 in lattice.

 address = F_PT(&site , fo),
 where &site is the address of the
 site and fo is a field_offset.

 Usually, the result will have to be
 cast to a pointer of the
 appropriate type.

 (It is naturally a char *).

*/
typedef int field_offset;
#define F_OFFSET(a) \
 ((field_offset)((char *) \
 &(lattice[0]. a))-((char *) \
 &(lattice[0])))
#define F_PT(site , fo)
 ((char *) (site) + (fo))
```

## Gathers

The discussion above, of gathering data from a neighboring site, can now be illuminated with an example. The trace of the product of the four link matrices going around each elementary square of the lattice is a useful quantity. This must be summed over all squares. The two cases of the square being in a purely spatial plane or a plane with time and a spatial direction are treated separately in the following code. Each square is defined by two perpendicular directions and the corner with the smallest coordinates. If  $\hat{\mu}$  and  $\hat{\nu}$  are two different unit vectors, the corners of the square are  $x$ ,  $x + \hat{\mu}$ ,  $x + \hat{\nu}$ , and  $x + \hat{\mu} + \hat{\nu}$ . Traversing the edges of the square starting in the  $\mu$  direction, the expression becomes

$$\text{Tr}(U(x, x + \hat{\mu})U(x + \hat{\mu}, x + \hat{\mu} + \hat{\nu})U(x + \hat{\mu} + \hat{\nu}, x + \hat{\nu})U(x + \hat{\nu}, x)).$$

This can be reexpressed in our other notation as

$$\text{Tr}(U_\mu(x)U_\nu(x + \hat{\mu})U_\mu^\dagger(x + \hat{\nu})U_\nu^\dagger(x)),$$

or, using the cyclic property of the trace, as

$$\text{Tr}\left(U_\nu^\dagger(x)U_\mu(x)U_\nu(x + \hat{\mu})U_\mu^\dagger(x + \hat{\nu})\right).$$

To complete the calculation based at site  $x$ , link matrices based at two nearest neighbor sites are needed. However, from the last expression it is clear that the first matrix multiplication with data that is local to site  $x$  can be done. That work is done while waiting for the messages to arrive.

Below is part of the code generic/plaquette4.c to accomplish this task. A detailed explanation follows the code.

```
void plaquette(Real *ss_plaq,
Real *st_plaq) {
/* su3mat is scratch space of
size su3_matrix */
su3_matrix *su3mat;
register int i, dir1, dir2;
register site *s;
register su3_matrix *m1, *m4;
su3_matrix mttmp;
Real ss_sum, st_sum;
msg_tag *mtag0, *mtag1;
ss_sum = st_sum = 0.0;

su3mat = (su3_matrix *)malloc(sizeof
(su3_matrix)*sites_on_node);
```

```
if(su3mat == NULL)
{
 printf("plaquette: can't malloc
su3mat\n");
 fflush(stdout); terminate(1);
}

for(dir1=YUP;dir1<=TUP;dir1++) {
/* dir1 plays role of mu */
 for(dir2=XUP;dir2<dir1;dir2++) {
/* dir2 plays role of nu */

 mtag0 = start_gather_site
(F_OFFSET(link[dir2]),
sizeof(su3_matrix),
dir1, EVENANDODD,
gen_pt[0]);
 mtag1 = start_gather_site
(F_OFFSET(link[dir1]),
sizeof(su3_matrix),
dir2, EVENANDODD,
gen_pt[1]);

FORALLSITES(i,s) {
 m1 = &(s->link[dir1]);
 m4 = &(s->link[dir2]);
 mult_su3_an(m4,m1,
&su3mat[i]);
}

wait_gather(mtag0);
wait_gather(mtag1);

FORALLSITES(i,s) {
 mult_su3_nn(&su3mat[i],
(su3_matrix *)
(gen_pt[0][i]),
&mtmp);

 if(dir1==TUP)st_sum +=
realtrace_su3
((su3_matrix *)
(gen_pt[1][i]),
&mtmp);
 else ss_sum +=
realtrace_su3
((su3_matrix *)
```

```

 (gen_pt[1][i]),
 &mtmp);
 }

 cleanup_gather(mtag0);
 cleanup_gather(mtag1);
}

g_floatsum(&ss_sum);
g_floatsum(&st_sum);
*ss_plaq = ss_sum /((Real)
(3*nx*ny*nz*nt));
*st_plaq = st_sum /((Real)
(3*nx*ny*nz*nt));

free(su3mat);
} /* plaquette4 */

```

Space is allocated for a field major variable su3mat that can hold one su3 matrix for each site. Then there are nested loops over the two directions that define the plane of the plaquette. The first step is to start the gathers of the two matrices that are based at nearest neighbor points. Note that the call is to start\_gather\_site. (See discussion under “►Evolution of the Code.”) In the first gather, F\_OFFSET is used to specify that the gathered link corresponds to direction dir2, the number of bytes to gather is sizeof(su3\_matrix), and values are gathered from neighbors in the direction dir1. Also, values are gathered from sites with both parities and the pointer array gen\_pt[0] will have the address of the neighbors’ data when the gather is complete. This call will create a “message tag” that has the information required by wait\_gather and cleanup\_gather. Its return value is the address of that data structure. In the second call, the roles of dir1 and dir2 are reversed and the pointer array gen\_pt[1] is used.

In the first FORALLSITES loop, m1 (m4) is set to the address of the link in direction dir1 (dir2), and then mult\_su3\_an is used to take the product of the adjoint link in direction dir2 with the link in direction dir1, and the result is stored in the array su3mat. Note that both the pointer s and index i are used in this loop. The program then waits for the two gathers to complete. Then the intermediate result su3mat can be multiplied on the right by the link pointing in direction dir2

gathered from neighbor dir1, which can now be found using gen\_pt[0]. Since there are no adjoints required, mult\_su3\_nn is used. Note that a single su3 matrix mttmp is used because the calculation of the trace is completed by the call to realtrace\_su3 before proceeding to the next site. In the call to realtrace\_su3, the first argument is the link whose address is in gen\_pt[1]. This routine calculates just the real part of the trace of the product of the adjoint of the first argument and the second argument. Finally, g\_floatsum is called, which is an application-specific global sum, saving the programmer from having to make a call to a specific message passing API, such as MPI. Actually, this code could be improved by doing a single call to g\_vecfloatsum, which would be more efficient. (This routine is not called that often.)

## Communication Options

The MILC code was developed before the advent of MPI and PVM. The initial platforms were the Intel iPSC/860 and Ncube 6400. By having a layer of application-specific calls between the programmer and the native message-passing calls, portability is greatly enhanced. All of the communication routines are contained in a single, target-specific file. In addition to the two targets just mentioned, there was initially code for the Intel iPSC simulator and the “vanilla” version that works with a single thread. Today, only com\_vanilla.c, com\_mpi.c, and com\_qmp.c (which is with the QMP message passing library defined by the USQCD software effort, described below) survive. The vanilla version of the code is remarkably useful for code development. It allows the programmer to develop and debug code in a serial environment that has all the correct syntax for parallel execution.

## Assembly Code Options

Many of the floating point operations within a QCD code involve  $3 \times 3$  matrices and vectors with multiples of 3 components. The code includes a library of such basic operations on matrices and vectors. To improve performance, some of the routines that take the most time are written in assembly code. At various times, assembly code has been developed for Intel i860, DEC Alpha, IBM RS6000, Cray T3D, Cray T3E, and Intel SSE. The code has been ported to the QCDOC special purpose computer. For the IBM Blue Gene/L and P, assembler

support is available through the USQCD libraries discussed below. Often the performance improvement from the assembly code is notable when the data is in the cache, but somewhat limited when the data must be fetched from memory.

### **Evolution of the Code**

The MILC code had its origins nearly 20 years ago and has evolved to include new applications and algorithms. It has also had to evolve to accommodate changes to computer architecture. One important change has been the widening of processor cache lines. The original MILC code was “site-major” in that all the variables for a given grid point or site are stored together. With a narrow cache line, at each stage of the calculation the required variables can be accessed from the site structure. However, as the cache line widens, an inefficiency arises when other variables are brought into the cache when they are not needed because they are stored next to the required data. The solution is to create “field-major” data that stores only a single type of variable, with one grid point’s data following another’s. Then there is the chance that data swept into the cache will be used when the code advances to do the calculation for the next grid point. Thus, the original start\_gather has been replaced with start\_gather\_site and start\_gather\_field, so the programmer can gather from either type of variable.

Around 2000, a second level of hierarchy was accommodated by use of OpenMP [11]. This was motivated by a computer that had eight processors, but a switch that only could support four MPI processes. This was not a notable success, but multithreading is being tried again as compilers have improved, and new computers such as the Cray XE6, Blue Waters and Blue-Gene/Q have or will have many more cores (or threads) that can be used in SMP mode. A port of the code to the IBM Cell Broadband Engine was done [12, 13] and work is ongoing to use GPUs as an accelerator [14, 15].

### **USQCD SciDAC Project**

The US Department of Energy under its Scientific Discovery through Advanced Computing (SciDAC) program [16] has been funding development of QCD code through the National Infrastructure for Lattice Gauge Theory project since 2001 [17]. This project unites developers from several of the major QCD groups in the

USA with the aim of sharing code to improve reliability, performance, and portability of QCD codes, and to reduce the barriers to development of new codes. The USQCD libraries [18] include QLA for basic linear algebra, QMP for application specific message passing, QIO for lattice-specific parallel input/output, and QDP for data parallel style computing that combines shifts from neighboring grid points and computation. There is also a QOP library that is highly optimized and (possibly) architecture specific. The MILC code has gradually been evolving to take advantage of these capabilities.

### **Availability of Code: Tutorials**

The MILC collaboration has made its code available to others for many years. In the beginning, the code was distributed on the basis of personal requests. The code is now distributed via a web page [1]. A mailing list has also been established at <http://denali.physics.indiana.edu/mailman/milccode>.

The documentation for the code is available at the site from which the code is distributed. There are also tutorials on the code prepared by Carleton Detar [19]. They include information about use of the SciDAC QCD libraries with the MILC code.

### **Use of MILC Code for Benchmarking**

The MILC code has been run on many parallel machines and is used for benchmarking by a number of organizations. When the Intel Paragon was being developed, slight inconsistencies in what should have been identical runs resulted in a redesign of the motherboards. The code became part of the hardware diagnostics. A MILC application su3\_rmd is part of the NERSC benchmark suite [20] used to evaluate potential purchases. It is also part of the SPEC 2006 CPU benchmark [21] and the SPEC MPI benchmark [22]. For the NSF Track 1 competition, which was won by the NCSA Blue Waters project [23], performance on a QCD problem supplied by members of MILC had to be analyzed by each applicant. The su3\_rmd application was used for this and will be part of the system tests.

### **Public Availability of Gauge Configurations: Gauge Connection, ILDG**

The MILC collaboration has also long made its gauge configurations available to other scientists.

Configurations including the effects of dynamical quarks are expensive to create, but they can be used for many physics projects once they are archived, so it makes sense to share this resource to make the best use of the public investment in the creation of the configurations. The Gauge Connection hosted by NERSC [24] was the initial repository of the configurations. More recently, the International Lattice Data Grid (ILDG) [25] has created a new standard format and a distributed collection of repositories. There are portals for the ILDG in Australia, Europe, Japan, the UK, and the USA. Many collaborations contribute configurations to the repository that are created using a variety of lattice actions.

## Related Entries

- [Cell Broadband Engine Processor](#)
- [Clusters](#)
- [Cray T3E](#)
- [Cray XT3 and Cray XT Series of Supercomputers](#)
- [Domain Decomposition](#)
- [Hypercubes and Meshes](#)
- [IBM Blue Gene Supercomputer](#)
- [MPI \(Message Passing Interface\)](#)
- [NCube](#)
- [NVIDIA GPU](#)
- [PVM \(Parallel Virtual Machine\)](#)
- [QCD \(Quantum Chromodynamics\) Computations](#)
- [QCDOC and QCDSF Computers](#)
- [Vector Extensions, Instruction-Set Architecture \(ISA\)](#)

## Bibliographic Notes and Further Reading

The MILC code is an application suite for lattice field theory. There is a vast physics literature on the subject. There has been an annual conference in the area for 25 years. For many years the proceedings were part of the Nuclear Physics B (Proceedings Supplement) series. Recently, the proceedings have been published online through Proceedings of Science (<http://pos.sissa.it>).

Several textbooks provide an introduction to lattice field theory and the algorithms that are used. They include:

*Quantum Fields on a Lattice*, I. Montvay and G. Münster, Cambridge, 1994; *Lattice Gauge Theories, An Introduction*, Third Edition, H. J. Rothe, World

Scientific, 2005; *Introduction to Quantum Fields on a Lattice*, J. Smit, Cambridge, 2002; *Lattice Methods for Quantum Chromodynamics*, T. DeGrand and C. DeTar, World Scientific, 2006; *Quantum Chromodynamics on the Lattice, An introductory presentation*, C. Gattringer and C. B. Lang, Springer, 2010.

The literature discussing the implementation and performance of lattice field theory codes is not as extensive as the physics literature. However, there is generally a Machines and Algorithms session at the annual Lattice conference. There have been a number of plenary reviews of computers built specifically for lattice QCD as well as articles about individual computers.

At the USQCD Web site [18], there are links to three other code suites Chroma, CPS, and FermiQCD that are publicly available. They are written in C++.

## Bibliography

1. <http://physics.indiana.edu/~sg/milc.html>. There is a link there to the code distribution site: <http://www.physics.utah.edu/~detar/milc>
2. <http://physics.indiana.edu/~sg/milc/milcpubs.pdf>
3. Quigg C (1983) Gauge theories of the strong, weak and electromagnetic interactions. Addison Wesley, New York
4. Wilson K (1974) Confinement of quarks. Phys Rev D 10:2445
5. Bazavov A et al (2010) Nonperturbative QCD simulations with 2+1 flavors of improved staggered quarks. Rev Mod Phys 82:1349 arXiv:0903.3598
6. Kogut JB, Susskind L (1975) Hamiltonian formulation of Wilson's lattice gauge theories. Phys Rev D 11:395
7. Thacker BA, Lepage GP (1991) Heavy-quark bound states in lattice QCD. Phys Rev D 43:196; El-Khadra AX, Kronfeld AS, Mackenzie PB (1997) Massive fermions in lattice gauge theory. Phys Rev D 55:3933
8. Bernard C et al (1991) Studying quarks and gluons on MIMD parallel computers. Int J High Perform Comput Appl 5:61; <http://hpc.sagepub.com/cgi/content/abstract/5/4/61>
9. Bernard C et al (1991) QCD on the iPSC/860. In: Herman HJ, Karsch F (eds) Workshop on Fermion algorithms, World Scientific, Singapore
10. <http://www.physics.utah.edu/~detar/milc/milcv7.html> or <http://www.physics.utah.edu/~detar/milcv7.pdf>
11. Tamhankar S, Gottlieb S (2001) Benchmarking MILC code with OpenMP and MPI. Nucl Phys B (Proc. Suppl.) 94:841
12. Shi G et al (2009) Implementation of scientific computing applications on the Cell Broadband Engine. Sci Program 17:135
13. Shi G, Kindratenko V, Gottlieb S (2008) Cell processor implementation of a MILC lattice QCD application. In: Proceedings of the XXVI international symposium on lattice field theory, Proceedings of science (LATTICE 2008), Williamsburg, p 278

14. Clark MA (2009) QCD on GPUs: cost effective supercomputing. In: Proceedings of the XXV international symposium on lattice field theory, Proceedings of science (LAT 2009) 003
15. Shi G, Gottlieb S, Torok A, Kindratenko V (2011) Design of MILC lattice QCD application for GPU clusters. In: Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium; <http://lattice.github.com/quda/>
16. <http://www.scidac.gov>
17. <http://www.scidac.gov/physics/quarks.html>; <http://www.usqcd.org>; <http://usqcd.fnal.gov>
18. <http://www.usqcd.org/software>
19. <http://www.physics.utah.edu/~detar/kitpc/>; <http://www.physics.utah.edu/~detar/hacklatt/>
20. <http://pdsi.nerc.gov/benchmarks.htm>
21. <http://www.spec.org/cpu2006/Docs/433.milc.html>
22. <http://www.spec.org/auto/mpi2007/Docs/104.milc.html>
23. <http://www.ncsa.illinois.edu/BlueWaters/>; <http://en.wikipedia.org/wiki/BlueWaters>
24. <http://qcd.nerc.gov/>
25. <http://ildg.sasr.edu.au/Plone>

## MIMD (Multiple Instruction, Multiple Data) Machines

ROLF RIESEN<sup>1</sup>, ARTHUR B. MACCABE<sup>2</sup>

<sup>1</sup>IBM Research, Dublin, Ireland

<sup>2</sup>Oak Ridge National Laboratory, Oak Ridge, TN, USA

### Definition

Distributed and parallel computing systems exist in many different configurations. We group them into six categories and show that the architectural characteristics of each category determines the type of application that will run most efficiently and scalability on these systems. Drawing the line between distributed and parallel systems is not easy because the definitions overlap and large systems combine aspects of both.

### Discussion

#### Introduction

Parallel and distributed computing share many traits and distinguishing the two is not easy. Usually, a sense of larger distances is associated with distributed computing, but many topics grouped under this term are relevant for parallel computing as well. Often specific systems are considered distributed computing systems,

e.g., computers working together and connected by the Internet. Instead of trying to define the two terms precisely, we will define classes of parallel computing systems, note their distinctions, and describe what applications and usage models are best suited for each class. We will order these systems according to their characteristics and will then be able to draw a line that separates distributed from what we would consider parallel computing. Nevertheless, since there is some overlap between classes of parallel computing systems, drawing that line of distinction will still leave some room for debate about what a distributed or a parallel computing system is.

### Parallel Computing Systems

Parallel computing has become pervasive and many different platforms exist to execute parallel programs. These platforms are available in the form of a machine that executes the parallel programs of a user, or more commonly, hidden behind a web interface in the form of a parallel service such as search or database access. This section defines six types of general-purpose parallel computing systems. Specialized parallel machines designed to execute one particular application have been built as well but will not be considered as part of the taxonomy described in this paper.

There are architectural differences between these systems that determine what these machines are used for. We will first look at several different parallel computing architectures and then explore the differences between them. After that we look at how these differences dictate usage, availability, performance, and construction (hardware and software).

### Volunteer Computing

Within any large collection of computers, there are usually a few machine that are idle or underutilized. In the 1980s, projects such as Condor enabled programs to use the networked computers in an office or a class room when they were idle. This allowed users to submit programs for execution during the night when these computers were not in use. With the growth of the Internet, the number of compute resources that may be idle and can be accessed remotely is now in the millions.

SETI@Home (Search for Extraterrestrial Intelligence) began in the mid-1990s to harness idle resources on the Internet. Personal computer owners, who want to volunteer some of their compute cycles, can download a screen-saver from the project home page. Whenever a computer is idle for a while, the screen-saver activates. Instead of just showing a moving graphic or blanking the screen, the SETI@Home screen-saver also contacts a centralized server and requests tasks to execute.

The goal of the SETI@Home project is to analyze radio signals from space and detect patterns that are not likely to occur naturally. For each work request, the server sends some data to the client to be analyzed. While the screen-saver is running, it processes the data, sends results back to the server, and requests more work, if it is still idle. Because users may interrupt the screen-saver or turn the computer off, the server keeps track of which data sets have been processed. It sends the same data set to multiple computers to ensure that at least one of them will finish the work, and it compares replicated work to make sure the results are valid.

The SETI@Home project became so successful in attracting volunteers that it could not keep all their computers busy anymore. A follow-on project called the Berkeley Open Infrastructure for Network Computing (BOINC) allows other projects to use volunteer resources. Currently running examples include protein folding, genetic linkage analysis, predicting the performance of quantum computers, finding weaknesses in cryptographic hash functions, creating a three-dimensional model of the Milky Way galaxy, and testing the sensitivity of climate models.

### Grid Computing

Grid computing became popular in the late 1990s. The term grid is used because, similar to an electric power grid, there are multiple producers and consumers which share their resources under predefined agreements. While one data center has surplus compute capacity after the workers leave for the day, another user may be able to buy these compute cycles. Certain operations or applications may perform faster on one type of computer, or can be executed more cheaply on another. The goal of grid computing is to bring consumers and producers together in an organized fashion that governs accounting and responsibilities. Most grid

users are both producers and consumers. While there are some similarities to volunteer computing, and the actual machines in use may be of the same types, grid computing is much more suitable for a commercial environment. Agreements to make a certain number of compute cycles, or type of system, or storage capacity available are formal and do not depend on the goodwill of anonymous users.

Some of the tools and methods used in volunteer and grid computing are similar and can be used interchangeably. However, grid computing differs from volunteer computing in that some of the systems in the grid are used for long-term storage and some resources, such as leased communication lines or particular systems, are dedicated. In volunteer computing, CPU-scavenging or cycle stealing is the norm, without guarantees when services will be available or a task will complete. In grid computing a certain level of Quality of Service (QoS) is expected.

Starting in about 2007, the term *cloud computing* started to be used. Cloud computing shares many similarities with grid computing, may be used in a grid, but also includes some aspects of volunteer computing and peer-to-peer networks. Whether the term will survive and what its exact definition will be seem to depend on the companies and organizations currently promoting and selling cloud computing systems.

M

### Cluster Computers

Cluster computing began in the late 1990s with systems cobbled together from commercial-of-the-shelf (COTS) components. These were typically PC computers with Ethernet connections located under a desk or on a shelving unit. Demand for clusters grew and commercial companies started to sell integrated turnkey systems. The individual computers that make up a cluster are now rack-able servers. On high-end systems Ethernet is replaced by faster and more costly networks such as Infiniband or Myrinet, and some clusters have redundant power supplies. The defining characteristic is that the individual parts can be bought separately and put together by an integrator in almost any fashion a customer requests.

Clusters reside in a single administrative domain. That means all the components that are part of a cluster are owned and managed by a single entity. Decisions

about what application to run on the cluster, who pays the electricity bill, and which individual user get access are made by a single organization. This differs from volunteer and grid computing which span multiple administrative domains. Frequently, some of the nodes that make up a grid are clusters.

Another difference to volunteer and grid computing is that a cluster usually consists of homogeneous components. The type and performance characteristics of CPUs, disks, memory, and network are either identical or compatible inside a single cluster. In a grid the resources are heterogeneous and one task of the system software for grid computing is to match the appropriate resource with the requirements of a given application. For example, if an application needs 64-bit x86 CPUs with at least 2 GB of memory, the system will not schedule that application on systems that lack these features. On the other hand, users of a cluster know its hardware characteristics and select or compile applications accordingly.

## Supercomputers

The term *supercomputer* is poorly defined but usually associated with high-end parallel systems. Because of their high degree of parallelism, these systems are sometimes called Massively Parallel Processors (MPP). Some of these machines fill large rooms, cost millions of dollars, and require huge amounts of electrical power and cooling. What distinguishes them from clusters is that they are designed as a single system, not built from individual, independent components. Early supercomputers used custom-built proprietary processors. That changed with the advent of high-performance general-purpose microprocessors. Today, supercomputers use the same processors that are used in desktop workstations and servers. The network of most of these machines is still custom and much more tightly coupled to the processor and memory subsystem than the COTS network cards that attach to PCs through an I/O bus. Often, multiple networks are present to serve specific functions: high-speed message passing, collective operations, file I/O, diagnostics, and administrative services. Although the line between a cluster and a supercomputer has been blurred, supercomputers stand out due to their higher-performing networks and their higher reliability.

Reliability in larger systems is a key factor that determines the *throughput* of a system. Throughput is how

many applications a system can finish in a unit of time. Supercomputers have dedicated hardware to monitor the whole system, sometimes use a dedicated network for diagnosis and control of components, and have the ability to work around failed components until they can be repaired. This is crucial for systems that exceed tens of thousand processors. With that many components, it is highly likely that at least one of them has failed at any given time, and locating and isolating these components cannot be done efficiently without a dedicated infrastructure and the necessary system software.

## Symmetric Multi Processors

The systems described so far are *distributed-memory* systems. The processors on a node have direct access to local memory, but to access data in the memory of a remote node, the data has to travel across a network in the form of a message before it can be used. Symmetric Multi processors (SMPs) are *shared-memory* machines. Each of the CPUs in an SMP can directly access all of the memory. We will see later that this allows the programming model to change. Instead of moving data to local memory before it can be accessed, SMP programming models can assume that the data is always accessible. Access time to the data may depend on which memory module holds the data.

Smaller SMPs use a memory bus that is used by all CPUs to access any of the memory attached to it. The cost, called *latency*, of accessing any memory location is roughly the same, no matter which CPU accesses which memory block. For larger SMPs, a single bus shared among all CPUs would be a bottleneck and a more sophisticated architecture, such as a crossbar switch, becomes necessary. When memory size and number of CPUs grow large, it becomes impossible to keep memory access times uniform. Non uniform Memory Architectures (NUMA) deal with this problem by keeping some memory local to a CPU. All of the memory is still accessible to all CPUs, but access times for non local memory is higher than the latency to access local memory.

When a large number of CPUs share a single memory, cache coherence becomes an issue which limits further scaling. Hardware to keep cache contents synchronized when many CPUs read and write the same memory location grows exponentially in size and cost. At some point the cost of keeping caches coherent becomes prohibitive. The only way to add more CPUs

to the system is by abandoning cache coherence and allow for non uniform memory access. That approach is taken by the supercomputers described in the previous section.

### Multicore Processors

The last few years have seen the arrival of multicore processors which combine multiple CPU cores on a single integrated circuit. Each core consists of registers, arithmetic and floating-point units, and all the other control logic that we are used to in individual CPUs. As long as Moore's law continues to hold, each new generation of integrated circuits will contain twice as many transistors as the generation eighteen months ago. Since pipelining and superscalar designs are reaching their limits, multiplying the number of cores on an integrated circuit is a fairly simple way of making use of the ever increasing number of transistors available. Parallel programming is difficult and a single more powerful CPU would be preferable. Physical limits, however, force CPU manufacturers to provide their customers with parallel systems on a chip.

The number of cores per processor is increasing and how the cores share memory ports and caches is evolving. With two or four cores it is common to give each core an L1 cache, and share the L2 cache and the ports to memory. With larger numbers of cores, multicore processors face some of the same problems that limit scaling of SMPs. Newer multicore designs will share L2 caches among groups of cores and use a Network on Chip (NoC) to connect the groups to each other and to the available memory ports. In other words, each multicore processor can be an SMP, a distributed memory system, or a combination of the two.

It should be clear by now that multicore processors are parallel systems with many properties in common with SMPs and even distributed memory systems when the number of cores grow large enough. Processors with hundreds of cores are predicted and finding ways to efficiently program these devices is currently occupying the research community and shares many traits with programming the other parallel systems discussed in this section.

### A Spectrum of Parallel Computing Systems

In this section we briefly introduced several different parallel computing platforms. While there are many similarities between them, there are major differences

that make each platform suitable for specific applications and the way they are used. In the next section we will look at the differences in more detail. In the section after that we will investigate how these differences impact various aspects of parallel computing.

We categorized these systems by describing them at a fairly high level of abstraction. In reality, most existing systems span multiple categories and cannot always be assigned to one or another category so easily. For example, volunteer computing makes use of desktop computers, but many of those computers now contain multicore processors. Sometimes they contain two or more multicore processors which make them multicore SMPs. Similarly, the nodes in a computational grid are often clusters or supercomputers. Programming these systems is difficult and different aspects have to be taken into consideration when matching an application to a class of parallel compute system.

### Difference in Architectural Characteristics

We have identified six major classes of parallel computing systems. There are many similarities among these systems, but some architectural differences exist that are important indicators for what uses and type of applications a system is most suitable. We will see that the classes of parallel computing systems can be arranged along axes that have a crucial impact on the use of these systems. One example of such an axis is physical distance between processing elements. In volunteer computing the individual processors can be spread out across the whole world and communication delays are measured in milliseconds and seconds. Sometimes the individual computers participating in a volunteer effort are turned on only at certain hours of the day. This increases response delays to hours. At the other end of the spectrum are multicore processors where a signal from one core to another has to travel fractions of a millimeter. We will now look at several such characteristics that differentiate parallel computing systems.

### Network

Volunteer computing and most grids use the Internet to connect processors into a parallel computer. Geographical distances may be large and the speed-of-light limit for electrical signals comes into play, especially with satellite links. The high overhead of network protocols such as TCP/IP, congestion along links, and slow routers

limit the bandwidth that can be achieved and impose millisecond latencies or higher.

Moving to clusters and supercomputers, latencies drop to microseconds and bandwidth improves because data has to travel shorter distances and higher capacity links are available. The network topology connecting the processing elements also changes: from the seemingly random graph of the Internet to structured topologies such as 2-D and 3-D meshes, trees, and hypercubes. These topologies provide higher *bisection bandwidth*. It is measured by letting half of the processors in a parallel computer communicate simultaneously with the other half. Assignment to each half is done such that as few links as possible must carry all the traffic. The rich topologies of clusters and supercomputers provide a much higher bisection bandwidth than the Internet where often low-speed links have to be shared among many end points.

Moving to SMPs and multicore computers, latency, bandwidth, and bisection bandwidth further improve. Data can now be shared in memory and latencies to access that data drop to nanoseconds.

How the source and the destination of a data transfer are addressed also changes as we move along the hierarchy. For volunteer and grid computing, each processing element has its own IP address which the routers of the Internet use to direct data packet to the correct destination. Clusters sometimes use the same mechanism, but supercomputers use the location of a node within the structured topology to send data to it. Instead of an address that needs to be decoded and interpreted, supercomputers use directions such as “two up and three to the left” in a 2-D grid. This makes routers simpler and faster. SMPs and multicore which have shared memory use memory addresses to access data. Very fast hardware decodes an address and accesses the appropriate memory location.

CPUs can use memory addresses directly with their load and store instructions. Moving traffic using turn-by-turn directions or determining at what IP address a particular computer resides requires hundreds to tens of thousands of CPU cycles.

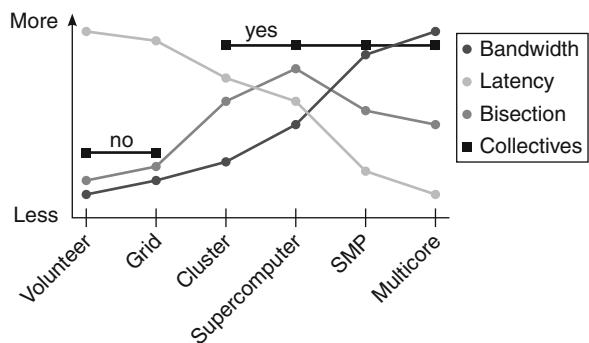
Sometimes it is necessary to distribute data from one processor to all others participating in a parallel computation. This is called a *broadcast*. In a *multicast*, data gets distributed to a select subset of the processors participating in the computation. More complex collective

operations are possible, for example, *reductions* for finding the largest value among the data each processor contributes. These operations are simple memory-to-memory copies in multicore processors and SMPs. Supercomputers sometimes have specialized hardware and a separate network for these kinds of operations. In clusters these operations become more expensive and on the Internet they require a lot of additional bandwidth and latency.

[Figure 1](#) illustrates the network characteristics of the six categories of parallel systems we are considering. We move from left, from the least integrated systems, to the right where we find the most integrated and tightly coupled systems. Doing so, data access latencies decrease, and bandwidth and bisection bandwidths increase. We will later see that the systems to the far right have fewer processors and fewer connections between them. Although the bandwidth keeps increasing, the bisection bandwidth is limited by the number of connections and therefore drops as we move from supercomputers to SMPs and multicore processors.

## Network Interface

Above we have described various network characteristics, but we have not discussed how processors are connected to the network. The network interface is a crucial component in a parallel system. In SMP and multicore systems no network interface is used to access data in memory. Instead, the role of the network interface is



**MIMD (Multiple Instruction, Multiple Data) Machines.**

[Fig. 1](#) Different parallel systems have different network characteristics. No specific values are assigned to each data point since variations in each parallel platform category are large. The data points and lines are meant to show trends

played by the memory controller. Both SMP and multicore systems may have, in addition, network interfaces if they are part of a larger parallel computer such as a cluster or a grid. However, in this section we are concerned with the functions provided by the interface between processor and the data that resides on other processors. For SMP and multicore processors, that interface is the memory controller, and for all other platforms it is the network interface.

We used bandwidth and latency to characterize the networks described in the previous section. Network interfaces are often described using the same metrics. However, it is important to see that these metrics may be different for network interfaces and the links in the network. In some cases the links in a network have a higher bandwidth capacity to accommodate multiple network interfaces.

The name interface suggests two sides: the connection to the network and the connection to the local processors. For the Ethernet cards used in volunteer and grid computing, there is a single connection from the interface to the network. (There can be multiple interfaces connecting a single machine to the network, though.) This is usually true for cluster computing as well, but supercomputers tend to integrate the network interface with a router that has multiple connections to the network. This makes it possible to build the different network topologies seen in these machines. If the interface supports it, data may move through it at multiple times the bandwidth of a single network link, if the routing algorithm makes use of more than a single link into the network. SMP and multicore interfaces, the memory controllers, connect to a memory bus or a switched network that directs data traffic to the appropriate memory bank.

The other side of the interface connects directly to the processors in the case of a memory controller for an SMP or multicore CPU. In supercomputers, the network interface is often connected to the memory bus giving it direct and low-latency access to memory. In clusters and systems that use commodity network interfaces, the interfaces are connected to an I/O bus and treated similarly to other I/O devices in the system.

In addition to bandwidth and latency, *message rate* is another important measure for network interfaces. Although latency may be low in some interfaces, it may not be possible to send another message right away. This

gap between messages limits how many messages can be sent per second. Message rate is determined by latency and the processing capabilities of the system and the network interface. Generally, moving from volunteer computing to multicore CPUs, message rate increases.

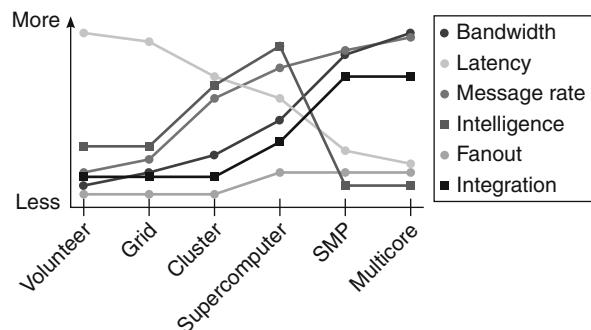
Simple Ethernet interfaces process one packet at a time and have very few additional capabilities. Sophisticated supercomputer network interfaces are small computers themselves with an embedded processor, memory, and fast data-movement hardware to connect to the network and the local memory. The network interface processor can be programmed to make decisions on where in memory to deliver a data packet, how to deal with errors in the network, and even implement collective operations. While the network processor is doing all that, the main CPU is free to continue its application processing.

**Figure 2** shows the trends for network interface characteristics for each class of parallel system we are investigating. Fanout means the number of connections between a network interface and the network. The line labeled intelligence describes the processing capabilities of the network interfaces. Integration indicates how close an interface is to local memory or the CPU.

M

## Remote Data Access

For SMP and multicore systems, accessing data associated with another processor is a simple load or store instruction that can be executed by the processor directly. For all other parallel architectures, remote data access implies sending and receiving a message



**MIMD (Multiple Instruction, Multiple Data) Machines.**

**Fig. 2** Different parallel systems have different network interface characteristics

that contains the data. These two access methods dictate which programming model is most suitable for them.

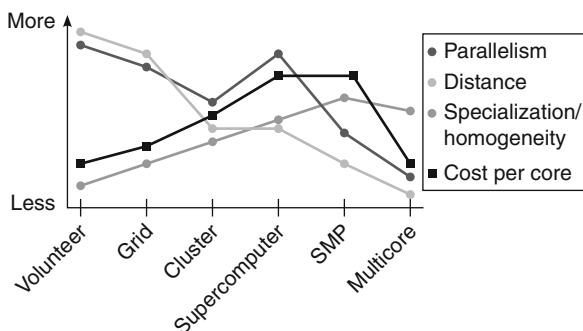
The latency and bandwidth at which remote data can be accessed depend on the access method (load/store or data transmission), and in the case of message passing, the performance and geographical distance of the network. Bulk data transfers benefit from higher bandwidths and are less affected by the latency of a network.

### Other Differences

There are several other differences between these parallel computing systems that are not directly related to their architecture. One of them is size, i.e., the number of processing elements and therefore the amount of available parallelism. In principle, volunteer computing could attract the largest number of processors to solve a particular problem. However, it is not common to gather more than a few thousand computers at a time to work on a single problem. The largest supercomputers have hundreds of thousands of processors and millions are expected in the next few years. SMPs and multicore processors configured internally as an SMP are limited to a few tens by their architecture. In larger multicore processors, the hundreds of cores will use a NoC to exchange data.

We have already mentioned geographical distance which grows smaller and allows faster data access as we move from volunteer and grid computing to multicore processors. [Figure 3](#) illustrates the trends.

Specialization and homogeneity increase moving from volunteer computing to multicore processors. Volunteer computers are of different brands and type, their



**MIMD (Multiple Instruction, Multiple Data) Machines.**

**Fig. 3** Various differences among parallel systems

peripherals differ, and their main purpose is most likely desktop computing. The compute nodes of a supercomputer are homogeneous and the only peripheral is the network interface. Additional nodes, sometimes of a different design, are responsible for I/O and user interactions. Most clusters are homogeneous, but some are built from whatever parts are available. Multicore processors today are homogeneous; all cores are of the same type. Future processors may include specialized cores mixed with general-purpose cores.

Cost per processing element varies. It is not zero for volunteer computing, since someone had to bear the purchase cost of the computer providing the “free” cycles. Since grids and clusters are purchased for a specific purpose and usually contain higher-performing computers and faster networks, their cost per CPU is higher. Supercomputer processors are expensive because of the cost of specialized network interfaces, high-performance network, and the infrastructure that integrates all the parts into a single, reliable system. Cost per processing element drops for multicore processors.

### Architecture Impact

The six major categories of parallel computing platforms differ in some crucial characteristics that determine how these systems are used and for which type of applications they are most suitable. For example, while purchasing multicore chips with a larger core count may be an economical way to increase parallelism, and allows for fast data access, the achievable level of parallelism is limited by the number of cores which technology permits on a single die. Volunteer computing can incorporate as many processors into a computation as are available, but geographical distance and network technology limits how fast remote data can be accessed. In this section we show how differences between these parallel computing platforms impact usage and available features.

### Application Type

*Master/slave* applications are well suited for volunteer computing. One process, designated the master, hands out tasks to computers that become available and ask for work to do. When a slave finishes its task, the master collects the results. It is the master’s role to coordinate tasks, for example, if one task depends on the results of another task. These types of applications are well-suited

for volunteer computing because the master can retry a task until at least one of the slaves finishes it. Note that this programming paradigm also works well on all other architectures discussed in this article. This is the case because the tasks are relatively independent and the limited communication does not tax the network. Systems with more powerful networks and faster data access can easily accommodate this style of computing, albeit at a higher cost.

Tightly coupled applications work better on tightly coupled systems such as SMPs and multicore processors. These types of applications work on a single set of data that is distributed across the memory of all participating processors. All of the tasks running on these processors perform the same operations, each on its portion of the data. Depending on the problem being solved, this may involve more or less communication with processors that are working on neighboring data elements. It is these communications, and the number and pattern of processors that need to communicate with each other, that determine how powerful the network must be for these applications to run efficiently.

## System Software

The system software controlling these systems and the services it provides depend on the type of the system. In volunteer and grid computing each computer runs a full-featured operating system (OS). The software to organize these computers into a parallel system is called *middleware* and runs on top of the OS, making use of its services.

In systems that are dedicated to running parallel applications, such as clusters and supercomputers, the OS is augmented and tailored to provide the services needed, such as message passing, and omits services that are not needed and may be disruptive e.g., email and printer servers. Some supercomputers use specialized lightweight OSs to limit administrative overhead and make resources more directly available to applications. In SMP and multicore computers a single instance of a full-featured OS controls all resources present.

## Programming Model

The programming model chosen to write an application depends on the problem to be solved as well as the platform that will execute the application. The two main

paradigms currently in use are explicit message passing and shared-memory programming. The former is well suited for applications that need to control where (in which memory) data resides and can coordinate data transfers such that the data is available locally when it is needed. Message passing maps well to the networked architectures, but also executes well on shared memory machines where the explicit mapping of data to memory regions often leads to better cache performance than when a compiler determines data layout.

Shared memory programming can only be done on SMPs and multicore processors, unless additional software layers transform memory accesses into data transmissions in distributed memory systems. Several shared-memory programming models provide higher levels of abstraction, but sometimes hide too much of the underlying architecture, such as memory layout, to reach the highest performance possible.

## Usage

The six different parallel computing platforms are used differently and for different purposes. The goal of *capacity* computing is to complete as many jobs in the shortest time possible, i.e., high throughput of jobs. This requires that enough processors are available to solve the problem and efficient scheduling such that all available resources are used.

Supercomputers and some SMPs are used for *capacity* computing. These systems were purchased to solve specific problems in the shortest amount of time possible. These *grand-challenge* problems cannot be solved on smaller or slower systems, at least not in a reasonable amount of time.

Supercomputers and most clusters are *space-shared*. Once nodes have been allocated to an application, the processors on these nodes only execute that application. In *time-sharing*, which is common on all other systems, other applications and tasks also run on the same processors. The OS accomplishes this by giving each process a small slice of time in round-robin fashion, creating the illusion that multiple processes run concurrently on the same CPU. Space-sharing lets an application progress faster since the compute resources are dedicated to that application. Time-sharing makes more efficient use of the available resources, but does not provide optimal performance for an application.

## Summary

The type of parallel platform influences what applications run well on it. In our spectrum from volunteer computing to multicore processors, applications that run efficiently on a network of volunteered compute resources run also well or better on more integrated and faster systems. Applications that run well on tightly integrated systems may not run well or scale to grid and volunteer computing. In general, cost increases as we move from volunteer computing to supercomputers. Cost for SMPs, depending on size, and multicore processors is lower than supercomputers, but the number of processing elements is limited.

In order to find the best system for a given parallel application, all factors, including architecture, performance, cost, and usage model, must be taken into consideration. Few systems are designed to run a single application or serve a single user. Therefore, compromises become necessary.

Distributed and parallel computing have many things in common. Volunteer and grid computing are usually considered distributed computing, while clusters and the other systems in the lineup are considered parallel computers. Many systems today are hybrids, such as clusters made up of SMP and multicore computers. This allows for a higher peak floating-point performance while keeping the cost reasonable. However, there are performance impacts, e.g., multiple CPUs/cores sharing a single network interface, and programming these multi-level parallel machines becomes more difficult.

## Future Directions

Parallel computing has moved from large supercomputers to the desktop. The amount of parallelism will increase with future generation processors and new programming models to make using these machines easier are being researched. Multicore processors will evolve to include specialized cores to accelerate floating-point performance and graphic processing for example. The number of processing units on these chips will continue to grow and they will begin to resemble SMPs, complete with non uniform memory access and complex networks connecting all components on the chip.

Clusters and supercomputers will continue to grow and eventually cross the exaflop barrier. Systems that

large will be plagued by component failures because of the sheer number of these components. This will require drastic measures to make the applications and the systems themselves more resilient to failures. System software will also evolve and be a crucial part to improve resiliency. Running a full-fledged OS on each of a million cores seems wasteful and may limit scalability. Research into lightweight OSs is progressing and may offer a solution to use these compute resources efficiently.

## Related Entries

- ▶ [Bandwidth-Latency Models \(BSP, LoGP\)](#)
- ▶ [Cache-Only Memory Architecture \(COMA\)](#)
- ▶ [Clusters](#)
- ▶ [Distributed-Memory Multiprocessor](#)
- ▶ [Flynn's Taxonomy](#)
- ▶ [Locality of Reference and Parallel Processing](#)
- ▶ [Metrics](#)
- ▶ [Moore's Law](#)
- ▶ [Nonuniform Memory Access \(NUMA\) Machines](#)
- ▶ [Operating System Strategies](#)
- ▶ [Shared-Memory Multiprocessors](#)
- ▶ [Single System Image](#)
- ▶ [SoC \(System on Chip\)](#)

## Bibliographic Notes and Further Reading

Condor was an early software system that allowed harnessing idle cycles on a collection of workstations [7, 8]. A detailed description of Condor and its two-decade history can be found in [14]. An interesting early paper describes “worms” as a mechanism to distribute compute tasks across a set of available computers [11].

Just before the widespread use of cluster computers, a team at the University of California at Berkeley made the case for networks of workstations [3]. They built on the experience of Condor and other such projects, but felt that the larger amount of memory then available, the larger capacity, the reliability of redundant arrays of workstation disks, and the dropping cost of CPUs enabled more wide spread use of such systems.

Soon after, the first Beowulf clusters came into use [12, 13]. The late 1990s saw many cluster projects.

Several of them distinguished themselves from networks of workstations by creating system software that mimicked that of large supercomputers [4].

SETI@home is described in [2] and in more detail in [15] and [6]. The infrastructure to run SETI@home evolved into BOINC [1] which now runs many other volunteer computing projects in addition to SETI@home. The various aspects of grid computing are described by the articles in [5].

Cluster computing has been the first contact with parallel computing and system design for many people. In [9], Pfister discusses many of the issues and concepts only briefly touched on in this introductory article.

## Bibliography

1. Anderson DP (2004) Boinc: a system for public-resource computing and storage. In: GRID '04, Proceedings of the 5th IEEE/ACM international workshop on grid computing, IEEE computer society, Washington, DC, pp 4–10
2. Anderson DP, Cobb J, Korpela E, Lebofsky M, Werthimer D (2002) SETI@home: an experiment in public-resource computing. Commun ACM 45(11):56–61
3. Anderson TE, Culler DE, Patterson DA, and the NOW team (1995) A case for NOW (networks of workstations). IEEE Micro 15(1):54–64
4. Brightwell R, Ann Fisk L, Greenberg DS, Hudson T, Levenhagen M, Maccabe AB, Riesen R (2000) Massively parallel computing using commodity components. Parallel Comput 26(2–3):243–266
5. Foster I, Kesselman C (eds) (1999) The grid: blueprint for a new computing infrastructure. Morgan Kaufmann, San Francisco
6. Korpela E, Werthimer D, Anderson D, Cobb J, Lebofsky M (2001) SETI@homemassively distributed computing for SETI. Comput Sci Eng 3(1):78–83
7. Litzkow M (1987) Remote Unix – turning idle workstations into cycle servers. In: Usenix summer conference, Phoenix, pp 381–384
8. Litzkow M, Livny M, Mutka M (1988) Condor – a hunter of idle workstations. In: Proceedings of the 8th international conference of distributed computing systems, San Jose
9. Pfister GF (1998) In search of clusters: the ongoing battle in lowly parallel computing, 2nd edn. Prentice-Hall, Upper Saddle River
10. Riesen R, Brightwell R, Bridges PG, Hudson T, Maccabe AB, Widener PM, Ferreira K (2009) Designing and implementing lightweight kernels for capability computing. Concur Comput Pract Exp 21(6):793–817
11. Shoch JF, Hupp JA (1982) The “worm” programs – early experience with a distributed computation. Commun ACM 25(3): 172–180
12. Sterling T, Savarese D, Becker DJ, Dorband JE, Ranawake UA, Packer CV (1995) Beowulf: a parallel workstation for scientific computation. In: Banerjee P (ed) Proceedings of the 1995 international conference on parallel processing, Urbana-Champaign pp I:11–I:14
13. Sterling T, Savarese D, Becker DJ, Fryxell B, Olson K (1995) Communication overhead for space science applications on the Beowulf parallel workstation. In: HPDC '95, Proceedings of the 4th IEEE international symposium on high performance distributed computing, IEEE computer society, Washington, DC, p 23
14. Thain D, Tannenbaum T, Livny M (2005) Distributed computing in practice: the Condor experience. Concurr Pract Exp 17(2–4):323–356
15. Werthimer D, Cobb J, Lebofsky M, Anderson D, Korpela E (2001) SETI@home – massively distributed computing for SETI. Comput Sci Eng 3(1):78–83

## MIMD Lattice Computation

► MILC

## MIN

► Networks, Multistage

## ML

► Concurrent ML

## MMX

► Vector Extensions, Instruction-Set Architecture (ISA)

## Model Coupling Toolkit (MCT)

► Community Climate System Model

## Models for Algorithm Design and Analysis

► Models of Computation, Theoretical

## Models of Computation, Theoretical

GIANFRANCO BILARDI<sup>1</sup>, ANDREA PIETRACAPRINA<sup>2</sup>

<sup>1</sup>University of Padova, Padova, Italy

<sup>2</sup>Università di Padova, Padova, Italy

### Synonyms

Computational models; Models for algorithm design and analysis

### Definition

A model of computation is a framework for the specification and the analysis of algorithms/programs. Typically, a model of computation includes four components: an *architectural component*, described as an interconnection of modules of various functionalities; a *specification component*, determining what is a (syntactically) valid algorithm/program; an *execution component*, defining which sequences of states of the architectural modules constitute valid executions of a program/algorithms on a given input; and a *cost component*, defining one or more cost metrics for each execution. A model of computation can be evaluated with respect to three conflicting requirements: the ease of algorithm/program design and analysis (*usability*); the ability of estimating, from the cost metrics provided by the model, the actual performance of a program on a specific real platform (*effectiveness*) and on a class of platforms (*portability*). An ample variety of models of parallel computation are known in the literature, which achieve different trade-offs among the above requirements.

### Discussion

#### Overview

A major objective of Computer Science and Engineering is the development of automatic solutions to computational problems. Formally, a *computational problem*  $\Pi$  is a binary relation between a set of *problem instances* and a set of *problem solutions*. The intended meaning of  $(x,y) \in \Pi$  is that  $y$  is a solution to instance  $x$ . Both instances and solutions are strings from suitable finite alphabets, typically encoding some kind of mathematical objects, such as numbers, matrices, sets, or

graphs. Informally, an *algorithm* is a procedure that specifies how to obtain an output string from an input string by the repeated application of primitive operations (functions) from some chosen set. What constitutes an *execution* of the algorithm on a given input must be unambiguously specified, and the execution process must terminate within a finite number of steps, generally a function of the given instance. Algorithm  $\mathcal{A}$  solves problem  $\Pi$  if, for any input instance  $x$  of  $\Pi$ , it outputs a solution  $y$  such that  $(x,y) \in \Pi$ . In the outlined context, a *model of computation* is a formalism for the specification of algorithms. Such a specification can be referred to as a *program*. A program whose execution terminates on every input specifies an algorithm. In principle, an algorithm is rigorously defined only by means of a program, within a chosen model of computation. In practice, however, when an algorithm is described in the literature, the main ideas are typically presented at a high level, while several ("straightforward, yet tedious") details are left implicit. The details can be provided in different ways, within the chosen model of computation, yielding different programs for what is informally considered to be the same algorithm. Sometimes, an approach to solving a problem is described at an even higher level of abstraction, where even the model of computation is left at least in part unspecified, providing what will be referred to as an *algorithmic strategy*. For example, the quicksort algorithmic strategy to sort a set of records consists in separating the records into two subsets, to be sorted recursively, the keys of the records in the first subset being all smaller than those in the second subset. This strategy can be made specific assuming serial or parallel execution, assuming sequential or random memory access, rearranging the records *in situ* or using a proportional amount of auxiliary storage, etc. Depending on the model of computation, further elements may be specified by an algorithmic description, such as the order of execution of actions (e.g., the order of the two recursive calls in quicksort), which processor is assigned which action (when processors form an explicit component of the model), which data must be stored in which memory region (particularly relevant if memory access time is not uniform), and so on. A variety of different algorithms can then be produced, all referred to as quicksort, in recognition of the common underlying algorithmic strategy.

In the 1930s, several models of computation were proposed including the *recursive functions* of Gödel, the *Turing machines* of Turing, and the  $\lambda$ -*calculus* of Church. These models were soon recognized to be equivalent to each other, in the sense that there are automatic translations of programs from one model to another that preserve the input-output behavior: On a given input, the translated program terminates if and only if the original program terminates and, if so, it produces the same output. Turing machines can be physically realized (taking the infinitude of the storage tape with a grain of salt), for example, as interconnections of Boolean gates and flip flops (taking noise with another grain of salt). Therefore, the effectiveness of the model is not under question. The *Church–Turing thesis* states that Turing machines (as well as any equivalent model) provide a *complete* model of computation, in the sense that any procedure that is intuitively reckoned as executable admits a specification in the model. The thesis has been debated but it is generally accepted, as no counterexamples have been discovered yet. The *halting theorem* of Turing and related results show that no complete model of computation exists if all programs within the model are guaranteed to terminate. In summary, Turing machines or equivalent models determine the boundary between the problems that can be solved algorithmically and those that cannot. Parallel computing does not alter this boundary, although it can greatly enhance the performance of real machines.

An algorithm automates the solution to a problem at the *logical level*: A procedure with a finite description can be applied to solve all of the (typically) infinitely many instances of the problem. But it is also of great practical interest to automate the solution at the *physical level*, by realizing its execution as a physical process. Physical processes utilize resources, which raises the question of *efficiency*, that is, of minimizing, or at least reducing, the *cost* due to resources consumed by the execution process, such as *space*, *time*, and *energy*. Thus, in addition to enabling the specification of an algorithmic strategy, models of computation can provide a framework to specify aspects of the physical execution of that strategy, to analyze the corresponding cost, and ultimately to design efficient algorithms. In summary, a model of computation typically includes the following four components (which may be trivial in some cases):

- An *architectural component*, described as an interconnection of modules of various functionalities. The state sets of each module and the rules that determine the possible state transitions are also provided.
- A *specification component*, determining what is a (syntactically) valid program.
- An *execution component*, defining which sequences of states of the architectural modules constitute valid executions of a given program on a given input.
- A *cost component*, defining one or more cost metrics for each execution.

Several properties are desirable in a model of computation. For example, the following three have been identified in [2] as key properties for a model  $\mathcal{M}$ :

- *Usability*, which refers to the ease with which an algorithm can be specified and its cost analyzed in the model.
- *Effectiveness*, with reference to a target platform  $\mathcal{P}$ , reflects how well the actual cost of execution of programs for  $\mathcal{P}$ , obtained by suitably translating programs specified on  $\mathcal{M}$ , can be derived from the cost metrics of  $\mathcal{M}$ .
- *Portability*, with reference to a (possibly wide) class  $\mathcal{C}$  of target platforms, characterizes the effectiveness of  $\mathcal{M}$  with respect to all of the platforms of  $\mathcal{C}$ .

While usability, which is inherent to the model's definition, is a somewhat subjective property and cannot be easily captured by a quantitative metric, effectiveness and portability are related to the target physical machines and, to some extent, they are amenable to quantitative analysis [2]. Unfortunately, the above three properties are often in conflict with one another. This is perhaps the main reason behind the considerable proliferation of models proposed over the years, resulting from different trade-offs among the various requirements, as briefly discussed below.

The desirable properties of models of computation and their trade-offs have played a role even for sequential computing, a paradigm broadly characterized by the constraint that operations are executed one at the time. Usability considerations have led to the success of the Random Access Machine (RAM) model. As commercial platforms have evolved deep memory hierarchies with half a dozen levels, the uniform memory-access

time of the RAM has made the model less and less effective with respect to these platforms. Several variants to the cost component of the model have been proposed [22], to capture the variability of access time, block transfer, pipelined transfer, and other features of current memory organizations. The scenario has become only more complex with the advent of parallel computers.

With a given execution of a given algorithm one can associate a *functional trace*, an acyclic directed graph whose nodes correspond to the primitive operations specified by the algorithm on input  $x$ , and where an arc from node  $u$  to node  $v$  indicates that the result of operation  $u$  is an operand for operation  $v$ . The functional traces of an algorithm represent what may be called the *logical* aspect of the computation. All models must provide means to specify the functional traces (as a function of the problem input). However, several other aspects arise when the computation is to be realized as a *physical process*, the evolution of the state of a computer. There is ample freedom in how the logical structure of an algorithm can be physically realized and there is ample freedom in choosing which aspects of the physical realization to capture explicitly within the model of computation. Some reflection on these degrees of freedom can provide some orientation when navigating among the multitude of models.

The physical execution of a computation requires various types of unit: *control units*, which coordinate the construction of the trace from the program and the input; *operation units*, which perform the primitive operations defined by the trace; *memory units*, (cells), which store values from the time they are generated to the time(s) they are used; and *communication units*, (channels), which connect different units. Parallel computers have several units of each type whose role in the computation has to be specified. The management of these *resources* is of central importance for parallel computing, and models of computation differ widely in the mechanisms they provide for specifying the use of machine resources.

A general question to be addressed is whether a given resource is better handled automatically or better left under the control of the algorithm designer and programmer. Unfortunately, this question has no simple, universal answer. Automatic resource management typically leads to better usability, and some time even

to greater effectiveness (e.g., when management decisions can benefit from dynamic information available only at execution time). However, there are cases where understanding the mathematical structure of the target computation is crucial to the efficient use of resources. In these cases, the algorithm designer can potentially provide a better resource management. However, the required level of competence may be high and, therefore, the amount of programming effort should be budgeted. Moreover, the performance attained through a poor resource management explicitly coded in the algorithm could turn out worse than the one ensured by a suboptimal automatic management.

The amount of resource control left to the algorithm designer is an interesting criterion for comparing the relative effectiveness of models of computation and can be used to organize some of them in a hierarchy where, proceeding from top to bottom, an increasing number of resource-management aspects is explicitly represented.

## Representative Models

At the top level of the hierarchy of models one can place those at the heart of *functional* [1] and *dataflow languages* [9], in which programs specify only the functional traces of an algorithm. Important cost metrics identifiable at this level are the *number of operations*, often referred to as *work* and denoted  $W$ , and the *length of a critical path*  $L$ , that is the length of a longest path in the functional trace graph. The ratio  $W/L$  provides an indication of the amount of parallelism *implicit* in the algorithm. In fact, an insightful result due to Brent [6] shows that, off-line, the operations of a trace can be scheduled in at most  $2L$  stages each including at most  $W/L$  operations. However, to actually make this parallelism *explicit* when dynamically executing a program is not necessarily trivial.

In physical systems, data have to be stored from the time they are produced to the time they are used, leading to the notion of *storage*, a set of cells that can represent different values at different times. Most models of computation explicitly introduce the notion of storage and let the computation be defined directly in terms of transformations to be applied to the storage content. The Turing machine, the RAM, imperative programming languages such as FORTRAN or C, and most parallel models

reviewed below are examples of storage-based models. In *shared-memory models*, a computation is specified as a sequence of transformations applied to a global *shared memory*. Parallelism can be expressed by parallel loop constructs, whose iterations can be executed simultaneously, or by parallel function calls. *Synchronization* mechanisms are also provided, to constrain the time sequence of the updates, since different sequences will generally produce different results. The work metric of a shared-memory program is the same as that of a functional program encoding the same algorithmic strategy. However, while for a functional program any topological ordering of the trace is legal, for a shared-memory program synchronization instructions may introduce further constraints.

The *Work-Depth model* introduced in [23] is an example of shared-memory model which has been widely used in the literature for the design and analysis of parallel algorithms. An algorithm in the Work-Depth model is described as a standard RAM algorithm enriched with parallel instructions, for example, a parallel loop, which make it possible to specify sets of parallel tasks (single elementary operations as well as complex procedures). Operands and results of the operations reside in a shared memory. The relative order of execution of instructions that belong to different parallel tasks is unconstrained. However, all tasks generated by a given instruction must complete before the next instruction can be executed. This constraint can be modeled by adding arcs to the trace graph. The length of the critical path in this augmented graph is called the *depth* of the shared-memory computation and denoted by  $D$ . Clearly,  $D \geq L$ . In general, implementing the same algorithm within less space may result in a greater depth, hence in a smaller amount of parallelism.

Formulating an algorithm within a shared-memory model, as opposed to a functional language, enables the algorithm designer and programmer to take control of a simple, but important aspect of resource management: the reuse of memory. In a system that executes a functional program, memory is automatically allocated to hold intermediate values, with no burden for the programmer. However, since determining when values will be no longer reused is computationally hard (being in general an undecidable property), an automatic system will often be unable to reuse memory space, in situations where a programmer would know

(by an analysis of the algorithm at hand) that the space could be safely reused. On the other hand, poor programmer's choices regarding memory reuse may inhibit precious parallelism ( $D \gg L$ ), as the proponents of functional programming have often pointed out.

In the hierarchy of models, one level below parallel shared-memory models one can place the *Parallel Random Access Machines* (PRAMs) [10, 12]. A PRAM comprises a set of  $P$  RAM processors; each processor has a unique identification number (id), is equipped with a local memory, and has also direct access to a shared memory. Whereas algorithms developed on shared-memory models specify only which variables are involved in each operation, PRAM algorithms also specify which processor is to carry out the operation. In fact, a PRAM program is syntactically very similar to a RAM program where instructions can make reference to the processor's id, thus enabling the same program to result in different executions on different processors. Instruction mechanisms to dynamically activate and deactivate processors are also provided. Under plausible scenarios, a PRAM could automatically execute a shared-memory algorithm incurring a slowdown logarithmic in the number of processors, to systematically handle the assignment of operations to processors. In a nutshell, a parallel data structure could be employed where operations to be executed are inserted and then deleted when an available processor takes them up for execution. For most algorithms of practical interests, however, this overhead can be avoided by a direct specification of the algorithm within the PRAM model. Since such a specification is often rather straightforward, the distinction between shared-memory and PRAM algorithms is often blurred in the literature, although the distinction ought to be retained, for conceptual clarity.

Shared-memory and PRAM models assume that memory cells form a uniform and global address space, where any subset of addresses can be simultaneously accessed at a cost independent of the subset. Physical machines depart significantly from this idealized assumption. First, memory is partitioned into banks, and while accesses to cells in different banks can be concurrent, accesses to cells in the same bank are serialized. Second, processing elements and memory banks are organized into a network, the topology of which can be a degree of freedom. The result is that the cost of accessing

a subset of memory locations is highly dependent upon the distribution of such subset across the physical memory. To investigate the impact of these physical constraints, a class of *network-of-processors* models has been widely studied. Typically, one such model consists of a network of  $P$  RAM processors where each processor is directly connected to one memory bank. The network specifies point-to-point connections among the processors according to a certain topology. Each processor can execute standard RAM instructions, accessing data in its local memory bank, as well as specific instructions to send and receive messages from the immediate neighbors determined by the network's topology. In terms of cost, it is assumed that a word can be exchanged between direct neighbors in constant time. A variety of interconnection topologies have been considered, such as the complete graph, arrays of various dimensions, hypercubes and hypercubic networks, trees and fat-trees, and expander-based networks. A model based on the complete graph, referred to as *Module Parallel Computer*, was introduced in [18] to investigate the impact of memory granularity decoupled from the further constraint imposed by a limited connectivity. However, due to physical and technological constraints, a small degree, preferably independent of the number of processors, is more realistic.

Particularly suitable for writing algorithms for networks of processors is the *message passing programming paradigm*, in which parallel computation results from the cooperation of a number of sequential processes that exchange messages. A message passing language can be obtained by augmenting a sequential language with send and receive instructions. At execution time, a set of processes is activated, each executing the given program. Processes have a unique identification number, which serves as an address for send and receive instructions and provides a mechanism to differentiate the actions of different processes. Each process has its own memory space, accessible only to that process. A distinct copy of a statically defined variable is available to each process; thus, an instruction involving that variable has different effects for different processes. Message passing has become the main programming paradigm in parallel high performance computing, particularly in the form provided by the *Message Passing Interface* (MPI) library [19]. An MPI program can be

written assuming a specific mapping of processes to the nodes of a given topology and constraining message exchange to immediate neighbors. Strictly adhering to the network-of-processors model presents the unwelcome consequence that the transfer of messages between arbitrary nodes has to be managed in software, a solution that is typically neither convenient nor efficient. To circumvent this obstacle, the design of efficient routing algorithms for various topologies has been widely studied. Today, virtually all commercial systems provide a hardware router that can automatically deliver messages from any node to any other node. The design and programming of algorithms is then relieved of the burden of managing the details of message transport. However, as discussed in the following paragraphs, careful data mapping and operation scheduling remain crucial to performance, resulting in communication patterns that the router can deliver faster.

In the outlined context, it is natural to wonder to what extent should algorithm design be aware of memory granularity and network topology. This question has been investigated, in part, by considering automatic ways of emulating a PRAM computation on a network [14]. Roughly speaking, a rather sophisticated analysis has shown that one PRAM step can be reduced to a constant number of arbitrary data permutations, on an amortized average, or to logarithmically many permutations, in the worst case. The average and the worst-case approaches are substantially different, but share an unwelcome property: In order to limit contention at individual memory banks, the PRAM memory is distributed among the available banks through hash functions or sophisticated schemes based on expander graphs, so that the physical layout of data turns out to be a highly scrambled version of the original PRAM layout, thus forfeiting any *locality structure* captured by the latter. For most networks of practical interest, an arbitrary permutation is rather costly. In contrast, many computations of interest can be mapped into such networks in a way that most memory accesses by a given processor involve its own memory bank or that of nearby processors. Hence, at least asymptotically, there is much to be gained in explicitly optimizing an algorithm for a given topology, by understanding the locality structure and exploiting it by suitable scheduling and mapping of data and operations. For most past and current machines,

this conclusion has been tempered by the fact that the actual data-transfer time is often negligible when compared to the high local latency associated with initiating an interprocessor communication (currently, in the order of microseconds). In the future, however, one can expect that local overheads will decrease (being mostly of a technological and economical nature) while transfer time is destined to increase as machines become larger (being rooted in the more fundamental constraint arising from the finiteness of the speed of light).

Substantial effort has been devoted to the development of efficient network algorithms together with lower-bound techniques to assess their performance [16]. The variety of possible interconnections opens a fertile field for theoretical investigations, but at the same time brings formidable challenges. The prospect of rethinking and reprogramming each algorithm for a number of different topologies is not very attractive in practice. A few avenues have been explored to mitigate this *portability* issue.

One approach is based on efficient simulations of a *guest* network by a *host* network, to enable the automatic porting to the host of algorithms optimized for the guest. A wealth of useful and often nontrivial simulation results have been obtained. Unfortunately, even a (worst-case) optimal general-purpose simulation does not guarantee that an optimal guest algorithm is transformed into an optimal host algorithm, although weaker performance guarantees can often be made, in the presence of some structural similarity between the host and the guest. A second approach is based on the introduction of computational models whose cost component includes parameters aimed at capturing some key performance-related properties of networks. Algorithms on these models can be *parameter aware*. The hope is that networks of interest can be well approximated by the model, with a suitable choice of the parameter values. The *Bulk Synchronous Parallel* (BSP) model [24] and the *LoGP* model [8] are the most popular representatives of this class, but numerous variants and refinements have been also proposed in the literature.

In the BSP model, the architectural component consists of a set of RAM processors that can exchange messages via a communication infrastructure. The execution of an algorithm consists of a sequence of rounds,

called *supersteps*: In each superstep a processor can execute operations on local data and send messages to other processors, which are received only at the beginning of the next supertep. The cost of a superstep is expressed as  $w + gh + \ell$ , where  $g$  and  $\ell$  are model parameters (respectively a measure of inverse bandwidth and of global synchronization time),  $w$  is the maximum number of operations executed by any processor, and  $h$  is the maximum number of messages sent or received by any given node. A shared-memory variant of BSP, called *Queuing Shared-Memory* (QSM) model, was introduced in [11], where in the cost of a superstep the maximum number of reads/writes issued by a processor takes the place of the quantity  $h$ , the latency parameter is dropped and the maximum contention for an individual shared-memory location is explicitly accounted for.

A significant generalization of BSP is the *Decomposable BSP* (D-BSP) model [21, Chap. 2], whose architectural component is based on a binary-tree conceptual structure where each leaf corresponds to a RAM processor and each internal node identifies a cluster of processors residing at the leaves of its subtree. By letting the computation proceed independently within clusters of a given level and charging bandwidth and synchronization costs increasing with the cluster size, the model encourages the algorithm designer to expose a form of data locality called *submachine locality*. The availability of different parameters at different tree levels affords a better approximation of specific processor networks.

As is well known, *data locality* is all important in sequential computing, owing to the hierarchical structure of storage. A strong relation between submachine locality and temporal locality of reference was revealed by efficiently simulating D-BSP algorithms on standard sequential memory hierarchies [21, Chap. 2]. An integration of the memory and network hierarchies arises when accounting for communication delays due to the finite speed of messages, both in the network and within the memory [5]. In recent years, parametric models that capture hierarchical features of both memory and network have been motivated by the increasing popularity and widespread adoption of platforms based on multicore architectures. These models typically feature a hierarchical memory system comprising one large shared memory connected to the processors through several levels of nested smaller memory modules (or

caches) which are shared by progressively smaller subsets of processors. Examples of these models are the *Hierarchical Multi-level caching* (HM) model by [7] and the *Multi-BSP* model by [25].

Models of computation which account for the memory and network hierarchies typically provide parameters that capture salient features of each level of these hierarchies (e.g., bandwidth and latency characteristics, memory or cluster sizes, etc.). Algorithms developed on these models are typically *parameter aware*, in the sense that they adapt their strategies based on the model's parameters in order to maximize data locality. An ambitious goal is the design of algorithms which exhibit (near-)optimal performance for a large range of the model's parameters without using these parameters to influence the course of their computation. The development of algorithms of this kind has been explored for the D-BSP model [3] and for the HM model [7], and the terms *network oblivious* and *multicore oblivious* algorithms have been respectively employed (in analogy with cache-oblivious algorithms which pursue a similar goal in the context of sequential memory hierarchies). It is remarkable that some problems (although not all) do admit oblivious algorithms.

A somewhat radical avenue to solve the challenge of portability, or rather to avoid it altogether, would be enabled by the convergence to just one type of machine architecture and organization for parallel computers. Can such a convergence be reasonably expected? One reason to build different machines is that they may better suited to different computational tasks. Therefore, an interesting theoretical question with important practical ramifications concerns the existence of *universal machines*. More specifically, the question of universality can be cast as follows. Consider the class of all machines with a given value of a cost metric, chosen as a mathematical abstraction of the actual cost of realizing the machine. A machine  $U$  is universal for this class if it can simulate any of its members. The quality of  $U$  can be characterized by two metrics: (a) the machine cost *blowup*, that is, the ratio between the cost of  $U$  and the cost of the machines in the class, and (b) the *slowdown*, that is, the worst-case ratio between the time taken by  $U$  and the time taken by some machine in the class to perform some computation. The existence of universal machines with blowup

and slowdown both very close to one would imply that little performance would be given up by building only machines of the universal type.

For the class of *Bounded Degree Networks* (BDNs), that is processor networks whose degree is constant with respect to the number  $P$  of nodes, if  $P$  is taken as the cost, then it is known that there are universal BDNs with blowup equal to one and with slowdown  $O(\log P)$ . Examples are the *shuffle-exchange* and the *cube-connected-cycles* BDNs, both of degree 3. More generally, a slowdown  $S \leq \log P$ , can be achieved by BDNs with a blowup  $P^{\Theta(1/S)}$  [20]. An alternate measure of cost, which better reflects the scenario of VLSI technology, is the *layout area*. The area of BDNs has been investigated in the *VLSI* model of computation [22], which enables the study of the space layout. A number of area-universal architectures have been proposed, for example, with polylogarithmic blowup and slowdown. These networks exhibit what has been dubbed the *fat-tree* structure, with processors at the leaves and switching subnetworks at the internal nodes. It is instructive to wonder why universal BDNs are not area universal networks. Mathematical analysis shows that the bisection bandwidth of universal BDNs grows proportionally to  $P/\log P$  with the number of processors, which in turn forces the area to grow proportionally to  $P^2/\log^2 P$ . Therefore, the layout of a BDN is scarcely populated with processors, hence it is not effective on algorithms that are more limited by computation than by communication requirements. An example is the multiplication of two square matrices, for which a two-dimensional mesh is a more effective architecture than a shuffle-exchange or a cube-connected-cycles network of the same area. (Technically, the bisection bandwidth of a network is the minimum bandwidth across any cut that splits the network into two subnetworks of (nearly) the same number of nodes.)

The universality results mentioned above, both for the processor and for the area measures of cost, are valid assuming constant delays for communications between direct neighbors in the BDN. It is provable that, in any layout of a network that is universal in this model, the average geometric distance between direct neighbors grows as some function of  $P$ . Therefore, the assumption of constant delays becomes physically unrealistic, for large enough machines. A VLSI-like model where

communication delays are assumed to be proportional to the distance between source and destination has been considered in [4]. In this type of model, a mesh topology of the same number of dimensions as the hosting physical space achieves universality with constant blowup and slowdown. Thus, the mesh topology, already adopted in several supercomputers, appears as a promising candidate to which computer architecture could converge, at least on a large scale.

In balance, it is safe to assume that in the short and in the middle term, a strong convergence among commercial platforms is not to be expected. The design of effective algorithms for such platforms will therefore require different models of computation or at least models that are suitably parametrized. However, even if the objective were to develop efficient algorithms for just one type of machine, a multiplicity of models would likely prove quite useful. Developing an algorithmic strategy at the functional or the shared-memory level, without being concerned at first with too many resource optimization issues, and subsequently refining the algorithm and adapting it to more constraining models may prove a more manageable task than designing directly for the target machine. Of course, as in most design processes that proceed in stages, decisions made in the early stages, based on partial cost metrics, may lead to an overall suboptimal design. However, such a design can often provide a valuable starting point for further efforts. Furthermore, a retrospective on parallel algorithm design would show that, when pursued with a grain of salt, the stage-by-stage approach is generally quite effective.

## Related Entries

- [Bandwidth-Latency Models \(BSP, LogP\)](#)
- [Brent's Theorem](#)
- [BSP \(Bulk Synchronous Parallelism\)](#)
- [Data Flow Graphs](#)
- [Distributed-Memory Multiprocessor](#)
- [Flynn's Taxonomy](#)
- [Interconnection Networks](#)
- [Locality of Reference and Parallel Processing](#)
- [Massive-Scale Analytics](#)
- [MPI \(Message Passing Interface\)](#)
- [Network Obliviousness](#)
- [PRAM \(Parallel Random Access Machines\)](#)

- [PVM \(Parallel Virtual Machine\)](#)
- [SPMD Computational Model](#)
- [Universality in VLSI Computation](#)
- [VLSI Computation](#)

## Bibliographic Notes and Further Reading

In [13, 17], many relevant models for parallel computation are surveyed and critically discussed. The PRAM and network-of-processors models are the topics of two excellent reference books, respectively by J. Jájá [15] and F.T. Leighton [16], which illustrate these models through an ample spectrum of algorithms and theoretical results. The book by J. Savage [22] proposes a comprehensive survey of models of computation, for both sequential and parallel processing, discussing their relative power and several related complexity issues. Finally, the first section of the recent Handbook of Parallel Computing [21], edited by S. Rajasekaran and J. Reif, comprises 16 chapters devoted to models of computation.

## Bibliography

1. Backus JW (1978) Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun ACM* 21(8):613–641
2. Bilardi G, Pietracaprina A, Pucci G (1999) A quantitative measure of portability with application to bandwidth-latency models for parallel computing. In: Proceedings of EUROPAR 99. LNCS, vol 1685. Toulouse, pp 543–551
3. Bilardi G, Pietracaprina A, Pucci G, Silvestri F (2007) Network-oblivious algorithms. In: Proceedings of the 21st IEEE Int. Parallel and Distributed Processing Symposium, Long Beach, CA, pp 1–10
4. Bilardi G, Preparata FP (1995) Horizons of parallel computing. *J Parallel Distr Com* 27:172–182. Also appeared in Proceedings of the INRIA 25th Anniversary Symposium, and as Tech. Rep. CS-93-20 Brown University 1993
5. Bilardi G, Preparata FP (1997) Processor-time tradeoffs under bounded-speed message propagation: Part I, upper bounds. *Theor Comput Syst* 30:523–546
6. Brent RP (1974) The parallel evaluation of general arithmetic expressions. *J ACM* 21(2):201–208
7. Chowdhury RA, Silvestri F, Blakeley B, Ramachandran V (2010) Oblivious algorithms for multicores and network of processors. In: Proceedings of the 24th IEEE Int. Parallel and Distributed Processing Symposium, Atlanta, pp 1–12. Best Paper Award (Algorithms Track)
8. Culler DE, Karp R, Patterson D, Sahay A, Santos E, Schauser KE, Subramonian R, Eicken TV (1996) LogP: A practical model of parallel computation. *Commun ACM* 39(11):78–85

9. Dennis JB (1974) First version of a data flow procedure language. In: Proceedings of the Symposium on Programming. LNCS, vol 19. New York, pp 362–376
10. Fortune S, Wyllie J (1978) Parallelism in random access machines. In: Proceedings of the 10th ACM Symp. on Theory of Computing, San Diego, pp 114–118
11. Gibbons PB, Matias Y, Ramachandran V (1999) Can a shared-memory model serve as a bridging-model for parallel computation? *Theor Comput Syst* 32(3):327–359
12. Goldschlager LM (1978) A unified approach to models of synchronous parallel machines. In: Proceedings. of the 10th ACM Symp. on Theory of Computing, San Diego, pp 89–94
13. Goodrich MT (1993) Parallel algorithms column 1: Models of computation. *ACM SIGACT News* 24(4):16–21
14. Harris TJ (1994) A survey of PRAM simulation techniques. *ACM Comput Surv* 26(2):187–206
15. JáJá J (1992) An introduction to parallel algorithms. Addison Wesley, Reading
16. Leighton FT (1992) Introduction to parallel algorithms and architectures: arrays, trees, hypercubes. Morgan Kaufmann, San Mateo
17. Maggs B, Matheson LR, Tarjan RE (1995) Models of parallel computation: a survey and synthesis. In: Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS), vol 2. Honolulu, pp 61–70
18. Mehlhorn K, Vishkin U (1984) Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Inform* 21:339–374
19. Message Passing Interface Forum. <http://www.mpi-forum.org/>
20. Meyer auf der Heide F (1986) Efficient simulations among several models of parallel computation. *SIAM J Comput* 15(1):106–119
21. Reif J, Rajasekaran S (ed) (2007) Handbook of parallel computing: models, algorithms and applications. CRC Press, Boca Raton, FL
22. Savage JE (1998) Models of computation – exploring the power of computing. Addison Wesley, Reading
23. Shiloach Y, Vishkin U (1982)  $O(n^2 \log n)$  parallel MAX-FLOW algorithm. *J Algorithm* 3(2):128–146
24. Valiant LG (1990) A bridging model for parallel computation. *Commun ACM* 33(8):103–111
25. Valiant LG (2011) A bridging model for multi-core computing. *J Comput Syst Sci* 77:154–166

## Modulo Scheduling and Loop Pipelining

ARUN KEJARIWAL<sup>1</sup>, ALEXANDRU NICOLAU<sup>2</sup>

<sup>1</sup>Yahoo! Inc., Sunnyvale, CA, USA

<sup>2</sup>University of California Irvine, Irvine, CA, USA

Loop parallelization is the most critical aspect of any parallelizing compiler, since most of the time spent in

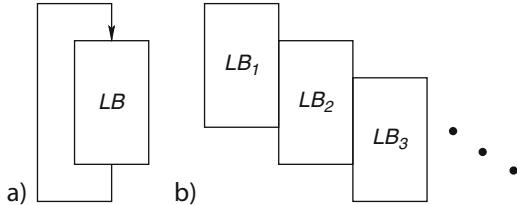
executing a given program is spent in executing loops. In particular, the innermost loops of a program (i.e., the loops most deeply nested in the structure of the program) are typically the most heavily executed. It is therefore critical for any parallelizing (Instruction Level Parallelism (ILP)) compiler to try to expose and exploit as much parallelism as possible from these loops.

Historically, loop unrolling, a standard non-local transformation, is used to expose parallelism beyond iteration boundaries. When a loop is unrolled, the loop body is replicated to create a new loop. Compaction of the unrolled loop body helps exploit parallelism across iterations as the operations in the unrolled loop body come from previously separate iterations. However, in general, loops cannot be fully unrolled, both because of space considerations, and because of the fact that loop bounds are often unknown at compile-time. Furthermore, the extent of parallelism exposed by loop unrolling is restricted by the amount of unrolling.

In order to extract higher levels of parallelism (i.e., beyond unrolling) in software, the various iterations of a loop are pipelined (akin to pipelining of operations in hardware [1, 2]) subject to dependences and resource constraints. This technique is known as *loop pipelining*. An example of loop pipelining is shown in Fig. 1. Figure 1b shows the pipelined execution of the different iterations of the loop shown in Fig. 1a, where  $LB_i$  represents the loop body of the  $i$ -th iteration.

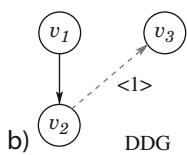
Loop pipelining has its roots in the hand-coding practices for microcode compaction [3, 4]. *Modulo scheduling* [5, 6] pipelines (or overlaps) the execution of successive iterations of a loop such that successive iterations are initiated at regular intervals. The interval between successive iterations is determined based on data dependences and resource constraints. This constant interval between the start of successive iterations is called the *initiation interval (II)*.

Modulo Scheduling was the first method proposed for *loop pipelining* or *software pipelining* (the term “software pipelining”, originally coined to describe a specific transformation [7], is nowadays commonly used interchangeably with loop pipelining to describe any and all transformations that deal with extracting parallelism along the lines discussed in this entry). The objective of loop pipelining is to find a recurring pattern (kernel) across iterations, whether themselves compacted or



**Modulo Scheduling and Loop Pipelining.** Fig. 1 Loop pipelining

```
do i = 1, N
 1: q = p mod i
 2: A[i] = q + r
 3: B[i] = A[i-1] + 1
a) end do
```



**Modulo Scheduling and Loop Pipelining.** Fig. 2 Intra and loop carried dependences

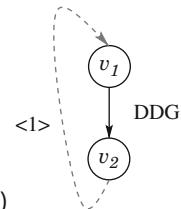
not previously, as they get pipelined and replacing the original loop with this pattern, plus a prologue and epilogue code. The problem of loop pipelining then reduces to how to determine the (best) kernel of this transformed loop.

## Preliminaries

In general, an operation in a loop body is dependent on one or more operations. These dependences may either be data dependences (flow, anti or output) [8] or control dependences [9, 10]. Data dependence between operations of the same iteration is termed *intra-iteration dependence*; data dependence between operations from different iterations is termed *inter-iteration* or *loop-carried dependence*. Subsequently, a dependence refers to a data dependence unless stated otherwise explicitly.

For example, in Fig. 2, the dependence between operation  $v_2$  and operation  $v_1$  is an intra-iteration dependence, whereas the dependence between  $v_2$  and  $v_3$ , shown by a dashed arrow in the data dependence graph (DDG) shown in Fig. 2b, is a loop carried dependence. The dependence distance (in number of iterations) between  $v_2$  and  $v_3$  is shown in angled brackets. Next, some terms used in the rest of the entry are defined.

```
do i = 1, N
 1: B[i] = A[i-1] + r
 2: A[i] = B[i] + 1
end do
```



**Modulo Scheduling and Loop Pipelining.** Fig. 3 Loop recurrences

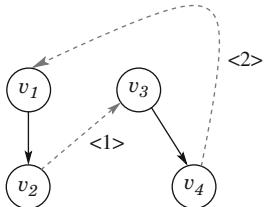
**Definition 1** Given a dependence graph  $G(V, E)$ , a path from an operation  $v_1$  to an operation  $v_k$  is a sequence of operations  $\langle v_1, \dots, v_k \rangle$ , such that  $(v_{i-1}, v_i) \in E$  for  $i = 1, 2, \dots, k$ .

**Definition 2** A cycle in a dependence graph is a path  $\langle v_1, \dots, v_k \rangle$  such that  $v_1 = v_k$  and the path contains at least one edge. A cycle is simple if, in addition,  $v_1, \dots, v_{k-1}$  are distinct.

A loop carried dependence in conjunction with intra-iteration dependences may form a simple cycle in the dependence graph. For example, in Fig. 3, the intra-iteration dependence and the loop carried dependence between the operations  $v_1$  and  $v_2$  form a simple cycle. A loop has a *recurrence* if an operation in one iteration of the loop has a direct or indirect dependence upon the same operation from a previous iteration, e.g., in Fig. 3 operations  $v_i^k$  and  $v_i^{k-1}$  constitute a recurrence, where  $v_i^k$  represents the  $i$ -th operation of the  $k$ -th iteration. In general, a recurrence may span several iterations, i.e., an operation  $v_i^k$  may depend on an operation  $v_i^{k-j}$ , where  $j > 1$ . The existence of a recurrence manifests itself as a simple cycle in the dependence graph. Subsequently, a cycle refers to a simple cycle in a dependence graph.

The *length* of a cycle  $c$ , denoted by  $len(c)$ , is defined as the sum of the latencies of operations in  $c$ . The *delay* of a cycle  $c$ , denoted by  $del(c)$ , is defined as the time interval between the start of operation  $v_1$  and the start of operation  $v_{k-1}$  in a given schedule plus one, where operations  $v_1, v_{k-1} \in c$ , see Definition 1. Let  $dist(c)$  denote the sum of the distances (corresponding to all the loop carried dependences) along  $c$ . For example, in Fig. 4 the path  $\langle v_1, v_2, v_3, v_4, v_1 \rangle$  constitutes a simple cycle. The length of the cycle is 4 and  $dist(c) = 1 + 2 = 3$ .

Note that the delay of a cycle  $c$  may not be equal to its length as the delay of a cycle is dependent on



**Modulo Scheduling and Loop Pipelining.** Fig. 4 A simple cycle

the actual scheduling of operations constituting the cycle. In general, for any cycle  $c$  in a dependence graph,  $\text{len}(c) \leq \text{del}(c)$ .

## Modulo Scheduling Algorithm

The objective of modulo scheduling is to optimize performance (execution time) of a loop over the execution of the entire loop, i.e., beyond the iteration boundaries (unlike basic block scheduling techniques which optimize the performance of a single iteration of the loop). Intuitively, modulo scheduling finds an initiation interval that is larger than what would be required by either resource availability or data dependences. Specifically, modulo scheduling takes a loop without conditionals as an input, looks at the loop body, and computes an initiation interval based on resource utilization, termed *resource-constrained initiation interval (ResII)*. Then, it looks at the recurrences in the dependence graph and evaluates the length of each recurrence. Subsequently, it computes an initiation interval based on recurrences, termed *recurrence-constrained initiation interval (RecII)* (RecII can be computed in polynomial time via, for instance, the Bellman-Ford algorithm [11, 12]). A *minimum initiation interval (MII)* is defined as follows:

$$\text{MII} = \max(\text{ResII}, \text{RecII}) \quad (1)$$

The minimum initiation interval is the smallest initiation interval for which a correct modulo schedule exists. Finally, it generates a schedule, termed a *modulo schedule*, by pipelining (or overlapping) successive iterations of the loop at intervals of MII control steps. The computation of MII based on resource utilization and recurrences ensures that no intra- or inter-iteration dependence is violated, and no resource usage conflict arises between operations of either the same or

distinct iterations. An illustration of modulo scheduling is shown in Fig. 5. Consider the loop and its data dependence graph shown in Figs. 5a and b respectively. Assuming a resource availability as shown in Fig. 5d and a single cycle latency for each resource, the corresponding modulo schedule is shown in Fig. 5c. The reader is advised to verify the schedule with respect to the dependences and resource usage. A detailed description of the modulo scheduling algorithm is presented later in this section.

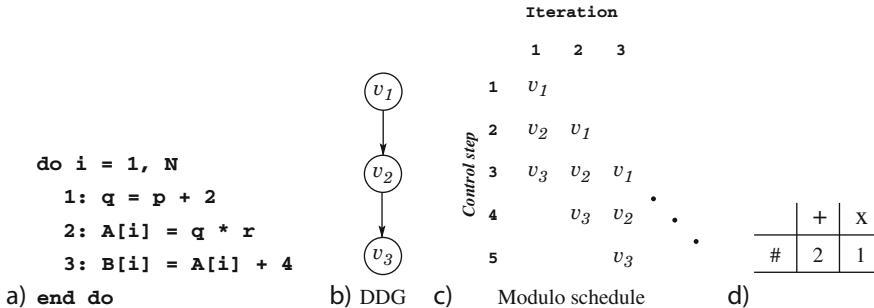
Let  $\mathcal{R}$  represent the library of resources available for scheduling loop  $\mathcal{L}$ . Resources are grouped into classes, each of which consists of identical resources. Let  $\mathcal{R}(i)$  denote the  $i$ -th resource class. Let the number of resources in the  $i$ -th class be denoted by  $\text{num}(i)$ . For example, in Fig. 5d,  $\mathcal{R}(1)$  represents the set of adders, where  $\text{num}(1) = 2$ . Similarly,  $\mathcal{R}(2)$  represents the set of multipliers, where  $\text{num}(2) = 1$ . For simplicity of exposition, each resource is assumed to have single cycle latency. The techniques discussed in this entry can be easily extended to deal with multi-cycle resources [13]. Let  $V$  denote the set of operations in the given loop body. Given a function  $r : V \rightarrow \{1, 2, \dots\}$  such that for each operation  $v \in V$ ,  $r(v)$  gives the number of the resource class used by  $v$ . For example, in Fig. 5d,  $r(v_1) = 1$  and  $r(v_2) = 2$ .

Modulo scheduling, as originally described in [5], makes the following assumptions: (1) The iteration count is known a priori (this, in practice, is not really necessary for the functioning of the algorithm), and (2) The loop body does not contain conditionals or unconditional jumps. It constrains each iteration to have an identical schedule. Modulo scheduling of an innermost loop consists of a number of pre-processing steps as described below.

1. [Generation of conditional free loop body]

In general, a loop body contains multiple control flow paths of which some control paths are executed more frequently than others. In such cases, the most frequently executed control path is selected based on either profiling information or heuristics [14, 15]. This defines the region to be modulo scheduled.

Alternatively, a control dependence can be converted to data dependence via predication [16, 17]. As a result, the resulting loop body looks like a single



**Modulo Scheduling and Loop Pipelining.** Fig. 5 An overview of modulo scheduling

basic block which is amenable to modulo scheduling. In presence of multi-level branches, hierarchical predication [18–20] or predicate matrices [21] may be used to convert multi-level control dependences to data dependences. Early exit branches and multiple branches back to the loop header are handled in a similar fashion.

2. [Minimization of Anti- and Output dependences] Anti- and output dependences can set up recurrence(s) in the data dependence graph which can potentially increase the minimum initiation interval (MII). The *dynamic single assignment form* [22–24] may be used to minimize the anti- and output dependences, which enhances performance by reducing the minimum initiation interval.

Let an integer-valued function PCOUNT and a set-valued function SUCC be defined such that, for each  $v \in V$ ,  $\text{PCOUNT}(v)$  is the number of immediate predecessors of  $v$  not yet scheduled and  $\text{SUCC}(v)$  is the set of all immediate successors of  $v$ . The algorithm for modulo scheduling is presented next.

**Algorithm 1** first computes the lower bound on MII. For this, the algorithm computes ResII and RecII. Since the loop body is yet to be scheduled, hence,  $\text{del}(c)$  cannot be computed. Therefore, as a first approximation RecII is computed based on the length of a cycle. Step 4(a) of the algorithm schedules a single iteration of a loop such that the data dependences are preserved and  $\text{usage}(i)$  of a given resource  $i$  across control steps separated by an interval which is a multiple of MII is less than or equal to  $\text{num}(i)$ . Step 4(b) of the algorithm checks whether MII is greater than RecII for the schedule obtained in Step 4(a). If the check fails then MII

---

**Algorithm 1** (Modulo Scheduling). The input to this algorithm is a dependence graph  $G(V, E)$  of the body a loop  $\mathcal{L}$ . The output is a modulo schedule.

---

1. [Compute resource usage]

Examine the loop body to determine the usage,  $\text{usage}(i)$ , of each resource class  $\mathcal{R}(i)$  by the loop body.

**for** each resource class  $\mathcal{R}(i)$  **do**

$\text{usage}(\mathcal{R}(i)) \leftarrow 0$

**endfor**

**for** each  $v \in V$  **do**

$\text{usage}(r(v)) \leftarrow \text{usage}(r(v)) + 1$

**endfor**

2. [Determine recurrences]

Enumerate all the recurrences in the dependence graph using, for example, any of the algorithms in [25, 26]. Let  $C$  be the set of all recurrences in the dependence graph. Compute  $\text{len}(c) \forall c \in C$ .

3. [Compute the lower bound of minimum initiation interval]

a) [Compute the resource-constrained initiation interval]

$$\text{ResII} = \max_{\forall \mathcal{R}(i) \in \mathcal{R}} \left( \left\lceil \frac{\text{usage}(\mathcal{R}(i))}{\text{num}(\mathcal{R}(i))} \right\rceil \right) \quad (2)$$

b) [Compute the recurrence-constrained initiation interval]

$$\text{RecII} = \max_{\forall c \in C} \left( \left\lceil \frac{\text{len}(c)}{\text{dist}(c)} \right\rceil \right) \quad (3)$$

c) [Compute the minimum initiation interval]

$$\text{MII} = \max(\text{ResII}, \text{RecII}) \quad (4)$$

4. [Modulo schedule the loop]

- a) [Schedule operations in  $G(V, E)$  taking only intra-iteration dependences into account]

Let  $U(i, j)$  denote the usage of the  $i$ -th resource class in control step  $j$

```

for each $v \in V$ do
 compute PCOUNT(v) and SUCC(v)
 /* Assign a control step number to v */
 $\ell(v) \leftarrow 1$
endfor
while $V \neq \emptyset$ do
 for each node $v \in V$ do
 /* Check whether data dependences are
 satisfied */
 if PCOUNT(v) = 0 then
 /* Check that no more than num($r(v)$)
 operations are scheduled on the
 resources corresponding to
 * $\mathcal{R}(r(v))$ at the same time modulo
 MII */
 Let $U \leftarrow \sum_{i=0}^{\lfloor \ell(v)/MII \rfloor} U(r(v),$
 $(\ell(v) - i * MII))$
 while $U > \text{num}(r(v))$ do
 $\ell(v) \leftarrow \ell(v) + 1$
 Update U
 endwhile
 for each $w \in \text{SUCC}(v)$ do
 PCOUNT($w \leftarrow \text{PCOUNT}(w) - 1$
 $\ell(w) \leftarrow \max(\ell(w), \ell(v) + 1)$
 endfor
 /* Schedule the operation */
 $V \leftarrow V - \{v\}$
 endif
 endfor
endwhile

```

- b) [Check the schedule obtained in Step 4(a) w.r.t. recurrences ]

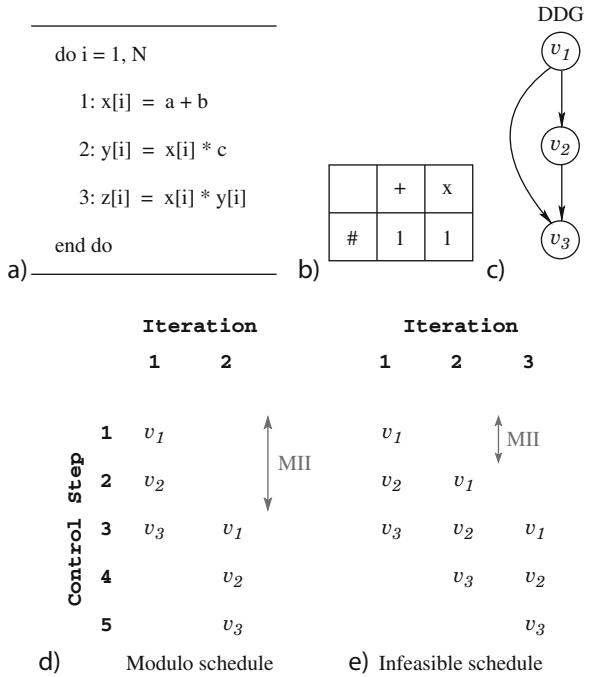
$$II' \leftarrow \max_{\forall c \in C} \frac{\text{del}(c)}{\text{dist}(c)}$$

```

if $MII < II'$ then
 $MII \leftarrow MII + 1$
 Goto step 4 (a)
endif

```

- c) [Pipeline the iterations ]
- Schedule the successive iterations MII control steps apart with each iteration having the same schedule (obtained in Step 4(a)).

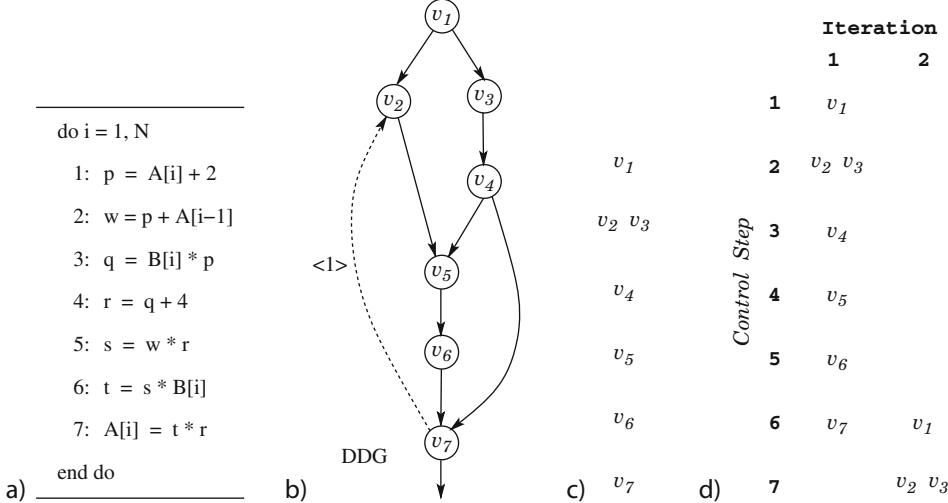


**Modulo Scheduling and Loop Pipelining. Fig. 6** Modulo scheduling in the absence of recurrences

is incremented by 1 and the algorithm iterates through Steps 4(a) and 4(b) until the check is satisfied. Subsequently, successive iterations of the loop are scheduled at an interval of MII. The following example illustrates modulo scheduling in the absence of recurrences.

*Example 1* Consider the loop shown in Fig. 6a and its DDG as shown in Fig. 6c. From the dependence graph one observes that there are no loop carried dependences in the given loop. Therefore, in this case  $MII = \text{ResII}$ .

The number of available resources (adder etc.) per control step is given in Fig. 6b. From Eq. 2,  $\text{ResII}$  for the given resource constraints is 2. The modulo scheduled loop is shown in Fig. 6d, with MII as the initiation interval. In Fig. 6d, control steps 1 and 2 correspond to the prologue of the transformed loop and control steps 3 and 4 constitute the kernel of the transformed loop. From the schedule one observes that 1 cycle is saved per iteration (except the first iteration) in the modulo schedule as compared to the sequential execution of the entire loop. Further overlap of iterations is not feasible as it would violate the resource constraints, e.g., in Fig. 6e, operation  $v_3$  of the first iteration and



**Modulo Scheduling and Loop Pipelining.** Fig. 7 Modulo scheduling in the presence of recurrences

operation  $v_2$  of the second iteration cannot be scheduled in the same control step as there is only one multiplier available.

A subtlety of Step 4(a) of [Algorithm 1](#) bears further discussion. The termination of the inner **while** loop stems from the fact that MII is greater than or equal to ResII. The constraint ( $MII \geq ResII$ ) guarantees that there exists an  $\ell(v)$  for which  $U$  is less than  $num(r(v))$ . In Step 4(a) of [Algorithm 1](#), at any control step, let *ready set* be a set of operations such that for any operation  $v$  in the ready set  $PCOUNT(v) = 0$ . Modulo scheduling involves selection of an operation among the operations in the ready set. Thus, alike list scheduling, [Algorithm 1](#) represents a family of algorithms since the method of selection of an operation from the ready set is not specified. In the presence of multi-function resources, (A multi-function resource can execute more than one type of operations.) determining ResII becomes non-trivial. In such cases, exact ResII can be computed by performing an optimal bin-packing of the reservation tables for all the operations. However, the drawback of such an approach is its exponential complexity.

A cycle  $c$  in the dependence graph imposes the constraint that the time interval between an operation in the current iteration and the same operation  $dist(c)$  iterations later must be at least  $del(c)$ . Consequently, one has the following recurrence constraint:

$$del(c) - RecII \cdot dist(c) < 0 \quad (5)$$

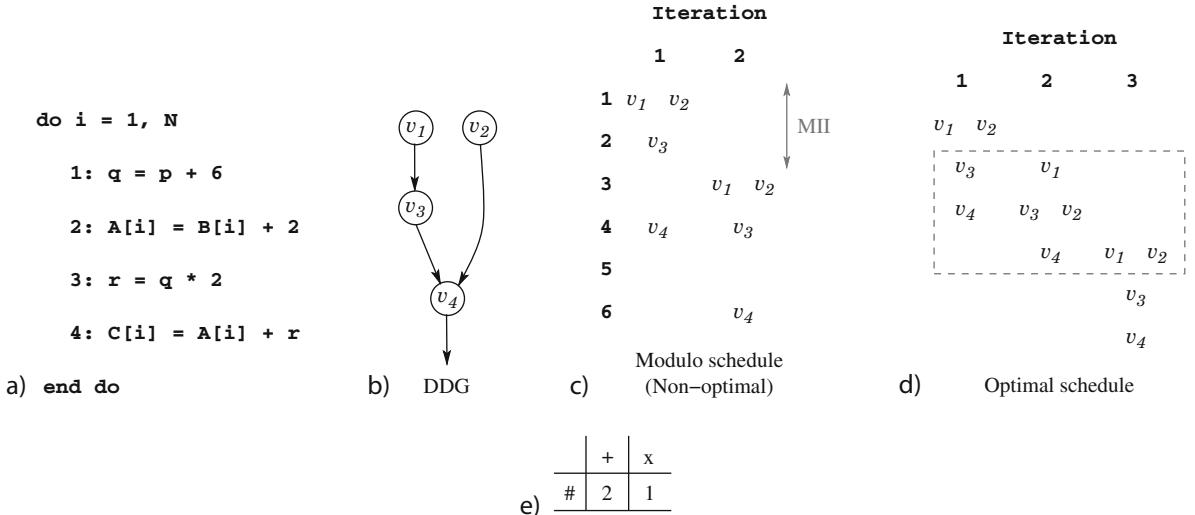
In the presence of multiple recurrences,  $RecII$  is given by [Eq. 3](#). Next, an example of modulo scheduling in the presence of recurrences is presented.

**Example 2** Consider the loop shown in [Fig. 7a](#) and its DDG as shown in [Fig. 7b](#). From [Fig. 7b](#) one observes that the dependence graph contains one cycle,  $c = \{v_2, v_5, v_6, v_7, v_2\}$ .

The length of the cycle ( $len(c)$ ) is 4 and its distance ( $dist(c)$ ) is 1. First, one computes the lower bound on the minimum initiation interval. Assuming the number of resources given in [Fig. 6b](#),  $ResII$  for the DDG given in [Fig. 7b](#) is 3. Since the loop body is yet to be scheduled, hence,  $del(c)$  cannot be computed. Therefore, as a first approximation [Algorithm 1](#) computes  $RecII$  based on the length of a cycle. In this case,  $RecII$  is 4. From [Eq. 1](#), one gets  $MII = 4$ . The corresponding schedule of the loop body is shown in [Fig. 7c](#). From [Fig. 7c](#) one notes that delay of the cycle  $c$  is 5 (=  $del(c)$ ). Since  $del(c) > len(c)$ , an MII of four will lead to a recurrence violation. Therefore, the value of MII is incremented by one and the loop is rescheduled. The final modulo schedule is shown in [Fig. 7d](#). As an exercise, the reader can verify that  $MII = 5$  does not lead to any recurrence violation.

### Remarks: Infeasibility of MII

Modulo scheduling a loop requires computing the recurrence constrained initiation interval ( $RecII$ ) which



**Modulo Scheduling and Loop Pipelining.** Fig. 8 Limitations of modulo scheduling

in itself requires computing the delay of each cycle in the dependence graph of the loop. However, the delay of a cycle is dependent on the schedule for which RecII is known. Several approaches have been proposed to resolve the circular dependence. Rau et al. [5] used linear search to find MII where MII is increased by one iteratively. The FPS compiler [27] used binary search to determine MII where the lower bound is computed using Step 3 of Algorithm 1, and the upper bound is the length of the locally compacted loop body. One could employ an enumerative branch-and-bound search of all possible schedules or use a trial-and-error approach guided by some heuristics. However, the former is computationally very expensive, while the latter does not always guarantee compact schedules. Huff [28] modeled the problem as a minimal cost-to-time ratio problem [29]. Feautrier [30], Govindarajan et al. [31], Eichenberger et al. [32], Altman et al. [33] model modulo scheduling as an integer linear programming [34] problem. However, the exponential complexity of integer linear programming algorithms render such techniques impractical for large loops. Similarly, techniques such as *simulated annealing*, *Boltzmann machine algorithm*, and *genetic algorithms* [35] have been proposed but can become very time expensive for large loops.

## Limitations

Modulo scheduling constrains each iteration to have an identical schedule and schedules successive iterations

at a fixed minimum initiation interval (MII). However, in general, optimal schedule for many loops cannot be achieved by duplicating the schedule of the first iteration at fixed intervals. The constraint that each iteration has the same schedule is ultimately an arbitrary choice, but a fundamental component of modulo scheduling, necessary to ensure convergence in the sense of finding a fixed MII.

For example, consider the loop and its data dependence graph shown in Fig. 8a and b respectively. Let  $v_i^k$  denote the  $i$ -th operation of the  $k$ -th iteration. The modulo schedule of the data dependence graph in Fig. 8b is shown in Fig. 8c. Given a resource library shown in Fig. 8e, ResII for the dependence graph is 2. Since there are no loop carried dependences, therefore,  $MII = ResII$ . Note that operation  $v_4^1$  cannot be scheduled in control step 3 as it would lead to a resource conflict with operations  $v_1^2, v_2^2$  scheduled in control step 3. Thus, a fixed initiation interval in conjunction with the modulo constraint can potentially create “gaps” in the schedule.

In order to extract higher levels of parallelism, one must forgo the constraints imposed by modulo scheduling, i.e., one has to

- Allow successive iterations to be scheduled at different initiation intervals.
- Allow iterations to have different schedules.

An optimal schedule is shown in Fig. 8d. Observe that the first two iterations have different schedules.

Subsequently, iteration 3 is scheduled like iteration 1, iteration 4 is scheduled like iteration 2, and so on. Further, the second and the third iterations are initiated with different initiation intervals. The schedule of the second iteration in Fig. 8d helps close the gap in the first iteration in Fig. 8c, as  $v_4^1$  can now be scheduled in control step 3 without any resource conflicts. Thus, relaxing the constraints (of modulo scheduling) facilitates compaction which yields better performance.

As scheduling of iterations progresses, the execution of operations tend to form a repeating pattern (also known as a *kernel*), as shown in Fig. 8d by the ‘boxed’ set of operations. While such kernels cannot be detected by modulo scheduling, the technique discussed in section “OPT: Optimal Loop Pipelining of Innermost Loops” can detect such kernels.

## Modulo Scheduling with Conditionals

In the previous section modulo scheduling of loops without conditionals was discussed. We now describe how to modulo schedule loops with conditionals.

### Hierarchical Reduction

*Hierarchical reduction* [7] was proposed to make loops with conditionals amenable for modulo scheduling. The hierarchical reduction technique, derived from G. Wood [36], reduces a conditional to a pseudo-operation whose scheduling constraints (both inter-iteration and loop carried dependences) represent the union of the scheduling constraints of the two branches. The significance of hierarchical reduction is:

- Conditionals do not limit the overlapping of different iterations.
- It enables migration of operations outside a conditional around the conditional. Branches of different conditionals can be overlapped via hierarchical percolation scheduling, a technique known as *trailblazing* [37].

The algorithm for hierarchical reduction can be summarized as follows: First, the THEN and ELSE branches of a conditional statement are scheduled independently. Next, the entire conditional is reduced to a pseudo-operation whose scheduling constraints represent the union of the scheduling constraints of the two branches. The latency of the pseudo-operation is the maximum of the length of the two branches; the

resource usage of the pseudo-operation is the maximum of the resource usage of the two branches. Dependences between operations inside the branches and those outside are replaced by dependences between the pseudo-operation representing the conditional and those outside. In the presence of nested control flow, each conditional is scheduled hierarchically, starting with the innermost conditional.

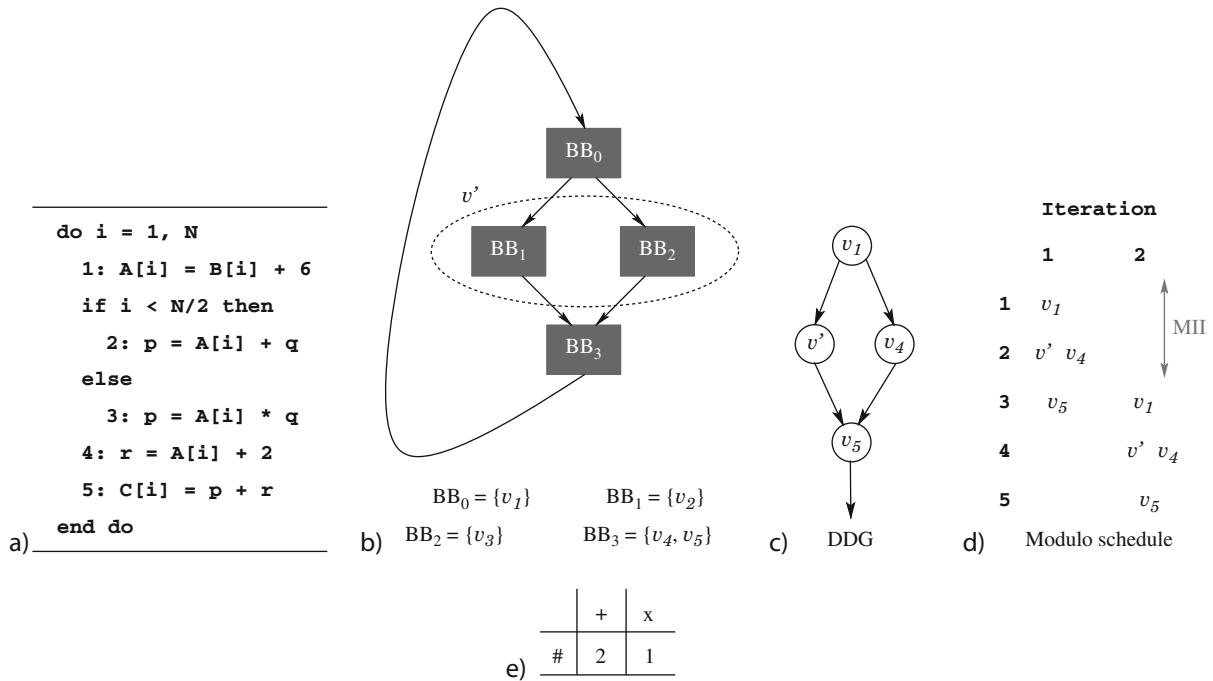
The dependence graph thus obtained is free of conditionals, hence, amenable for modulo scheduling. The dependence graph is then modulo scheduled using Algorithm 1. During code generation two sets of code, corresponding to the two branches, are generated. Any code scheduled in parallel with the conditional is duplicated in both branches.

*Example 3* Consider the loop shown in Fig. 9a. The corresponding control flow graph is shown in Fig. 9b, where  $BB_i$  denotes the  $i$ -th basic block. From Fig. 9a, one notes that the operations  $v_2$ ,  $v_3$ , and  $v_4$  are dependent on operation  $v_1$ . Further, even though  $v_4$  is in basic block  $BB_3$ , it can be scheduled in parallel with the conditional. As discussed above, the conditional, i.e. basic blocks  $BB_1$  and  $BB_2$  are replaced with the pseudo-operation  $v'$ . The dependence graph of the transformed loop is shown in Fig. 9c.

The set of resources used by  $v'$  consists of an adder and a multiplier. Assuming the number of resources given in Fig. 9e, the dependence graph is then modulo scheduled using Algorithm 1 and is shown in Fig. 9d.

### Enhanced Modulo Scheduling

Predication (also known as *if-conversion*) [16] is another technique to convert loops with conditionals into straight line code. Predication removes conditionals by computing a condition for the execution of each operation. Predicated execution assumes architectural support for predicate registers which hold the condition for execution of the operations. A predicate register is specified for each operation and predicate define operations are used to set the predicate registers based on the appropriate condition. Predication-based modulo scheduling techniques [38, 39] schedule operations along all execution paths together. Thus, the MII for predicated execution is constrained by the resource usage of all the loop operations rather than those along



**Modulo Scheduling and Loop Pipelining.** Fig. 9 Modulo scheduling with conditionals

the execution path with maximum resource usage of all paths.

*Enhanced Modulo Scheduling (EMS)* [40] integrates hierarchical reduction and predicated execution to alleviate their limitations. Predication eliminates the need for pre-scheduling the conditional constructs (required in hierarchical reduction); regeneration of conditionals eliminates the need for additional hardware (required by predicated execution), thus EMS can be used on processors without support for predicated execution. An overview of the algorithm is now presented. First, the loop body (with conditionals) is converted into a straight line predicated code using the RK algorithm [17]. The predicated loop body is then modulo scheduled using [Algorithm 1](#) in conjunction with modulo variable expansion [7] to rename registers with overlapping lifetimes. Finally, the predicate define operations are replaced with conditionals.

### Modulo Scheduling with Multiple Initiation Intervals

Hierarchical reduction transforms a loop with conditionals such that the basic algorithm can be used for modulo scheduling the transformed loop. However, in

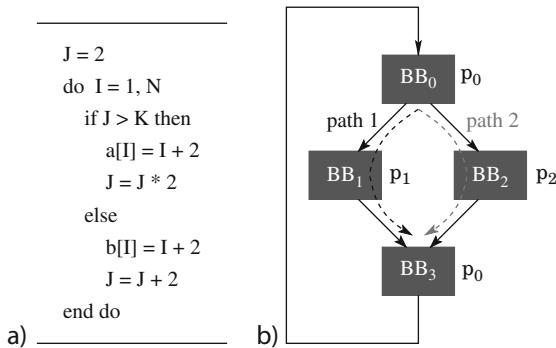
general, two paths of a conditional may imply two different dependences (both intra-iteration and loop carried dependences) and thus, may have different initiation intervals. The initiation interval of the transformed loop is thus constrained by the worst path i.e.,

$$MII = \max_{\forall i} II_{path(i)} \quad (6)$$

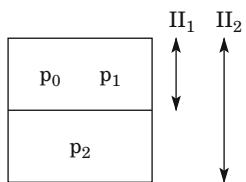
where  $II_{path(i)}$  is the initiation interval of path  $i$ . From [Eq. 6](#) one observes that a path with a shorter MII is penalized due to other paths. Similarly, in the case of predicated execution MII is determined by the sum of operations from all paths. Thus, all paths are penalized due to the fixed MII. As an illustration, consider the example in [Fig. 10a](#).

Assuming an availability of one adder and one multiplier, MII of basic block  $BB_1$  is 1 whereas the MII of basic block  $BB_2$  is 2. A fixed MII-based modulo scheduling would have assigned an MII of 2 to both the paths, which slows down the execution of path 1.

In [41], modulo scheduling with multiple initiation intervals was proposed for architectures with support for predicated execution. In [Fig. 10b](#), the predicate for each basic block is shown next to each basic block. Predicate  $p_0$  is the true predicate. After if-conversion, path 1



**Modulo Scheduling and Loop Pipelining.** Fig. 10 A loop with control flow



**Modulo Scheduling and Loop Pipelining.** Fig. 11 Multiple initiation intervals

corresponds to the operations predicated on  $p_0$  and  $p_1$ , and path 2 to the operations predicated on  $p_0$  and  $p_2$ . The initiation intervals for the two paths are shown in Fig. 11.

From Fig. 11 one notes that during execution of path 1, only operations from path 1 are fetched and executed. However, during the execution of path 2, operations from both the paths are fetched, and operations predicated on  $p_1$  are squashed. Path 2 corresponds to a longer initiation interval  $II_2$  due to larger number of operations. Notice that path 1 is never penalized. The execution paths of a loop body often have different execution frequencies. In order to minimize penalty during the scheduling process, the algorithm assigns higher priorities to the more frequently executed paths, i.e., smaller initiation intervals are assigned to the more frequently executed paths. A detailed description of the algorithm can be found in [41].

## Discussion

Several techniques [35, 42, 43] have been proposed in the past to find the global minimum of the search

space of feasible schedules. However, the high complexity of such approaches makes them computationally very expensive for practical purposes. *Iterative modulo scheduling* [13] algorithm provides a heuristic to control both the search for a valid schedule and the choice of a next state in the search space when the search arrives at a dead end, i.e., when the current schedule is found infeasible. The algorithm is iterative in the sense that it schedules and re-schedules operations to find a “fixed-point” solution which simultaneously satisfies all the scheduling constraints. When the scheduler finds that there is no slot available for scheduling the current operation, it displaces one or more previously scheduled, conflicting operations. These are in turn re-scheduled as the search continues. If the search fails to yield a valid schedule even after a large number of steps, then the initiation interval is increased by a small amount and the entire process is repeated. From Eq. 5, a larger MII delays the execution of a recurrence operation. In the worst case, an MII equal to the schedule length transforms into a list schedule (non-modulo) which is always feasible.

A large number of techniques have been proposed over the last three decades for loop pipelining. In [44], Gasperoni and Schwiegelshohn proposed an algorithm for scheduling loops on parallel processors. In [45], Wang and Eisenbeis proposed a technique called *Decomposed Software Pipelining* wherein software pipelining is considered as an instruction level transformation from a vector of one dimension to a matrix of two dimensions. In particular, the software pipelining problem is decomposed into two subproblems – one is to determine the row numbers of operations in the matrix and another is to determine the column numbers. In [46], Ruttenberg et al. compared optimal and heuristic methods for modulo scheduling. Allan et al. surveyed various techniques for software pipelining in [47].

## Kernel Recognition

All techniques described in this entry can be perceived as kernel recognition techniques. The basic idea behind kernel recognition is to unroll the loop and compact operations from different iterations “as much as possible” (In the presence of vectorizable operations in the loop body the compaction of operations must be constrained in some way to force the emergence of a

repeating pattern.) while looking for a repeating block of instructions. Once a repeating pattern is found in the unrolled, compacted, overlapping bodies of the iterations, the repeating pattern (also referred to as *kernel*) constitutes the loop body of the new loop with a prologue and an epilogue. URPR (Unrolling, Pipelining and Rerolling) algorithm [48] (an extension of the microcode loop compaction algorithm URCR [49]) was one of the early algorithms proposed for loop pipelining based on kernel recognition. URPR assumes that the iteration count is known a priori, the loop body has no conditionals, no subroutine calls, no I/O statements. Some of the other early works include [6, 7, 50]. The techniques differ in type of their underlying assumptions. Broadly speaking, resource constraints (functional units, registers) and conditionals have been the basic distinguishing features. In [51], a loop pipelining technique called *OPT* was proposed to achieve the effect of unbounded unrolling and compaction of a loop. The schedule obtained using *OPT* is *time optimal*, i.e., no transformation of the loop based on the given data dependences can yield a shorter running time for innermost loops without conditionals. Additional hardware assists, such as predicate matrices, specific to loop pipelining were proposed in [21]. In [52], *perfect pipelining* was proposed for loop pipelining of loop with conditionals. Akin to *OPT*, the kernel recognized by perfect pipelining corresponds to unbounded unrolling and compaction; furthermore, perfect pipelining finds a kernel for all paths despite arbitrary flow of control within the loop body. Next the *OPT* algorithm geared toward optimal loop pipelining of innermost loops is discussed.

### OPT: Optimal Loop Pipelining of Innermost Loops

Techniques such as URPR exploit parallelism across the iterations of a loop by unrolling the loop before compaction. If the loop is unrolled (say)  $k$  times then parallelism can be exploited in this unrolled loop body by compacting the  $k$  iterations, but the new loop still imposes sequentiality between every group of  $k$  iterations. Although additional unrolling and compaction can be done to expose higher levels of parallelism, this becomes expensive very rapidly. *OPT* [51] overcomes this problem by achieving the effect of unbounded unrolling and compaction of a loop. The

schedule obtained using *OPT* is *time optimal*, i.e., no transformation of the loop based on the given data dependences can yield a shorter running time for the loop.

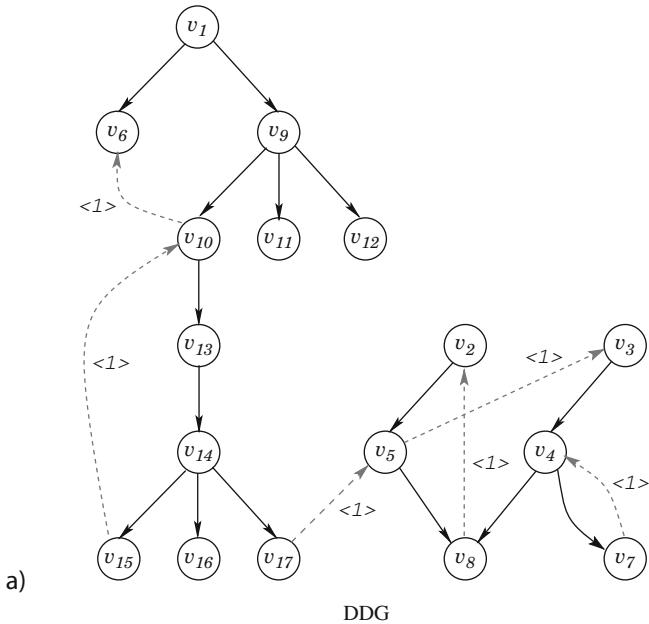
The basic technique examines a partially unrolled loop, say the first  $i$  iterations, and schedules the operations of those iterations as soon as possible. However, the greedy approach may introduce “gaps” – control steps with no operations – between the operations of an iteration due to loop carried dependences; the presence of gaps may in turn forbid the formation of a kernel. In such cases, the gaps are shrunk, i.e., the operations are rescheduled such that there is no effect on the critical path length, in order to force the occurrence of a kernel. A kernel emerges naturally after scheduling a large enough portion of an unrolled loop; the kernel arises as a result of combination of (fixed) dependences and finite number of operations in the loop body. A kernel can be inferred from a greedy schedule and where gaps, if any, are bound by a constant, in at most  $O(n^2)$  iterations [51]. Next, some terms are defined and the algorithm for optimal parallelization of innermost loops is presented. Let  $v_i^k$  denote the  $i$ -th operation of the  $k$ -th iteration.

**Definition 3** Let  $\text{num}(c)$  denote the number of loop carried dependences in a cycle  $c$ . The **slope** of cycle  $c$  is defined as the ratio  $\text{len}(c)/\text{num}(c)$  [53].

The slope of a cycle establishes a bound on the rate that operations in the cycle can be executed. Let  $\text{slope}(v)$  be the maximum slope of any cycle on which  $v$  depends. If  $v$  is not dependent on any cycle, then  $\text{slope}(v) = 0/1$ . When an iteration is scheduled, it is spread across some interval of control steps  $t_1, \dots, t_k$ . Operations from an iteration tend to cluster into groups of mutually dependent operations (referred to as a *region*) with gaps between the regions. For example, in Fig. 12b there are two maximal regions.

**Definition 4** Let  $A_1, \dots, A_j$  be the maximal regions of an iteration, where  $A_i = t_i, \dots, t_{i'}$  and  $t_{i'} < t_{i+1}$  for all  $i$ . Then  $\text{gap}(A_i, A_{i+1}) = t_{i+1} - t_{i'}$ .

In Fig. 12b, iterations 4 and 5 have the same maximal regions; the only difference between the iterations is the size of the gap. Two iterations  $i$  and  $i + c$  are said to be *alike* if they have the same maximal regions and the gaps in iteration  $i + c$  are as large or larger than in



Iteration

| 1                                      | 2                          | 3                                | 4                          | 5                          |
|----------------------------------------|----------------------------|----------------------------------|----------------------------|----------------------------|
| $v_1 \ v_2 \ v_3$                      | $v_1$                      | $v_1$                            | $v_1$                      | $v_1$                      |
| $v_4 \ v_5 \ v_6 \ v_9$                | $v_9$                      | $v_9$                            | $v_9$                      | $v_9$                      |
| $v_7 \ v_8 \ v_{10} \ v_{11} \ v_{12}$ | $v_3 \ v_{11} \ v_{12}$    | $v_{11} \ v_{12}$                | $v_{11} \ v_{12}$          | $v_{11} \ v_{12}$          |
| $v_{13}$                               | $v_2 \ v_4 \ v_{13}$       | $v_{13}$                         | $v_{13}$                   | $v_{13}$                   |
| $v_{14}$                               | $v_5 \ v_6 \ v_7 \ v_{14}$ | $v_6 \ v_{14}$                   | $v_6 \ v_{14}$             | $v_6 \ v_{14}$             |
| $v_{15} \ v_{16} \ v_{17}$             | $v_{15} \ v_{16} \ v_{17}$ | $v_3 \ v_{15} \ v_{16} \ v_{17}$ | $v_{15} \ v_{16} \ v_{17}$ | $v_{15} \ v_{16} \ v_{17}$ |
|                                        | $v_8 \ v_{10}$             | $v_4 \ v_{10}$                   | $v_{10}$                   | $v_{10}$                   |
|                                        |                            | $v_2 \ v_7$                      |                            |                            |
|                                        |                            | $v_5$                            |                            |                            |
|                                        |                            | $v_8$                            | $v_3$                      |                            |
|                                        |                            |                                  | $v_2 \ v_4$                |                            |
|                                        |                            |                                  | $v_5 \ v_7$                |                            |
|                                        |                            |                                  | $v_8$                      | $v_3$                      |
|                                        |                            |                                  | $v_2 \ v_4$                |                            |
|                                        |                            |                                  | $v_5 \ v_7$                |                            |
|                                        |                            |                                  | $v_8$                      |                            |

b)

M

Modulo Scheduling and Loop Pipelining. Fig. 12 A greedy schedule

iteration  $i$ . The gaps in iteration  $i + c$  can be shrunk to the size of the gaps in iteration  $i$  if there is no dependence path from an operation in  $A_k$  of iteration  $i$  to an operation in  $A_j$  ( $k < j$ ) of iteration  $i + c$ , where iterations  $i$  and  $i + c$  are alike with  $j$  maximal regions. For example, in Fig. 12b iterations 4 and 5 are alike and there is no dependence path from the first region of iteration 4 to second region of iteration 5. Thus, it is safe to shrink the gap in iteration 5 to the size of the gap in iteration 4. Though in Fig. 12b the gaps are completely closed, it is not always safe to completely close the gaps. Gaps in an iteration may be completely closed subject to the following condition [54]:

**Condition 1** Let iteration  $i$  and  $i + c$  be alike. Assume that all iterations  $i$  through  $i + c$  have the same number of maximal regions. Let  $A_j^k$  be the  $j$ -th maximal region on the  $k$ -th iteration, where  $i \leq k \leq i + c$ . If

$$\text{gap}(A_j^k, A_{j+1}^k) \leq \text{gap}(A_j^{k+1}, A_{j+1}^{k+1})$$

then the gaps in iteration  $i + c$  and  $i$  may be completely closed.

Next, the OPT algorithm is presented.

The function  $\text{ShrinkGaps}(\mathcal{L}_k, \mathcal{L}_i)$  shrinks the gaps in iteration  $\mathcal{L}_i$  as discussed earlier. In theory, there are loops for which the strategy of making iterations look alike cannot succeed in polynomial time. Let the denominator of  $\text{slope}(v)$  be the period of  $v$ . The length of a kernel based on this approach is at least the least common multiple of the operation periods. If many operations in a loop body have large and relatively prime periods, then the complexity of the algorithm is potentially exponential in  $n$ . The efficiency of the algorithm is dependent on the cost of computing a greedy schedule. This can be easily done using a modified topological sort of the dependence graph. The cost is proportional to the number of operations scheduled.

*Example 4* Consider the data dependence graph shown in Fig. 12a originally proposed as an example in [55]. The corresponding greedy schedule (i.e., operations are scheduled as early as possible subject to data dependences) is shown in Fig. 12b. Even though the greedy schedule corresponds to maximum parallelization, however, from Fig. 12b one observes that a kernel does not emerge as successive iterations are scheduled.

---

**Algorithm 2** The input to this algorithm is a singly nested loop  $\mathcal{L}$  with a dependence graph  $G(V, E)$ . The output is an optimal schedule of  $\mathcal{L}$ . Let  $\mathcal{L}_k$  represent the  $k$ -th iteration of  $\mathcal{L}$ .

---

```

kernelfound ← false
i ← 0
while ¬kernelfound do
 Unroll the loop body once
 i ← i + 1
 Schedule operations in the unrolled loop body
 $G'(V', E')$ as soon as possible
 for each $\mathcal{L}_k \in \mathcal{L}$, where $0 \leq k \leq i - 1$ do
 ShrinkGaps($\mathcal{L}_k, \mathcal{L}_i$)
 endfor
 /* Look for kernel */
 Let s_i be the set of operations at control
 step i in the schedule of
 $G'(V', E')$ and L denote the length of the schedule.
 i ← 1
 S ← \emptyset
 while $S \neq V' \vee i \neq L$ do
 if s_i contains multiple copies of
 an operation v then
 $S \leftarrow \emptyset$
 else if $s_i \cap S \neq \emptyset$ then
 $S \leftarrow s_i$
 else
 $S \leftarrow S \cup s_i$
 endif
 i ← i + 1
 endwhile
 if $S = V$ then
 kernelfound ← true
 endif
endwhile

```

---

| Iteration                      |                        |                            |                        |                        |                        |   |
|--------------------------------|------------------------|----------------------------|------------------------|------------------------|------------------------|---|
| 1                              | 2                      | 3                          | 4                      | 5                      | 6                      | 7 |
| $v_1 v_2 v_3$                  | $v_1$                  | $v_1$                      |                        |                        |                        |   |
| $v_4 v_5 v_6 v_9$              | $v_9$                  | $v_9$                      |                        |                        |                        |   |
| $v_7 v_8 v_{10} v_{11} v_{12}$ | $v_3 v_{11} v_{12}$    | $v_{11} v_{12}$            | $v_1$                  |                        |                        |   |
| $v_{13}$                       | $v_2 v_4 v_{13}$       | $v_{13}$                   | $v_9$                  |                        |                        |   |
| $v_{14}$                       | $v_5 v_6 v_7 v_{14}$   | $v_6 v_{14}$               | $v_{11} v_{12}$        |                        |                        |   |
| $v_{15} v_{16} v_{17}$         | $v_{15} v_{16} v_{17}$ | $v_3 v_{15} v_{16} v_{17}$ | $v_{13}$               | $v_1$                  |                        |   |
|                                | $v_8 v_{10}$           | $v_4 v_{10}$               | $v_6 v_{14}$           | $v_9$                  |                        |   |
|                                | $v_2 v_7$              | $v_{15} v_{16} v_{17}$     | $v_{11} v_{12}$        |                        |                        |   |
|                                | $v_5$                  | $v_{10}$                   | $v_{13}$               | $v_1$                  |                        |   |
| <hr/>                          |                        |                            |                        |                        |                        |   |
| The kernel                     |                        |                            |                        |                        |                        |   |
|                                |                        | $v_2 v_4$                  | $v_{15} v_{16} v_{17}$ | $v_{11} v_{12}$        |                        |   |
|                                |                        | $v_5 v_7$                  | $v_{10}$               | $v_{13}$               | $v_1$                  |   |
|                                |                        | $v_8$                      | $v_3$                  | $v_6 v_{14}$           | $v_9$                  |   |
|                                |                        |                            | $v_2 v_4$              | $v_{15} v_{16} v_{17}$ | $v_{11} v_{12}$        |   |
|                                |                        |                            | $v_5 v_7$              | $v_{10}$               | $v_{13}$               |   |
|                                |                        |                            | $v_8$                  | $v_3$                  | $v_6 v_{14}$           |   |
|                                |                        |                            |                        | $v_2 v_4$              | $v_{15} v_{16} v_{17}$ |   |
|                                |                        |                            |                        | $v_5 v_7$              | $v_{10}$               |   |
|                                |                        |                            |                        | $v_8$                  | $v_3$                  |   |
|                                |                        |                            |                        |                        | $v_2 v_4$              |   |
|                                |                        |                            |                        |                        | $v_5 v_7$              |   |
|                                |                        |                            |                        |                        | $v_8$                  |   |

**Modulo Scheduling and Loop Pipelining.** Fig. 13 Optimal schedule corresponding to the dependence graph in Fig. 12a

In Fig. 12a, the cycle containing the operations  $v_2, v_5, v_8$  has the greatest slope (three). Operations  $v_3, v_4$ , and  $v_7$  are dependent on this cycle and thus have the same slope. All other operations have a slope of zero (0/1). From Fig. 12b one notes that the schedule is eventually split into two groups that repeat every iteration, one with a slope of three, the other with a slope of

zero. In an attempt to generate a kernel, the algorithm reschedules the operations not on the critical path so that they have the same slope as the operations on the critical path. In Fig. 12b, operations with a slope of zero (0/1) in iterations 4 and 5 could be delayed without affecting the length of the schedule. Eliminating the “gaps” in the iterations produces the (optimal) schedule

shown in Fig. 13. The boxed area is the kernel for the loop; scheduling additional iterations (without gaps) reproduces these three control steps. Note that no operation on the critical path has been delayed as a result of re-scheduling.

## Related Entries

### ►Trace Scheduling

## Bibliography

1. Kogge P (1977) The microprogramming of pipelined processors. In: Proceedings of the fourth international symposium on computer architecture, ACM, New York, pp 63–69
2. Kogge PM (1981) The architecture of pipelined computers. Hemisphere Publishing Corporation, Washington
3. Tokoro M, Takizuka T, Tamura E, Yamaura I (1978) A technique of global optimization of microprograms. In: Proceedings of the 11th annual workshop on microprogramming, Pacific Grove, pp 41–50
4. Charlesworth A (1981) An approach to scientific array processing: the architectural design of the AP-120B/FPS-164 family. *IEEE Comput* 14(9):18–27
5. Rau BR, Glaeser CD (1981) Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In: Proceedings of the 14th annual workshop on microprogramming, Chatham, pp 183–198
6. Hsu PS (1986) Highly Concurrent Scalar Processing. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign
7. Lam M (1988) Software pipelining: an effective scheduling technique for VLIW machines. In: Proceedings of the SIGPLAN '88 conference on programming language design and implementation, ACM, Atlanta
8. Kuck D, Kuhn R, Padua D, Leasure B, Wolfe MJ (1981) Dependence graphs and compiler optimizations. In: Conference record of the eighth annual ACM symposium on the principles of programming languages, Williamsburg
9. Ferrante J, Ottenstein K, Warren J (1987) The program dependence graph and its use in optimization. *ACM T Progr Lang Sys* 9(3):319–349
10. Zima H, Chapman B (1991) Supercompilers for parallel and vector computers. Addison-Wesley, New York
11. Bellman R (1958) On a routing problem. *Q Appl Math* 16(1):87–90
12. Cormen TH, Leiserson CE, Rivest RL (1990) Introduction to algorithms. The MIT Press, Cambridge
13. Rau BR (1995) Iterative modulo scheduling. Technical Report HPL-94-115, Hewlett Packard Laboratories, Palo Alto
14. Mahlke SA, Lin DC, Chen WY, Hank RE, Bringmann RA (1992) Effective compiler support for predicated execution using the hyperblock. In: Proceedings of the 25th international symposium of microarchitecture, Portland, pp 45–54
15. Mahlke SA, Hank RE, Bringmann RA, Gyllenhaal JC, Gallagher DM, Hwu WW (1994) Characterizing the impact of predicated execution on branch prediction. In: Proceedings of the 27th international symposium of microarchitecture, San Jose, pp 217–227
16. Allen JR, Kennedy K, Porterfield C, Warren J (1983) Conversion of control dependence to data dependence. In: Conference record of the tenth annual ACM symposium on the principles of programming languages, Austin
17. Park J, Schlansker M (1991) On predicated execution. Technical Report 58–91, Hewlett Packard Laboratories, Palo Alto
18. Tirumalai P, Lee M, Schlansker M (1990) Parallelization of loops with exits on pipelined architectures. In: Proceedings of the 1990 conference on supercomputing, IEEE Computer Society / ACM, New York, pp 200–212
19. Mahlke SA, Chen WY, Hwu WW, Rau BR, Schlansker MS (1992) Sentinel scheduling for VLIW and superscalar processors. *ACM SIGPLAN Notices*, 27(9):238–247
20. Mahlke SA, Chen WY, Bringmann RA, Hank RE, Hwu WW, Rau BR, Schlansker MS (1993) Sentinel scheduling: a model for compiler-controlled speculative execution. *ACM T Comput Syst* 11(4):376–408
21. Milicev D, Jovanovic Z (1998) Predicated software pipelining technique for loops with conditions. In: Proceedings of the 12th international parallel processing symposium, Orlando
22. Feautrier P (1988) Array expansion. In: Proceedings of the 2nd international conference on supercomputing, ACM, St. Malo
23. Feautrier P (1991) Dataflow analysis of scalar and array references. *Int J Parallel Prog* 20(1):23–52
24. Rau BR (1992) Data flow and dependence analysis for instruction level parallelism. In: Proceedings of the fourth workshop on languages and compilers for parallel computing, Springer-Verlag, London, pp 236–250
25. Tiernan JC (1970) An efficient search algorithm to find the elementary circuits of a graph. *Commun ACM* 13(12): 722–726
26. Mateti P, Deo N (1976) On algorithms for enumerating all circuits of a graph. *SIAM J Comput* 5(1):90–99
27. Touzeau RF (1984) A fortran compiler for the fps-164 scientific computer. In: Proceedings of the 1984 SIGPLAN symposium on compiler construction, ACM, New York, pp 48–57
28. Huff RA (1993) Lifetime-sensitive modulo scheduling. In: Proceedings of the SIGPLAN '93 conference on programming language design and implementation, Albuquerque, pp 258–267
29. Lawler EL (1976) Combinatorial optimization: networks and matroids. Holt, Rinehart and Winston, New York
30. Feautrier P (1994) Fine-grain scheduling under resource constraints. In: Proceedings of the seventh annual workshop on languages, compilers and compilers for parallel computers, Ithaca
31. Govindarajan R, Altman ER, Gao GR (1994) Minimizing register requirements under resource-constrained rate-optimal software pipelining. In: Proceedings of the 27th international symposium of microarchitecture, ACM/IEEE, San Jose, pp 85–94
32. Eichenberger AE, Davidson ES, Abraham SG (1995) Optimum modulo schedules for minimum register requirements. In:

- Proceedings of the 9th international conference on supercomputing, ICS '95, ACM, Barcelona, pp 31–40
33. Altman ER, Govindrajan R, Rao GR (1995) Scheduling and mapping: software pipelining in the presence of hazards. In: Proceedings of the SIGPLAN '95 conference on programming language design and implementation, La Jolla
  34. de Dinechin BD (1994) Simplex scheduling: more than lifetime-sensitive instruction scheduling. In: Proceedings of the international conference on parallel architectures and compilation techniques (PACT), Montreal
  35. Reeves CR (1993) Modern heuristic techniques for combinatorial problems. Wiley, New York
  36. Wood G (1979) Global optimization of microprograms through modular control constructs. In: Proceedings of the 12th workshop on microprogramming, IEEE Press, Piscataway, pp 1–6
  37. Novack S, Nicolau A (1995) Trailblazing: a hierarchical approach to percolation scheduling. *Int J Parallel Prog* 23(1)
  38. Rau BR, Yen WL, Yen W, Towle A (1989) The Cydra 5 departmental supercomputer: design philosophies, decisions and trade-offs. *IEEE Comput* 22(1):12–35
  39. Dehnert JC, Hsu PYT, Bratt JP (1989) Overlapped loop support in the Cydra 5. In: Proceedings of the third international conference on architectural support for programming languages and operating systems (ASPLOS-III), ACM, Boston, pp 26–38
  40. Warter NJ, Bockhaus JW, Haab GE, Subramaniam K (1992) Enhanced modulo scheduling for loops with conditional branches. In: Proceedings of the 25th international symposium of microarchitecture, ACM/IEEE, Portland
  41. Warter NJ, Partamian N (1995) Modulo scheduling with multiple multiple initiation intervals. In: Proceedings of the 28th international symposium of microarchitecture, ACM/IEEE, Ann Arbor
  42. Jacobs D, Prins J, Siegel P, Wilson K (1982) Monte carlo techniques in code optimization. In: Proceedings of the 15th workshop on microprogramming, IEEE Press, Piscataway, pp 143–148
  43. De Gloria A, Faraboschi P, Olivieri M (1992) A non-deterministic scheduler for a software pipelining compiler. In: Proceedings of the 25th international symposium of microarchitecture, ACM/IEEE, Portland, pp 41–44
  44. Gasperoni F, Schwiegelshohn U (1992) Scheduling loops on parallel processors: a simple algorithm with close to optimum performance. In: Proceedings of the second joint international conference on vector and parallel processing: parallel processing, pp 625–636
  45. Wang J, Eisenbeis C (1993) Decomposed software pipelining. Technical Report RR-1838, INRIA-Rocquencourt, France
  46. Ruttenberg J, Gao GR, Stoutchini A, Lichtenstein W (1996) Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. In: Proceedings of the SIGPLAN '96 conference on programming language design and implementation, Philadelphia, pp 1–11
  47. Allan VH, Jones RB, Lee RM, Allan SJ (1995) Software pipelining. *ACM Comput Surv* 27(3):367–432
  48. Su B, Ding S, Xia J (1986) URPR - an extension of urcr for software pipelining. In: Proceedings of the 19th workshop on microprogramming, ACM, New York
  49. Su B, Ding S, Xia J (1984) An improvement of trace scheduling for global microcode compaction. In: Proceedings of the 17th workshop on microprogramming, New Orleans
  50. Ebcioğlu K (1987) A compilation technique for software pipelining of loops with conditional jumps. In: Proceedings of the 20th workshop on microprogramming, Colorado Springs
  51. Aiken A, Nicolau A (1988) Optimal loop parallelization. In: Proceedings of the SIGPLAN '88 conference on programming language design and implementation, Atlanta
  52. Aiken A, Nicolau A (1987) Perfect pipelining: a new loop parallelization technique. Technical Report 87-873, Dept. of Computer Science, Cornell University
  53. Callahan D, Cocke J, Kennedy K (1988) Estimating interlock and improving balance for pipelined machines. *J Parallel Distr Com* 5(4):334–358
  54. Aiken AS (1988) Compaction-based parallelization. PhD thesis, Dept. of Computer Science, Cornell University
  55. Cytron R (1984) Compile-time Scheduling and Optimization for Asynchronous Machines. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign

## Molecular Evolution

### ► Phylogenetics

M

## Monitors

- Monitors, Axiomatic Verification of
- Synchronization
- Transactional Memory

## Monitors, Axiomatic Verification of

CHRISTIAN LENGAUER  
University of Passau, Passau, Germany

### Definition

A monitor is a shared abstract data structure that can be used by several processes in parallel. To keep its data in a consistent state, calls to its operations (or *methods*) are executed under mutual exclusion, and there is the possibility of suspending and resuming the caller, depending on the state of the monitor data.

## Discussion

### Overview

For a brief introduction to axiomatic verification with Hoare logic, see the related entry, ►Owicki-Gries Method of Axiomatic Verification. Here, the first issue is the verification of sequentially accessed abstract data structures (frequently called *classes*), and the second the measures that have to be taken to synchronize concurrent calls to monitor operations and make them mutually exclusive.

### Hoare Rules for Classes

The Hoare rules for classes are simple. The main requirement is that the data satisfy an invariant  $I$ .  $I$  must hold after initialization of the data (e.g., in Java, after each constructor call), possibly, under some assumptions; and it must hold at the end of each class method  $op$  provided it holds at its start:

Constructor methods:  $\{pre\} op \{I\}$

Other methods:  $\{pre(op) \wedge I\} op \{post(op) \wedge I\}$

$pre(op)$  and  $post(op)$  are the pre- and postcondition for method  $op$ .

### Hoare Rules for Monitors

To protect concurrent calls properly, two levels of synchronization are added.

### Short-Term Scheduling

Short-term scheduling ensures the mutual exclusion of the execution of concurrent method calls. It is a problem-independent requirement and can be installed by the compiler, that is, the programmer need not worry about it. This is so, for example, in Concurrent Pascal and Modula-2 but not in Java, where the programmer has to add the modifier **synchronized**.

No additional Hoare axioms are necessary. Under the assumption of mutual exclusion, it can be assumed that the monitor data are not being corrupted in parallel while a monitor method is being executed.

### Mid-Term Scheduling

Mid-term scheduling ensures that the monitor data are in the state expected and required by the method being executed. For example, when removing an item from a shared buffer, the buffer must not be empty. If it is empty, the method must be suspended and resumed after an item has been inserted.

Mid-term scheduling is a problem-dependent issue, that is, the programmer must be able to specify it. The usual device is a built-in abstract data structure that holds a FIFO queue (named *cond*) of identifiers of processes that have entered the monitor and have been suspended. The queue is initially empty and comes with two methods:

*wait* suspends the process which calls it, and appends it to the queue; the next process waiting for monitor entry, if any, is granted access

*signal* removes a process from the queue and resumes it after suspending the calling process

There are several signaling strategies which differ in how the process that calls *signal* is treated and when the resumed process is given access to the monitor; these are sketched later in this entry.

Hoare axioms are needed for calls to *wait* and *signal*. Let  $B(cond)$  be the condition to be enforced. The rules proposed originally by Hoare [9] are:

Suspension:  $\{I\} cond.wait \{I \wedge B(cond)\}$

Resumption:  $\{I \wedge B(cond)\} cond.signal \{I\}$

That is, one can presume the validity of condition  $B(cond)$  at the point of resumption if one proves it to hold at the point of suspension. The invariant must be part of both pre- and postconditions because the monitor changes hands.

The waiting condition can be tuned a little bit to avoid unnecessary waiting [10]. Let  $E$  be such that  $I \wedge E \Rightarrow \neg B(cond)$ :

Suspension:  $\{I \wedge E\} cond.wait \{I \wedge B(cond)\}$

Resumption:  $\{I \wedge B(cond)\} cond.signal \{I \wedge E\}$

### Auxiliary and Private Variables

The verification of monitors is akin to the restricted setting of Owicky and Gries (►Owicki-Gries Method of Axiomatic Verification): the conditional critical region of Owicky and Gries [16] corresponds to the body of a monitor operation. That is, auxiliary variables are necessary for the relative completeness of the proof method.

For monitors and other shared abstract data structures, there exists a special kind of auxiliary variable, called a *private variable*. Of a private variable, one instance exists for each process that accesses the monitor [13] and, in a proof, it is indexed by the identifier of the process (see the example below).

The auxiliary variable axiom of Owicky and Gries ensures that auxiliary variables need not be implemented.

### Example

Consider the implementation of an integer semaphore as a monitor, using a syntax close to Concurrent Pascal [2]. Say the semaphore is employed to govern the use of  $m$  instances of a resource by  $n$  parallel processes (assuming  $n \geq m$ ) in an infinite loop. The claim to be proved is that the resources are managed correctly and efficiently [13].

In Concurrent Pascal, a monitor is defined by giving it a name and stating its parameters, then declaring the monitor data and operations, and finally providing an initializing statement.

*Semaphore implementation:*

```
type semaphore = monitor (m : integer);

var s : integer,
 q : condition;

proc P;
begin
 if s = 0 then q.wait fi;
 s := s-1
end;

proc V;
begin
 s := s+1;
 q.signal
end;

begin s := m end
```

*Main program:*

```
resource r(sem : semaphore(m)) ::

cobegin S1 // ... // Sn coend

Si :: while true do
 noncritical section;
 sem.P;
 critical section;
 sem.V;
 noncritical section
od
```

Note that the critical section is not mutually exclusive but can be entered in parallel – by up to  $m$  processes.

### Auxiliary variables:

Inside the monitor, a two-valued private variable is declared:

```
var INC : 0..1
```

which is indexed implicitly with the identifier *caller* of the process currently accessing the monitor. Thus, in the proof, *INC* is an array of  $n$  elements; each element is private to one of the processes accessing the monitor. In the initialization, all elements of the array are set to 0.

### Monitor invariant:

The invariant states that the number of current decrements of the semaphore corresponds to the number of elements of array *INC* currently set to 1:

$$I : 0 \leq s = m - \sum_{caller=1}^n INC[caller]$$

### Proof outline for the main program:

The main program iterates indefinitely, which is specified by the false postcondition. The program manipulates array *INC* such that the number of elements currently set to 1 corresponds to the number of processes currently executing the critical section:

```
{m ≥ 0}
resource r(sem : semaphore(m)) ::

cobegin S1 // ... // Sn coend
{false}

Si :: {I ∧ INC[i] = 0}
while true do
 {I ∧ INC[i] = 0}
 noncritical section;
 {I ∧ INC[i] = 0}
 sem.P;
 {I ∧ INC[i] = 1}
 critical section;
 {I ∧ INC[i] = 1}
 sem.V;
 {I ∧ INC[i] = 0}
 noncritical section
 {I ∧ INC[i] = 0}
od
{false}
```

The proofs for *P* and *V* have to certify their respective manipulations of *INC* as well as the establishment of the invariant, provided it holds before their calls.

### Proof outline for *P*:

```
proc P;
```

```

begin
 $\{I \wedge INC[caller] = 0\}$
 if $s = 0$ then
 $\{I \wedge INC[caller] = 0 \wedge s = 0\}$
 $q.wait$
 $\{I \wedge INC[caller] = 0 \wedge s \neq 0\}$
 fi;
 $\{I \wedge INC[caller] = 0 \wedge s > 0\}$
 $s := s - 1;$
 $\{INC[caller] = 0 \wedge s \geq 0\}$
 $INC := 1$
 $\{I \wedge INC[caller] = 1\}$
end

```

*Proof outline for V:*

```

proc V ;
begin
 $\{I \wedge INC[caller] = 1\}$
 $s := s + 1;$
 $\{INC[caller] = 1 \wedge s > 0\}$
 $INC := 0;$
 $\{I \wedge INC[caller] = 0 \wedge s \neq 0\}$
 $q.signal$
 $\{I \wedge INC[caller] = 0\}$
end

```

*Proof outline for the monitor initialization:*

```

 $\{m \geq 0\}$
 $s := m;$
for $i := 1$ to n do $INC[i] := 0$ od
 $\{I \wedge s = m \wedge (\bigwedge_{i=1}^n INC[i] = 0)\}$

```

*Claim:*

A process is suspended if and only if  $m$  other processes are executing the critical section.

*Proof sketch:*

Note that the invariant holds at all points at which the monitor can change hands, that is, where processes may enter or leave the critical section or are suspended or resumed. The only points at which the invariant does not hold is in the postconditions of the semaphore decrement (in  $P$ ) and increment (in  $V$ ) but, at both these points, it is reestablished immediately under the mutual exclusion provided by the monitor's short-term scheduling.

In the following arguments, note that  $B(q) = s > 0$ :

$\Rightarrow$ : That processes are being suspended when  $m$  processes are executing the critical section follows

directly from the invariant, which guarantees that, in this case,  $s = 0$ . Note that  $s = 0 \Rightarrow \neg B(q)$ .

$\Leftarrow$ : Processes are suspended only when necessary, since  $pre(q.wait) \Rightarrow \neg B(q)$ .

### Other Signaling Strategies

The problem of signaling received further attention almost immediately. The question is when exactly the signaler should leave the monitor and when the signee should be resumed. The different signaling strategies make use of, in all, three waiting queues:

The *condition queue*: the monitor's FIFO queue for mid-term scheduling.

The *entry queue*: the monitor's FIFO queue for short-term scheduling.

The *urgent queue*: an additional FIFO queue for short-term scheduling that takes priority over the entry queue (only used in strategy SU).

Five signaling strategies have been proposed [1]:

signal + continue (SC): Java

Signaler continues.

Signalee moves from the condition queue to the entry queue.

signal + exit (SX): Concurrent Pascal

Signaler leaves the monitor for good.

Signalee is removed from the condition queue and resumed.

signal + wait (SW): Howard [10], Modula, Euclid

Signaler is added to the entry queue.

Signalee is removed from the condition queue and resumed.

signal + urgent wait (SU): Hoare [9], Pascal-Plus

Signaler is added to the urgent queue.

Signalee is removed from the condition queue and resumed.

automatic (AS): Owicky-Gries general setting [15] (the **await** statement).

Implicit SC.

Hoare axioms, which are by nature more complicated than the ones mentioned previously in this entry, were proposed early on [11].

A drawback of the SC strategy is that a process, upon resumption, may encounter the monitor data not in the expected state – some other process may have interfered! Andrews [1] states that the advantages of the SC

strategy are that it is simpler – it suffices to provide only a single condition queue per monitor object, not one queue per condition to be waited on – and that the proofs of *wait* and *signal* are less context-dependent and thus easier to do in isolation. He also conjectures that the SX strategy fosters operation fragmentation.

## Related Entries

- ▶ [Verification of Parallel Shared-Memory Programs](#), [Owicki-Gries Method of Axiomatic](#)
- ▶ [Synchronization](#)

## Bibliographic Notes and Further Reading

The verification rules for abstract data structures were first proposed by Tony Hoare [8] soon after his seminal paper on axiomatic program verification [7].

The monitor concept was invented in the early 1970s by Tony Hoare [9] and Per Brinch-Hansen [2], and it was quickly included in several programming languages of the time. Some problems with monitors were quickly discovered and discussed [12]. One consequence is to avoid call hierarchies, that is, calling one monitor operation while executing an operation of another monitor [14].

Over the years, the monitor concept has been reviewed repeatedly and its variants have been classified [1, 3]. An extended monitor concept for Java with the option of several signaling schemes and with prioritized resumption has been proposed [4].

Ole-Johan Dahl reviewed the axiomatic verification of monitors at the honorary symposium for Tony Hoare in 1999 on the occasion of his retirement from Oxford University [6] and, 5 years earlier, at a similar function for Tony Hoare's 60th birthday [5]. Dahl seemed unaware of Howard's early work [11]. In 1999, when electricity failed in the middle of the talk due to the flooding of a transformer in bad weather, Dahl completed his presentation with clarity and composure without visual aids in the windowless room, which was illuminated only by the exit signs – and took questions afterward!

## Bibliography

1. Andrews GR (1991) Concurrent programming – principles and practice. Benjamin/Cummings, Redwood City
2. Brinch-Hansen P (1975) The programming language Concurrent Pascal. IEEE Trans Softw Eng SE-1(2):199–207

3. Buhr PA, Fortier M, Coffin MH (1995) Monitor classification. ACM Computing Surveys 27(1):63–107
4. Chiao H-T, Wu C-H, Yuan S-M (2000) A more expressive monitor for concurrent Java programming. In: Bode A, Ludwig T, Karl W, Wismüller R (eds) Euro-Par 2000: parallel processing, Lecture notes in computer science 1900. Springer-Verlag, Heidelberg, pp 1053–1060
5. Dahl O-J (1994) Monitors revisited. In: Roscoe AW (ed) A classical mind, Series in computer science, Chap 6. Prentice Hall International, Hemel Hempstead, pp 93–103
6. Dahl O-J (2000) A note on monitor versions. In: Davies J, Roscoe AW, Woodcock J (eds) Millennial perspectives in computer science, Cornerstones of computing. Palgrave, Basingstoke, pp 91–97
7. Hoare CAR (1969) An axiomatic basis for computer programming. Commun ACM 12(10):576–580, 583
8. Hoare CAR (1972) Proof of correctness of data representations. Acta Inform 1(4):271–281
9. Hoare CAR (1974) Monitors: An operating systems structuring concept. Commun ACM 17(10):549–557. Corrigendum: 18(2):95
10. Howard JH (1976) Proving monitors. Commun ACM 19:273–279
11. Howard JH (1976) Signaling in monitors. In: Proceedings of the 2nd international conference on software engineering (ICSE'76), pp 47–52, San Francisco, 1976
12. Keedy JL (1979) On structuring operating systems with monitors. ACM SIGOPS Oper Syst Rev 13(1):5–9
13. Lengauer C (1978) On the axiomatic verification of concurrent algorithms. Master's thesis, Department of Computer Sciences, University of Toronto, August 1978. Technical Report CSRG-94
14. Lister AM (1977) The problem of nested monitor calls. ACM SIGOPS Oper Syst Rev 11(3):5–7
15. Owicki SS, Gries D (1976) An axiomatic proof technique for parallel programs. Acta Inform 6(4):319–340
16. Owicki SS, Gries D (1976) Verifying properties of parallel programs. Commun ACM 19(5):279–285

## Moore's Law

JOHN L. GUSTAFSON  
Intel Corporation, Santa Clara, CA, USA

## Synonyms

There are no exact synonyms for Moore's law with respect to chip technology. Kryder's law for hard disk storage, Butter's law for fiber optic transmission speeds, and Nielsen's law for consumer home network speeds are examples of very similar formulations for other technologies. These may be regarded as derivatives of Moore's law, and they are often described as "A Moore's law for mass storage" etc. Technologists often

cite Moore's law very broadly to refer to any technology that changes as an exponential function of time.

## Definition

Moore's law states that the number of transistors on a chip doubles every 24 months.

More precisely, the law is an empirical observation that the density of semiconductor integrated circuits one can *most economically manufacture* doubles about every 2 years. Thus, the cost of manufacturing a transistor drops by half about every 2 years. Some definitions use 18 months as the doubling period instead of 2 years; see the Discussion section below for clarification.

Caltech professor and chip pioneer Carver Mead coined the term "Moore's law" in 1970.

## Discussion

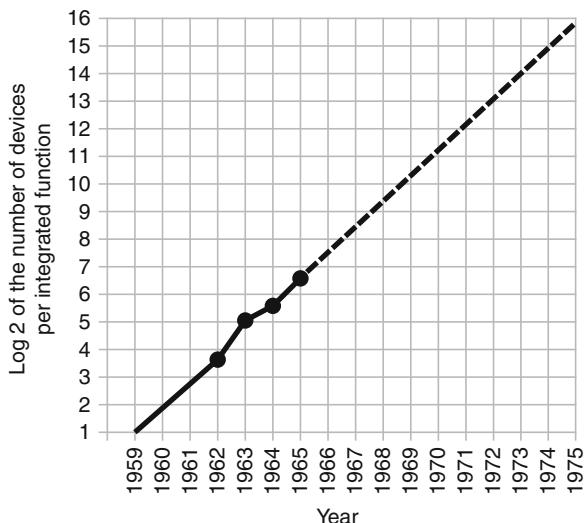
### History

The idea of putting complete circuits on a monolithic block of semiconductor material originated with Jack Kilby (Texas Instruments, 1958) and, independently, Robert Noyce (Fairchild Semiconductor, 1959). Gordon E. Moore's 1965 paper on the progress in integrated electronics [1] noted the rapid rate of density improvements in just those 6 years. He made the bold prediction that "with unit cost falling as the number of components per circuit rises, by 1975 economics may dictate squeezing as many as 65,000 components on a single silicon chip."

Figure 1 shows Moore's original graph [1].

Only four data points were used to create the original graph, and Moore's original estimate was twice as optimistic as the rate of improvement given in later revisions: "The complexity for minimum component costs has increased at a rate of roughly a factor of two per year." Notice that these are the transistor counts for individual functions, not for processors, since the first complete microprocessors on a chip would not be possible until several years later.

At the time, the driving motivation for integrated circuitry was to satisfy the needs of the military and the US space program, both of which had a high tolerance for the cost of parts needed to do a particular job. Moore observed, even in 1965, that the decrease in cost would eventually usher in an era of personal computing and "personal portable communications equipment," that is,



**Moore's Law. Fig. 1** Moore's original graph estimating the trend of device density

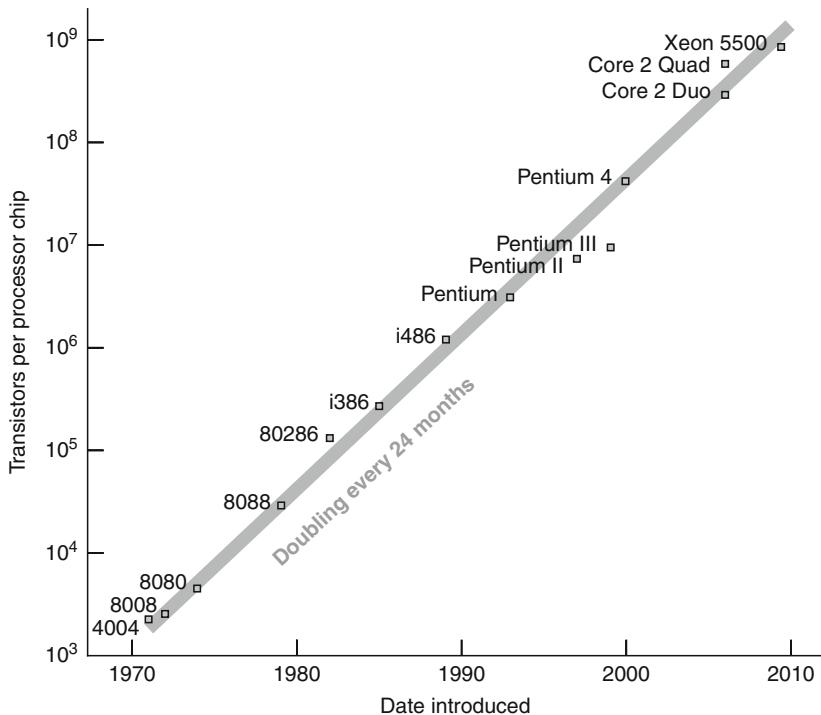
cellular phones. A cartoon accompanying Moore's article showed a set of store kiosks for cosmetics, notions, and "handy home computers," intended as a joke to overstate Moore's prediction. The humor and intended exaggeration of the cartoon is almost completely lost on the modern reader, since it depicts a scene not unlike that found in current shopping malls.

Around 1970, Caltech Professor Carver Mead, a pioneer of very large-scale integration (VLSI) technology, coined the term "Moore's law" for the density doubling rule of thumb. Moore updated his paper in 1975 [2]. That paper used a more complete set of data to revise the rate of improvement to a density doubling every 24 months.

The more recent graph in Fig. 2 begins where Moore's original graph left off, and encompasses four decades of Intel family processors. It makes clear that for processor chips, a doubling every 2 years is an excellent match to the empirical data.

### A Self-fulfilling Prophecy

What started as an empirical observation about the integrated circuit business became a *self-fulfilling prophecy* that pervades computing. Moore himself was perhaps the first to apply that phrase to his law, in a 2005 interview. The law is as much an economic guideline as a technical one [3]. Moore's paper began as a forecast but became an industry guideline. As a well-known trend



Moore's Law. Fig. 2 Moore's law for CPU device density

M

believed by an entire industry, there are risks to introducing products that are either above or below the trend line, since the entire supply chain from the suppliers to chip fabrication facilities to the consumers of the electronic systems built from those chips is calibrated to the expectations of the forecast.

If a commercial entity aims below Moore's trend line, the result will be a product that has insufficient performance to compete in the marketplace. If it aims above Moore's trend line, it will find the infrastructure unavailable to produce the parts except at extremely high costs, and they will not be in balance with the performance of chips available from other vendors. Thus, the entire semiconductor industry marches to the same beat as the safest business practice.

### Moore's Law and Software

Niklaus Wirth observed that software demands for hardware speed and memory capacity grow faster than Moore's law, resulting in a net decrease in system performance [4]. He quotes M. Reiser's remark "Software is getting slower more rapidly than hardware becomes faster," and a variant on C. Northcote Parkinson's

cynical observation (Parkinson's law) that work expands to fill the time available for its completion:

"Software expands to fill the available memory."

Bill Gates made a similar observation, but said that the net result is to keep system speed about the same, not to get worse. Gates speculates that this is for more than one reason: programmers add more and more features but also code less and less efficiently as hardware grows in capability. Worrying less about the efficiency of software makes the software less costly to produce. This is an example of how Moore's law can enable trade-offs that save costs in areas of computing other than chip technology. Many applications and operating systems now contain tens of millions of lines of software as a side effect of Moore's law.

The cost of software was trivial relative to the cost of hardware in the early days of electronic computing. The operation cost of a 1960s mainframe computer was over a hundred times as much as the cost of a programmer on a per-hour basis. Thus, it was common for programmers to program at the lowest level of machine language to achieve sufficiently high performance. Moore's law

has dropped the price of hardware so dramatically that when one seeks a new application, programming the application is often the most expensive aspect of using the computing system. Therefore, there is considerable incentive to use the cost savings of Moore's law improvements to reduce programming cost by putting more of the performance burden on the hardware and less on the programmer.

### Misconceptions About Moore's Law

Perhaps the most common misconception about Moore's law is that it predicts *performance* will double periodically, as measured by clock speed or any other performance metric. This is partially supported by device-level physics, since decreasing the size of a transistor allows it to switch faster. However, because not all performance features scale with the transistor performance (such as the speed of signals on the chip), performance improvements generally require redesign as well as the shrinking of features. Moore did not make any predictions about performance nor did he ever cite a doubling interval of 18 months. The "18 month" period of doubling is the result of an attempt by former Intel executive David House to estimate and publicize a growth rate for the performance improvement in microprocessors. While transistor counts are easy to quantify, it is much more difficult to find a rigorous definition of absolute computer performance.

### Implications for Parallel Computing

In recent years, Moore's law has become a major reason that interest in parallel computing methods has become prominent and intense. The most recent and obvious consequence of the increased density of VLSI is that a single processing chip usually contains more than one processor. This has brought the issues of parallel computing to developers of even very small and inexpensive devices.

A more profound effect of Moore's law that drives parallel computing has been in operation for decades: *not all technology features and requirements scale at the same rate as Moore's law*, forcing designers to make architectural changes. For example, whereas processor capability and memory capacity have improved about a million-fold with Moore's law in the last 35 years, the time latency between the processor and the (off-chip) memory has improved only very slightly over

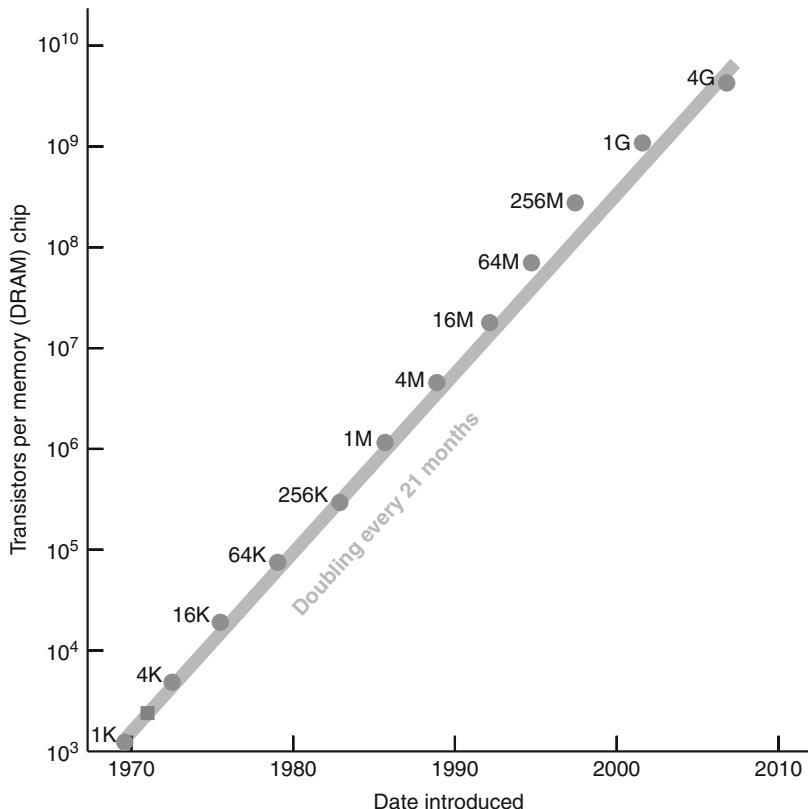
that same period. The simple "von Neumann architecture," in which a single monolithic memory supplies data and instructions to a single monolithic processor, is now impractical because of this disparate scaling. Yet, programmers have created a vast body of serial software to fit that increasingly untenable model. Modifications to the von Neumann architecture (such as automatic caching of memory or the automatic overlapping of operations and memory transfers) have helped to convert Moore's law device density improvements into higher performance without changes to that traditional serial software, but only up to a point. Beyond that point, the programmer faces the need for parallel computing techniques to exploit the higher device density of VLSI for continued improvements in performance.

A simple way to understand this is to consider cost. Imagine that to double performance of a computer system while maintaining the illusion of a single memory space and a single thread of execution will increase the cost by a factor of four. The alternative is to buy two computers without changing the design, which nominally doubles the cost of the processing hardware. Added to this is the cost of connecting them and the cost of reprogramming so they can operate in parallel. Depending on the application, the connection and reprogramming costs might be quite a bit less than another factor of two, so the parallel approach has lower total cost.

Moore's law predicts a steady state in the drop of the cost of a megabyte of memory. The data for the first introduction of each new generation of dynamic random-access memory (DRAM) chip actually shows a doubling approximately every 21 months, not every 24, as shown in Fig. 3.

While memory parts became cheaper for computer makers, the price per megabyte of memory charged by vendors diverged sharply in the 1990s between makers of distributed memory parallel computers and those making monolithic designs [5], as shown in the Table 1 below, compiled in 1992.

The first four entries reflect the commodity pricing of DRAM for personal computers and workstations. The last three entries reflect the price of proprietary memory designs from vector supercomputer makers of the same period, and are higher by almost two orders of magnitude. The nCUBE 2 and Intel Paragon, being distributed memory parallel systems, were able to



Moore's Law. Fig. 3 Device density for dynamic-access memory

Moore's Law. Table 1 Price for distributed memory versus monolithic memory, ca. 1992

| Computer           | List price to add 1 megabyte of memory |
|--------------------|----------------------------------------|
| Apple Mac IIxi     | \$124                                  |
| 386 PC             | \$112                                  |
| Sun SPARCstation 2 | \$81                                   |
| SGI Indigo         | \$172                                  |
| nCUBE 2            | \$350                                  |
| Intel Paragon      | \$344                                  |
| NEC SX-3           | \$2,740                                |
| Hitachi EX Series  | \$4,000                                |
| CRAY Y-MP          | \$7,800                                |

price memory much closer to what was being offered for individual workstations since they did not attempt to preserve the programming view of a serial (von Neumann) architecture. By the late 1990s, "Beowulf"

clusters for supercomputing were built, literally out of consumer-grade personal computers and so had the same cost per megabyte of DRAM. This illustrates how Moore's law creates upheavals in prevailing computer architectures, and in particular led to the now-ubiquitous use of distributed memory and parallel processing to create the fastest computing systems.

### The ASCI Program and Other "Above Trend" Efforts

The Accelerated Strategic Computing Initiative was a late 1990s US Department of Energy program intended to replace nuclear weapon testing with computer simulation. In recognizing the vast improvement in computing power required to achieve this, the proponents included "accelerated" in the name to indicate the desire to *exceed Moore's law* [6]. While hundreds of millions of dollars were spent on the very large parallel supercomputing systems developed for the program and placed

at the national weapons laboratories, this was still such a small fraction of the overall computing market that there was no impact to accelerate Moore's law. The density and price of the systems did not deviate from the predicted trend. The systems were somewhat more powerful than the trend of supercomputer speed versus year, simply because they were more expensive per system on an inflation-adjusted basis.

The 2002 Japanese Earth Simulator was another striking national effort at stretching beyond the performance trend that succeeded through more massive expenditure. The cost for the Earth Simulator was an estimated \$350 million [7]. Since parallel supercomputer developers do not have sufficient influence in the total computing market to bend Moore's law, they simply spend more money to build larger facilities. The largest computing facilities such as those built by Google and Microsoft contain several acres of computing equipment with attendant power, cooling, and interconnect.

## The Future of Moore's Law

### Early Pessimism

In 1973, Carver Mead stated in a classroom lecture at the California Institute of Technology that Moore's law would come to a physics-based "day of reckoning" around 1984, when transistors would be so small that a single electron would determine its on-off state. Part of the inaccuracy of his prediction may have been the result of using a 1-year period of doubling instead of the 2-year period Moore cited in his revision in 1975. Another error may have been to assume that transistor geometries would scale down uniformly in all three dimensions; current devices are tall and thin on the chip relative to their horizontal dimensions, like a very high hedge maze, which has delayed some of the impending limits of scaling semiconductor physics.

### Optical Limits

Similar pessimistic predictions have appeared, often based more on engineering limitations than on fundamental physical laws. For example, the primary manufacturing method has been photolithography, in which light passing through a mask creates chemical changes in a polymer (photoresist) causing it to harden in precise patterns. Optical theory says one cannot create images

more accurate than about one wavelength of light, because of diffraction effects. This predicts that Moore's law must end when device features became smaller than a single wavelength of visible light. That limitation would have meant that device features could not be smaller than about 0.5 μm. Yet, mass-manufactured chips, at the time of this writing, have features as small as 0.045 μm. Engineers overcame the barrier using much smaller wavelength light in the ultraviolet, in combination with sophisticated mathematical corrections to the diffraction effects so that features smaller than a single wavelength could image reliably. What appeared to be an absolute physical limit was actually simply an engineering problem to overcome.

### Heat and Power Limits

At present, heat and power dissipation are engineering challenges for the future of Moore's law. Chips that use more than about 140 W are difficult to cool economically, so designers are using changes to device parameters as well as changes to the architecture. Lowering the operating voltage is one technique, but equally helpful is to *reduce the clock speed*. While reducing the clock speed slows serial processing, it can allow designers to pack more processors onto a single chip because multiple processors still fit into the power budget. This is one reason the chip industry embraced "multicore" parallel architectures. Efforts to raise clock speed were at the point of diminishing returns, but the use of parallel computing allowed the performance per chip to stay on the Moore's law trend line.

### Design Effort Limits

Chip designers laid out the earliest chips manually. They made masks with black tape, by hand, hence the term "taping out" a chip. As the number of devices grew to the hundreds, computers and automated equipment displaced that part of the design effort. However, even as the ability of computer software tracked the need to manage millions and then billions of individual devices, some speculated that the process would become prohibitively arduous and hence human-limited. While some devices (such as memory chips) involve many repetitions of a single small design "cell," the more complex chips (such as processors used in servers) involve the custom design and management of many different cell types. Placing the devices and routing the connections

between them in an optimal way is an NP-hard problem that taxes both human designers and the software tools. The production of a new state-of-the-art processor involves thousands of engineers and about 200 different software applications for various stages of the design and manufacture. While this level of investment has tended to reduce the number of industrial companies that are able to create competitive semiconductor devices, it has not resulted in a slowing of Moore's law.

### Reliability at the Atomic Scale

If the feature sizes of chips continue to shrink as predicted by Moore's law, chips will soon contain features that are only a few atoms across. This limit occurs around 2015–2020. At such sizes, quantum effects and temperature noise will force us to fundamentally rethink the way circuits accomplish computation. Note that an individual atom has multiple quantum states that can store information, so Moore's law does not necessarily end when device features are at the individual atom level of detail.

### Lack of Market Need

Some have even speculated that Moore's law will end not for technological reasons but because all of our electronics will be "good enough," and thus manufacturers will not seek further improvements to size and cost. While this argument might hold for any given device type, like a four-function calculator, many areas of human endeavor (such as high-performance computing to simulate physical behavior) have seemingly boundless appetites for improvements and have no ceiling on what they demand for speed and storage.

### Manufacturing Facility Cost

Another economic argument against continued progress is the exponentially increasing cost of the manufacturing facilities for semiconductor chips. According to Dan Hutcheson, CEO of VLSI Research, the cost of building a fabrication facility doubles every 4 years. This forecast is one he called "Rock's law," after early Intel venture capitalist Arthur Rock [8]. While larger facilities create economies of scale, Rock's law seemingly collides with the cost-reduction part of Moore's law. However, the cost *per transistor* may continue to drop even as the plants that produce them become increasingly expensive to build. A closely related metric is the cost *per unit*

area of building a chip. Like the facility cost, this metric is increasing, even though the cost per transistor is decreasing.

### Perspective

The rate of technology advance in chip technology is a rate of improvement without precedent in human history. Some have observed that if automobile technology followed Moore's law, a car would now cost less than a penny, would use drops of gasoline instead of gallons, and would safely go hundreds of thousands of miles per hour. The rate of change has been so staggering, and so different from that of other industries, that many computing businesses fail if their management tries to apply the kind of product marketing used for non-technology companies. A recurring situation is that the parts cost to a computer maker drops by a factor of two, and the company's executives face the choice of passing this savings to their customers (which they fear will cause their revenue to drop with Moore's law) or maintaining their prices with the aim of increasing their profits. The consequence of the latter choice is often that another company with little to lose and everything to gain by introducing a competitive product undercuts them by passing on the savings of Moore's law to customers. Even in an economy that has moderate inflation, consumers of technology have come to expect inexorable improvement in the price and capabilities of any electronic device.

For many years, Moore's law served as a *damper* on parallel computing efforts. Why rewrite software when in less than 2 years you can simply buy a serial computer that is twice as fast? Now, in contrast, the shrinking of transistor size and cost means the issues of parallel computing are everywhere.

M

### Related Entries

- [Exascale Computing](#)
- [Fault Tolerance](#)
- [Intel Core Microarchitecture, x86 Processor Family](#)
- [Memory Wall](#)
- [Metrics](#)
- [Power Wall](#)
- [SoC \(System on Chip\)](#)
- [VLSI Computation](#)

## Bibliographic Entries and Further Reading

Ray Kurzweil has extended Moore's law to many other areas of technological change that seem to be advancing exponentially. *The Singularity is Near: When Humans Transcend Biology* (Viking 2005; Penguin 2006) [9]. In his "The Law of Accelerating Returns," <http://www.kurzweilai.net/articles/art0134.html?printable=1>, Kurzweil points out that each apparent limit to the feasibility of progress for some technology is overcome by innovations that jump to a slightly different technology that avoids the limit, and Moore's law is an example of this phenomenon.

Niklaus Wirth lectured and wrote often about how software "bloat" seemed to negate and even overwhelm the advances provided by Moore's law. A representative example is "A Plea for Lean Software" [4].

The September 2006 issue of *IEEE Solid State Circuits* is a special issue on Moore's law. It includes the article by David E. Liddle referenced below that points out many of the subtle and economic implications of Moore's law. See [http://www.ieee.org/portal/site/sscs/menuitem.656c9fab99af5fb8699305765bac26c8/index.jsp?&pName=sscs\\_enews\\_home&month=09&year=2006](http://www.ieee.org/portal/site/sscs/menuitem.656c9fab99af5fb8699305765bac26c8/index.jsp?&pName=sscs_enews_home&month=09&year=2006).

One of the best in-depth articles on the chip manufacturing technology underlying Moore's law is by Jon Stokes, in Ars Technica (last updated September 2008). <http://arstechnica.com/hardware/news/2008/09/moore.ars>.

## Bibliography

1. Moore GE (1965) Cramming more components onto integrated circuits. Electronics 38(6). [ftp://download.intel.com/museum/Moores\\_Law/Articles-Press\\_Releases/Gordon\\_Moore\\_1965\\_Article.pdf](ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf)
2. Moore GE (1975) Progress in digital integrated electronics. In: Proceedings of the IEEE electron devices meeting, vol 21. San Francisco, CA, pp 21–25
3. Liddle DE (2006) The wider impact of Moore's law. IEEE Solid State Circuits
4. Wirth N (1995) A plea for lean software. Computer 28(2):64–68. <http://cr.yp.to/bib/1995/wirth.pdf>
5. Gustafson J (1992) The vector gravy train. Supercomputing Review. <http://www.scl.ameslab.gov/Publications/Gus/Gravy.html>
6. Gustafson J (1998) Computational verifiability and feasibility of the ASCI program. IEEE Computing 5(1). doi:<http://doi.ieeecomputersociety.org/10.1109/99.660304>
7. Lohr S (2004) Technology; IBM decides to market a blue streak of a computer. The New York Times, June 21
8. Cnet news (2009) [http://news.cnet.com/Semi-survival/2009-1001\\_3-981418.html](http://news.cnet.com/Semi-survival/2009-1001_3-981418.html)
9. Kurzweil R (2005) The singularity is near: when humans transcend biology. Viking Press, New York

## MPI (Message Passing Interface)

WILLIAM GROPP

University of Illinois at Urbana-Champaign, Urbana, IL, USA

### Synonyms

Message passing

### Definition

MPI is an ad hoc standard for writing parallel programs that defines an application programmer interface (API) implementing the message-passing programming model. MPI is very successful and is the dominant programming model for highly scalable programs in computational science. The fastest parallel computers in the world, with more than 200,000 cores, run programs written in MPI, and the fastest applications (achieving over 1 PetaFLOPs in 2010) use MPI.

MPI was created in two stages. The original MPI standard [7], developed between 1992 and 1994, specified a library interface for point-to-point and collective communication, along with various support routines and other features. The second stage, from 1995 to 1997, extended MPI to include parallel I/O, remote memory access, and creation of additional processes, among other features.

## Discussion

### MPI-1

The MPI standard provides a way to make use of the Communicating Sequential Processes (CSP) programming model. In MPI, conventional processes make explicit calls to library routines defined by the MPI standard to communicate data between two or more processes.

In the message-passing model, processes communicate by sending messages. This is a two-sided operation: the sending process describes the data to be sent

and the receiving process describes how to receive the message. More specifically, the sender and the receiver each must describe the following: What data is sent? To whom is it sent? Where (in memory) is it received at the destination process? In addition, the receiver may need to know: who sent the data and how much data was sent. In addition, it is common to provide one additional nonnegative integer of information to be communicated from the sender to the receiver. This is called the *message tag*. It can be thought of as a way to identify a specific kind or type of message to the receiving process, and the receiving process can specify that a particular receive operation will only accept a message with a specific tag (in earlier message-passing systems, this tag was sometimes called the *message type*).

MPI generalizes the notion of what data is send and to whom the data is sent. The most common specification for data is a pointer to data and a length (e.g., in bytes). For example, the Unix write routine describes the data to write this way. In MPI, this is generalized in two ways. First, the type of data is described in terms of an *MPI\_Datatype*. The basic MPI Datatypes match the basic datatypes in the programming language being used with MPI. Thus for an “int” in C, there is an MPI\_INT datatype; for an “INTEGER” in Fortran, there is an MPI\_INTEGER datatype. This allows users to describe the amount of data to be sent in natural terms (10 ints) rather than as a specific number of bytes, which may depend on the particular platform (e.g., is an int 4 or 8 bytes). In addition, this allows an MPI implementation to convert the data being sent to match the receiving process. For example, if the sender uses 4 byte ints in big-endian format and the receiver used 8 byte ints in little-endian format, the MPI implementation can ensure that the data is correctly received. Thus, to specify the data that is to be sent (or the location into which data is to be received), MPI specifies a 3-tuple: pointer, count, and MPI Datatype.

The MPI Datatype parameter allows an additional generalization. In MPI, there are routines to define new, user-defined datatypes in terms of the any defined (basic predefined or user-defined) datatype. These datatypes may describe multiple data items and need not be contiguous. For example, MPI provides a routine to define a “vector” datatype that describes blocks of a datatype, separated by a stride in memory. By using such a datatype (or one of the other MPI datatype

constructors) in the 3-tuple describing the data to send (or to receive), the programmer can describe an arbitrary set of data to send or receive.

MPI also generalizes how the source and destination processes are described. Perhaps the simplest way to identify the source and destination is to number all of the processes, starting from zero (this works to identify MPI processes belonging to the same executing MPI program; it does not and is not intended to identify the MPI processes outside of the MPI program as a hostname and process id (pid) pair would on typical cluster). The number between zero and the number of processes minus 1 is called the *rank* of the process. In early message-passing systems, this rank was all that was necessary to identify the source and destination of a process.

In MPI, the rank of a process is relative to a collection of processes. These collections are called *MPI\_Groups*, and the programmer may construct new groups from any existing *MPI\_Group*. When an MPI program starts, there is a group that contains all processes in the MPI program. This group is combined with a *communication context* (more on that below) in an MPI object, called a *Communicator*. In MPI-1, all communication is relative to a communicator. When an MPI program starts, the communicator MPI\_COMM\_WORLD is defined, and each process can determine which rank it is relative to this communicator and how many processes are part of this communicator. Allowing the creation of new communicators containing different numbers of processes makes it much easier to build applications and libraries that can work with subsets of processes. Thus, in MPI, the destination process for a send (or the source process in a receive) is specified with the 2-tuple of rank and communicator.

In addition to sending and receiving data, MPI provides routines to initialize and end an MPI program; these are the routines MPI\_Init and MPI\_Finalize, and routines to determine the number of processes in a communicator (and in the MPI program if the communicator is MPI\_COMM\_WORLD) and the rank of a process in a communicator. These routines are MPI\_Comm\_size and MPI\_Comm\_rank, respectively. With these routines, and with MPI\_Send to send a message and MPI\_Recv to receive a message, we can show a simple MPI program.

The following example of an MPI program illustrates these basic MPI routines. In this complete C program, the process with rank 0 in MPI\_COMM\_WORLD sends two integer values to the process with rank 1, which then prints out the values received.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
 int rank, size, sbuf[2], rbuf[2];
 int dest, source, tag;
 MPI_Status mystat;
 MPI_Init(&argc, &argv);/* Initialize MPI */
 MPI_Comm_rank(MPI_COMM_WORLD,
 &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 if (rank == 0) {
 sbuf[0] = 1; /* Any data process zero wants to
 send */;
 sbuf[1] = size;
 dest = 1; /* destination rank */
 tag = 0; /* message tag */
 MPI_Send(sbuf, 2, MPI_INT, dest, tag,
 MPI_COMM_WORLD);
 }
 else if (rank == 1) {
 source = 0; /* Source rank */
 tag = MPI_ANY_TAG; /* receive the next
 message from source, independent of the tag */
 MPI_Recv(rbuf, 2, MPI_INT, source, tag,
 MPI_COMM_WORLD, &mystat);
 printf("Received %d %d from sender\n", rbuf[0],
 rbuf[1]);
 /* mystat.MPI_SOURCE contains the source
 rank;
 mystat.MPI_TAG contains the tag of the message
 */
 }
 printf("Process %d is exiting\n", rank);
 MPI_Finalize(); /* No MPI calls after MPI_Finalize
 executes */
 return 0;
}
```

Most of these features have already been discussed; the one MPI item that is new here is the *MPI\_Status* object. This is used to contain information about the

source, tag, and size of the message. It is also important to understand how this parallel program executes. When receiving, a specific source and tag may be specified, or MPI\_ANY\_SOURCE or MPI\_ANY\_TAG used, respectively. In that case, the *MPI\_Status* object may be used to determine the source and tag, respectively. These also allow the specification of nondeterministic programs.

In MPI, each process executes with its own address space. There are no shared variables or direct access to memory in another process (in MPI-1). If the above program is started with two processes (which may be on two processors, two cores of the same processor chip, or even two processes executing on the same core), each process has a full copy of all of the variables. For example, in the above example, there is an “int sbuf[2]” in each process. In addition, MPI programs execute independently; it is only when an MPI routine is called that two or more processes may communicate. In particular, language features or routines that are not part of MPI will execute independently in each process. In this example, the printf right before the call to MPI\_Finalize is executed by each process and in whatever order occurs as the program executes.

## Major Features of MPI-1

MPI provides a rich set of routines. They are described in separate chapters in the MPI standard, and include both point-to-point (two party) and collective (many party) communication.

The core of MPI-1 is the point-to-point communication routines. We have already seen the two most basic routines – MPI\_Send and MPI\_Recv. MPI provides a number of variations on each of these. One of the most important is the nonblocking versions of these routines. In this case, the corresponding MPI routine only initiates the communication – a separate MPI routine is required to ensure that the communication has completed. This permits the MPI implementation to overlap the execution of these operations with other computation or communication. Another variation is different send modes. In addition to the basic send mode, there is a *synchronous* send mode, where the send operation completes only when the matching receive begins, and the *ready* send mode, which requires that the matching receive has already been initiated on the destination

process (permitting optimization of the message delivery). MPI also provides for *persistent* communication for operations that are repeated, such as communication within a loop. MPI also provides routines to test for the presence of a message and to cancel nonblocking communication.

MPI Datatypes have already been mentioned; MPI provides routines to create datatypes that represent regularly strided data (vector), blocks of different size (similar to the Unix IOV structure), and (in MPI-2) blocks of the same size. By specifying a data layout with an MPI datatype, rather than by having the user first gather the data into a contiguous buffer and then send it, the MPI library can eliminate a memory copy and, in some cases, exploit special instructions to more efficiently move data.

In addition to point-to-point communication, MPI provides a rich set of collective communication and computation routines. For example, the routine MPI\_Bcast allows one process to broadcast the same data to every other process in an MPI communicator. Other routines provide one-to-many (MPI\_Scatter), many-to-one (MPI\_Gather), and many-to-many (MPI\_Alltoall) communication. These routines are provided because efficient, scalable implementations require great care and depend upon the specifics of the underlying network hardware. Collective computation is also very important, and MPI provides routines to perform parallel prefix (MPI\_Scan), reduction (MPI\_Reduce), reduction to all (MPI\_Allreduce), and a distributed reduction (MPI\_Reduce\_scatter). Like the point-to-point communication routines, data is described by the 3-tuple of pointer, count, and datatype.

An innovative feature of the MPI API is the profiling interface. Each MPI routine is available in two versions: one with the MPI prefix and one with a PMPI prefix. The PMPI version performs exactly the same operations as the MPI version. The user is permitted to override the MPI versions. Thus, to count the number of times that MPI\_Send is used in a program, all an MPI programmer needs to do is to compile the following code and link it with their application:

```
#include "mpi.h"
static int sendCount = 0;
int MPI_Send(void *buf, int count, MPI_Datatype
 dtype, int rank, int tag, MPI_Comm comm)
```

```
{
 sendCount++;
 return PMPI_Send(buf, count, dtype, rank, tag,
 comm);
}
int MPI_Finalize(void)
{
 int rank;
 MPI_Comm_rank(MPI_COMM_WORLD,
 &rank);
 printf("Process %d called MPI_Send %d times\n",
 rank, sendCount);
 return PMPI_Finalize();
}
```

This works on any system and does not rely on operating system-specific features. A number of tools take advantage of this feature to provide performance and correctness debugging information about an MPI program.

As mentioned above, communication in MPI is relative to communicators that contain a group of processes and a communication context. This communication context is not accessible to the user but is a hidden property of the communicator. The reason for this is to support modular programming, where different suppliers provide software libraries that use MPI. These libraries must be certain that their communication cannot be intercepted by other routines in the application. Without private communication contexts, it isn't possible to make this guarantee. Thus, each communicator in MPI contains a group of processes and a private communication context. A new communication context is created every time a new communicator is created. The routine MPI\_Comm\_dup creates a new communicator with the same group of processes but a new communication context; it is recommended that libraries using MPI call this routine to get a private communicator and use that communicator for all communication.

While the communication context is not directly accessible to the user, the group of process is and may be manipulated by the user. However, for many users, the best way to create a new communicator is by “splitting” an existing communicator into new ones with MPI\_Comm\_split.

MPI also provides a set of routines to create and use virtual process topologies; these are arrangements of

processes so that each process has a natural set of neighbors. MPI\_Cart\_create, for example, takes as input an MPI communicator and the description of a Cartesian mesh and returns a new communicator, where the neighbors of each process in the mesh of processes can be determined and where the processes may be mapped onto the physical network in such a way that communication with neighbor processes is especially efficient.

Other features in MPI-1 allow the definition of data to be cached on a communicator and routines to define the behavior when a recoverable error is encountered. The rich set of features of MPI, along with the focus on performance and completeness, helped MPI rapidly become the preferred way to use the message-passing programming model on parallel computers. But within two years of the release of MPI, users were asking for additional features. To respond to that need, the MPI Forum reconvened and began defining a set of extensions to MPI.

## MPI-2

MPI-2 [6] defines a set of *extensions* to MPI-1: any valid MPI-1 program is a valid MPI-2 program. While MPI-2 roughly doubled the number of routines in MPI, most of the additions were in four areas: remote memory access, dynamic processes, parallel I/O, and C ++ and Fortran 90 language bindings.

## Major Features of MPI-2

Remote memory access (RMA) provides a “one-sided” model of communication, where one process specifies both the source and the destination of the communication. Sometimes called “put/get” programming, MPI provides routines to put data into a remote process (MPI\_Put), get data from a remote process (MPI\_Get), and update data in a remote process (MPI\_Accumulate). This resembles more conventional shared memory programming, however, with several important differences. First, all data transfers still happen as a result of a call to an MPI routine. The process performing the transfer is called the origin; the process that is the target of the transfer is called the target. Second, the local and remote completion of that transfer may require additional MPI calls. While the details are

quite lengthy and subtle, there are two modes in MPI. The first, called active target, requires each process to call a routine to begin and end both use of RMA routine (the access epoch) and access by other processes (the exposure epoch). The *Bulk Synchronous Programming* model has a very similar form.

The second mode is call passive target. In this mode, the origin process makes initiates and completes the RMA operation without requiring that any MPI routine be called on the target process. This provides something similar to a conventional shared memory model. The reason for these two modes is that there are implementation issues and opportunities for optimization with each; the programmer can pick the one that is most appropriate for their needs.

The following code fragment is an example of the MPI one-sided interface. It implements the same communication of data and the point-to-point example above. This example uses the active target synchronization with the routine MPI\_Win\_fence. MPI also provides routines to specify the region of memory that is exposed to access or updates from remote processes. The *MPI\_Win* object contains this information and can be considered as the one-sided version of the MPI Communicator.

```

MPI_Win_create(rbuf, 2, MPI_INT, sizeof(int), ...,
 &win);
MPI_Win_fence(0, win);
if (rank == 0) {
 MPI_Put(sbuf, 2, MPI_INT, 1, 0, 2, MPI_INT,
 win);
}
MPI_Win_fence(0, win);
If (rank == 1) {
 printf("Received %d %d from sender\n", rbuf[0],
 rbuf[1]);
}
MPI_Win_free(&win);

```

In the MPI-1 model, the number of processes is fixed when the program begins. In MPI-2, routines are provided to create new MPI processes as part of the same running MPI program (e.g., MPI\_Comm\_spawn) and to connect two MPI programs that are already running (e.g., MPI\_Comm\_connect and MPI\_Comm\_accept). These routines return a special kind of

communicator (called an intercommunicator) that contains two groups of processes. In the case of MPI\_Comm\_spawn, the two groups are the group of processes that called MPI\_Comm\_spawn and the group of processes created by MPI\_Comm\_spawn. Intercommunicators are part of MPI-1, but until MPI-2, they were rarely used.

MPI-2 also provides a rich set of parallel I/O routines. The viewpoint taken is that writing to a file is like sending a message and reading from a file is like receiving a message. Following that model, MPI provides routines to read and write data, using the same 3-tuple to describe the data in the MPI program. By using user-defined MPI datatypes to set a “file view” (describing which parts of a file each of the MPI processes can access), very general I/O patterns can be described in a single MPI call. And in addition to independent I/O routines, MPI provides *collective* I/O routines, where all processes (in the communicator use to open the file with MPI\_File\_open) perform coordinated I/O operations. These collective I/O routines can provide far more scalable performance than the independent I/O routines.

MPI-1 specified bindings to C and Fortran 77, and MPI-2 added a binding to C++ and defined Fortran 90 bindings. These bindings are relatively simple; there was little attempt, for example, to provide a high-level C++ binding.

In the original MPI model, a collection of processes is created (by some mechanism not defined by the MPI standard) to begin the execution of an MPI program. However, many implementations provided a program, mpirun, to start an MPI program. To provide a more standard interface, the MPI-2 standard recommends (but does not require) a program mpiexec to start MPI programs.

MPI-2 has been a mixed success. The MPI I/O routines are widely available and used by software libraries such as HDF5 and parallel netCDF. The one-sided (RMA) routines are available but with uneven implementation quality; in addition, while the MPI RMA model permits an interesting class of applications (including client-server), it is a poor match to other one-sided applications. The absense of a read-modify-write operation makes operations such as fetch-and-increment very difficult. Despite these problems, MPI

remains the dominant parallel programming model in computational science.

## The Future of MPI

MPI continues to evolve. The MPI Forum has begun discussions on the next version of MPI, which will be called MPI-3, and will most likely be a further extension of the current MPI-2 API. Among discussion for MPI-3 are the following features.

Blocking collectives communication operations can limit the scalability of algorithms. During the development of MPI-2, the MPI Forum felt that users could implement nonblocking versions of the collectives simply by creating a thread and calling a blocking MPI collective routine from within the thread. In practice, this was not sufficient. In some cases, the overhead of thread support was too costly, particularly for collectives involving small amounts of data. In addition, some systems, such as the IBM Blue Gene/L (one of the most powerful parallel computers) allowed only one thread per core, making this approach inefficient. MPI-3 is likely to offer nonblocking collectives, similar to the nonblocking point-to-point and nonblocking parallel I/O routines in MPI.

The remote-memory access interface defined in MPI-2 was designed to support certain types of operations, such as halo exchanges used in numerical simulations, and it works well for those. But it is ill suited to more dynamic uses of RMA, and it is not an effective API for the operations needed by implementations of Partitioned Global Address Space (PGAS) languages. Other APIs, such as ARMCI and GASNET, offer a more flexible interface (while requiring either more hardware support or less precise semantics than MPI). The MPI Forum is exploring ways to extend the MPI RMA model to make it a more general purpose.

One strength of MPI has been its support for components and multilingual programming. The MPI Forum is looking at ways to ensure that MPI works well with a variety of programming models, from SMP parallelism to coexistence with PGAS languages such as UPC and CAF.

Large scale system with millions of processor cores, on which MPI is likely to be used, are likely to suffer frequent faults. MPI (starting with the original MPI-1) does provide a flexible error-handling model. But after

an error, the behavior of MPI is undefined. The MPI Forum is attempting to define what faults and situations the MPI implementation should continue through (while indicating an error) to enable the development of portable, fault-tolerant programs.

The above show that MPI will evolve to address new capabilities and requirements in large-scale parallel computing systems. See [www mpi-forum.org](http://www mpi-forum.org) for information on the current status of MPI and the MPI standardization efforts.

## More on History of MPI

MPI really began at the Workshop on Standards for Message Passing in a Distributed Memory Environment in Williamsburg, Virginia, held in April, 1992. The meeting was called by Ken Kennedy, who had the foresight to see both the need and opportunity for a specification of a standard library for message passing. A preliminary proposal, developed by Dongarra, Hempel, Hey, and Walker, led the community to come together to develop a standard. An organizational meeting held at Supercomputing in Minneapolis in November, 1992, laid out the goals, along with a commitment by the author of this entry to maintain an implementation of the current draft of the standard. This draft and reference implementation helped to both refine the standard and provide initial implementations on a variety of platforms, aiding in the adoption of the new standard. Today, there are many high-quality implementations, both open source (such as MPICH2 and Open MPI) and proprietary implementations from the major parallel computer vendors.

MPI was developed by a group of researchers, application specialists, and vendor representatives. An open process was used with regular (every 6–8 weeks) meetings. Email (and later wiki) discussions were used to engage the broader community. Votes, apportioned as one per participating institution, were used to choose the standard; voting at two consecutive meetings was required, allowing for “buyer’s remorse.” More details on the process are in [4].

## Conclusion

MPI is possibly the most successful parallel programming model ever, having reached age 17 (as of 2011) and

continuing to go strong. Built upon the sound basis of communicating sequential processes and using a truly open process to define the interface, MPI, particularly for the highly scalable systems, has become the parallel programming model to beat.

## Related Entries

- ▶ [CSP \(Communicating Sequential Processes\)](#)
- ▶ [HDF5](#)
- ▶ [MPI-IO](#)
- ▶ [NetCDF I/O Library, Parallel](#)

## Bibliographic Notes and Further Reading

An annotated version of the MPI standards are available in [1, 8]. Official versions of the documents are available at [www mpi-forum.org](http://www mpi-forum.org). A tutorial introduction to MPI-1 and MPI-2 is available in [2, 3]. There are many other tutorial and research publications on MPI; there is one annual conference devoted to MPI: the EuroMPI (formerly Euro PVMMPI) meetings produce a proceeding every year in Springer’s Lecture Notes in Computer Science (e.g., see [5]).

## Bibliography

1. Gropp W, Huss-Lederman S, Lumsdaine A, Lusk E, Nitzberg B, Saphir W, Snir M (1998) MPI – The complete reference, vol 2, The MPI-2 extensions, MIT Press, Cambridge, MA
2. Gropp W, Lusk E, Skjellum A (1999) Using MPI: portable parallel programming with the message passing interface, 2nd edn. MIT Press, Cambridge, MA
3. Gropp W, Lusk E, Thakur R (1999) Using MPI-2: advanced features of the message-passing interface, MIT Press, Cambridge, MA
4. Hempel R, Walker DW (1999) The emergence of the MPI message passing standard for parallel computing. Comput Stand Interfaces 21:51–62
5. Keller R, Gabriel E, Resch MM, Dongarra J (2010) Recent advances in the message passing interface – 17th European MPI users’ group meeting. Springer, Stuttgart
6. Message Passing Interface Forum (1991) MPI2: A message passing interface standard. High Perform Comput Appl 12: 1–299
7. Message Passing Interface Forum (1994) MPI: a message passing interface standard. Int J Supercomput Appl 8:159–416
8. Snir M, Otto S, Huss-Lederman S, Walker D, Dongarra J (1998) MPI-the complete reference, vol 1, The MPI Core. MIT Press, Cambridge, MA

## MPI-2 I/O

► MPI-IO

## MPI-IO

JEAN-PIERRE PROST  
Morteau, France

### Synonyms

MPI-2 I/O

### Definition

MPI-IO is a portable interface defined by the Message Passing Interface (MPI) Forum in order to perform parallel I/O operations within distributed memory programs, leveraging MPI key concepts such as communicators, datatypes, and collective operations. It was first introduced as the I/O chapter of the second specification of the Message Passing Interface, referred to as MPI-2.

### Discussion

#### Introduction

In June 1994, the MPI Forum released their first draft MPI 1.0, defining point-to-point and collective communication operations between tasks (i.e., virtual processes) within a given context, called a communicator. Point-to-point communication operations could be either blocking or nonblocking. This draft also introduced the concept of a datatype describing a virtual data layout in the memory of the sending or the receiving process(es). In March 1995, a second draft MPI 1.1 was released to correct errors and make clarifications from the first draft.

Until July 1997, implementations of the MPI 1.1 draft were released by major computer companies and by open source initiatives, and practical experiences with the “standard” were gathered by numerous distributed memory application developers.

Among the shortcomings of the draft specification that this experimentation helped identify was the lack

of support for parallel I/O in a portable way across existing and emerging distributed and parallel file systems, limiting the scope of distributed memory applications to in-memory processing or to coding using either proprietary I/O interfaces or the Posix interface that these file systems supported.

Therefore, in July 1997, the MPI Forum published the second version of MPI, referred to as MPI 2.0, that contained an I/O chapter, specifying a portable interface for parallel I/O aimed at MPI applications. MPI-IO was born.

The MPI key concepts are leveraged by the MPI-IO specification. The basic idea behind MPI-IO is to consider reading a file like receiving data from the file into virtual memory and writing a file like sending data from virtual memory into the file. The file can therefore be assimilated to either a sending task in read operations or a receiving task in write operations.

With this analogy in mind, independent I/O read and write operations are like point-to-point receive and send operations, respectively, and collective I/O operations are like collective communication operations between the group of initiating tasks in the context of a communicator and the file. These read and write operations, just like communication operations, can be blocking or nonblocking. And describing from where to read data within the file or where to write data into the file is only a matter of defining a file datatype, referred to as a file view, after opening the file.

In order to gain better advantage of parallel file system specifics and reach optimal performance, without sacrificing application portability across parallel systems, file hints are introduced. Each target run time environment is free to support a given file hint, and setting an unsupported file hint to an open file simply has no (positive or negative) effect.

#### Access Pattern Expression

In order to specify the access pattern associated with an open file, a file view must be defined by each task accessing the file, unless the default view is to be used and each task sees the file as a linear byte stream.

A file view essentially specifies:

- An absolute displacement into the file (in bytes), specifying where the file pointer ought to be placed

in the file before the next access takes place by the calling task;

- An elementary datatype (etype), representing the MPI datatype of each element within the file to be accessed by the calling task;
- A file datatype (filetype), representing the layout of the elements to be accessed by the calling task; this layout must be an MPI datatype derived from the etype.

Typically, all tasks participating to the collaborative access of a given file set complementary views of the file, in order to partition the file data among themselves. [Figure 1](#) shows how a file can be partitioned in a cyclic manner across three tasks by chunks of two elements each.

A file view is associated with an open file by modifying the file handle returned when the file was opened. At any given time, only one file view can be set by a given task on a given open file. Setting a new file view changes the current file view.

*Note:* The default file view corresponds to a displacement of zero bytes, and to the etype and the filetype set to MPI\_BYTE.

## Data Access Functions

File data can be accessed either independently by each task, which has opened the file, or collectively by all tasks having opened the file.

The area of the task's virtual memory involved in the data access is specified by a memory buffer's address, an MPI datatype, and a count.

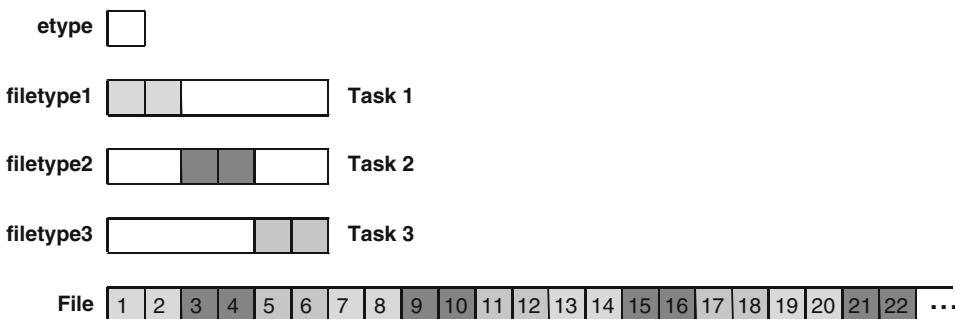
The region of the file to be accessed is specified by the file view associated with the open file and a file

position, which represents the location within the file where the access is to begin.

The file position is expressed either as an explicit offset within the file view, i.e., as a number of etype elements to be skipped within the file view before reading or writing file elements, or as the current position of the task's file pointer.

Each open file has two types of file pointers maintained by the MPI environment, an individual file pointer for each task and a shared file pointer shared by all tasks having opened the file. The file pointer to be used for a given data access is determined by the name of the data access function called. After opening a file, each task's file pointer and the shared file pointer are set to zero and point to the byte within the file associated to the displacement specified in the file view or to the beginning of the file if the active file view is the default view. Note that the same file view must be set by all tasks having opened the file when the shared file pointer is to be used.

All data access can be either blocking or nonblocking. In the case of blocking calls, the call returns to the task when the data has been read from (or written to) the file. In the case of nonblocking calls, the call returns immediately to the task, which may choose later on to either test whether (call to the MPI\_Test or MPI\_Testall functions) or wait until (call to the MPI\_Wait or MPI\_Waitall functions) the data has been read from (or written to) the file. Note however that nonblocking collective data access functions are called split collective since they are made of two suboperations, a (nonblocking) begin operation that initiates the access and a (blocking) end operation that completes the access for all tasks synchronously.



**MPI-IO. Fig. 1** Partitioning an open file across three tasks

MPI-IO. Table 1 Data access function characteristics

| Data access function name    | Independent/collective | Blocking/nonblocking | File position   |
|------------------------------|------------------------|----------------------|-----------------|
| MPI_File_read_at             | Independent            | Blocking             | Explicit offset |
| MPI_File_write_at            | Independent            | Blocking             | Explicit offset |
| MPI_File_iread_at            | Independent            | Nonblocking          | Explicit offset |
| MPI_File_iwrite_at           | Independent            | Nonblocking          | Explicit offset |
| MPI_File_read_at_all         | Collective             | Blocking             | Explicit offset |
| MPI_File_write_at_all        | Collective             | Blocking             | Explicit offset |
| MPI_File_read_at_all_begin   | Collective             | Split collective     | Explicit offset |
| MPI_File_read_at_all_end     |                        |                      |                 |
| MPI_File_write_at_all_begin  | Collective             | Split collective     | Explicit offset |
| MPI_File_write_at_all_end    |                        |                      |                 |
| MPI_File_read                | Independent            | Blocking             | Individual fp   |
| MPI_File_write               | Independent            | Blocking             | Individual fp   |
| MPI_File_iread               | Independent            | Nonblocking          | Individual fp   |
| MPI_File_iwrite              | Independent            | Nonblocking          | Individual fp   |
| MPI_File_read_all            | Collective             | Blocking             | Individual fp   |
| MPI_File_write_all           | Collective             | Blocking             | Individual fp   |
| MPI_File_read_all_begin      | Collective             | Split collective     | Individual fp   |
| MPI_File_read_all_end        |                        |                      |                 |
| MPI_File_write_all_begin     | Collective             | Split collective     | Individual fp   |
| MPI_File_write_all_end       |                        |                      |                 |
| MPI_File_read_shared         | Independent            | Blocking             | Shared fp       |
| MPI_File_write_shared        | Independent            | Blocking             | Shared fp       |
| MPI_File_iread_shared        | Independent            | Nonblocking          | Shared fp       |
| MPI_File_iwrite_shared       | Independent            | Nonblocking          | Shared fp       |
| MPI_File_read_ordered        | Collective             | Blocking             | Shared fp       |
| MPI_File_write_ordered       | Collective             | Blocking             | Shared fp       |
| MPI_File_read_ordered_begin  | Collective             | Split collective     | Shared fp       |
| MPI_File_read_ordered_end    |                        |                      |                 |
| MPI_File_write_ordered_begin | Collective             | Split collective     | Shared fp       |
| MPI_File_write_ordered_end   |                        |                      |                 |

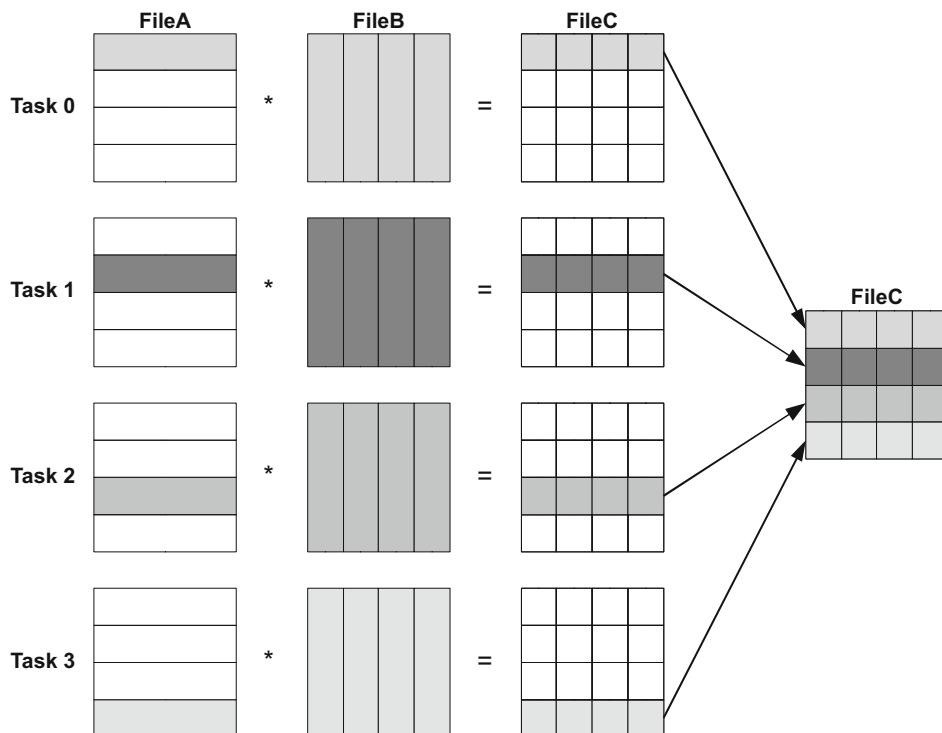
Table 1 specifies for each data access function whether the access is independent or collective, whether it is blocking or nonblocking, and whether it uses an explicit offset or the individual (or shared) file pointer (fp) to express the file position.

Assuming *fh* refers to an open file, whose file view has just been set to the view described in Fig. 1,

Table 2 above identifies the calling task and the *etype* elements read or written by each data access function call stipulated. It is assumed that *buf* points to a valid task memory area that contains the appropriate number of *etype* elements being read or can contain the appropriate number of *etype* elements being written.

**MPI-IO. Table 2** Examples of data access function calls

| Data access function call and calling task(s)                                                                                                                                                                                                                                                                                                                                                            | Etype elements read or written                                  |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------|
| Task 1: <code>MPI_File_read_at(fh, 2, buf, 3, etype, status);</code>                                                                                                                                                                                                                                                                                                                                     | Task 1: elements 7, 8, 13                                       |
| Task 1: <code>MPI_File_write_at_all(fh, 0, buf, 4, etype, status);</code><br>Task 2: <code>MPI_File_write_at_all(fh, 0, buf, 4, etype, status);</code><br>Task 3: <code>MPI_File_write_at_all(fh, 0, buf, 4, etype, status);</code>                                                                                                                                                                      | Task 1: 1, 2, 7, 8 – Task 2: 3, 4, 9, 10 – Task 3: 5, 6, 11, 12 |
| Task 2: <code>MPI_File_read(fh, buf, 5, etype, status);</code><br>Task 2: <code>MPI_File_read(fh, buf, 2, etype, status);</code>                                                                                                                                                                                                                                                                         | Task 2: 3, 4, 9, 10, 15<br>Task 2: 16, 21                       |
| Task 3: <code>MPI_File_iwrite(fh, buf, 3, etype, status);</code>                                                                                                                                                                                                                                                                                                                                         | Task 3: 5, 6, 11                                                |
| Task 1: <code>MPI_File_read_all_begin(fh, buf, 3, etype);</code><br>Task 2: <code>MPI_File_read_all_begin(fh, buf, 3, etype);</code><br>Task 3: <code>MPI_File_read_all_begin(fh, buf, 3, etype);</code><br>Task 1: <code>MPI_File_read_all_end(fh, buf, status);</code><br>Task 2: <code>MPI_File_read_all_end(fh, buf, status);</code><br>Task 3: <code>MPI_File_read_all_end(fh, buf, status);</code> | Task 1: 1, 2, 7 – Task 2: 3, 4, 9 – Task 3: 5, 6, 11            |

**MPI-IO. Fig. 2** Matrix partitioning across four computing tasks

### File Consistency

File consistency determines the outcome of concurrent and conflicting accesses to the same file by multiple tasks. Two accesses are concurrent if the respective data spans associated with the accesses overlap. They conflict

if they are concurrent and at least one access is a write operation.

Sequential consistency of concurrent and conflicting accesses to the same file guarantees each access is atomic and the outcome of the accesses is the same as

```

#include "sys/types.h"
#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include "mpi.h"

#define P 4 /* number of tasks */
#define Nb 64 /* slice size */
#define Ng (Nb * P) /* global size of matrix */

static char fileA[] = "fileA";
static char fileB[] = "fileB";
static char fileC[] = "fileC";

void
main(int argc, char *argv[])
{
 int i, j, k;
 int iv;
 int myrank, commsize;
 int extent;
 int colext;
 int slicelen;
 int sliceext;
 int length[3];
 MPI_Aint disp[3];
 MPI_Datatype type[3];
 MPI_Datatype ftypeA;
 MPI_Datatype ftypeB;
 MPI_Datatype ftypeC;
 MPI_Datatype slice_typeB;
 MPI_Datatype block_typeC;
 MPI_Datatype slice_typeC;
 int mode;
 MPI_File fh_A, fh_B, fh_C;
 MPI_Offset offsetA;
 MPI_Offset offsetB;
 MPI_Offset offsetC;
 MPI_Status status;
 int val_A[Nb][Ng], val_B[Ng][Nb], val_C[Nb][Nb];

 /* initialize MPI */
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 MPI_Comm_size(MPI_COMM_WORLD, &commsize);
}

```

**MPI-IO. Fig. 3** Sample code for an out-of-core 2D matrix multiplication

that, which would have been achieved if the accesses were performed in a serial order, although that order is unspecified.

By default, sequential consistency is only ensured by MPI-IO for concurrent accesses to the same file by the tasks, which opened it in the same collective operation.

Sequential consistency for conflicting accesses can only be ensured across a set of tasks, if they enabled the atomic mode when they collectively opened the file.

For all other concurrent and conflicting accesses, sequential consistency cannot be ensured unless the

```

MPI_Type_extent(MPI_INT, &extent);
coext = Nb * extent;
slicelen = Ng * Nb;
sliceext = slicelen * extent;

/* create filetype for matrix A */
length[0] = 1;
length[1] = slicelen;
length[2] = 1;
disp[0] = 0;
disp[1] = sliceext * myrank;
disp[2] = sliceext * P;
type[0] = MPI_LB;
type[1] = MPI_INT;
type[2] = MPI_UB;
MPI_Type_struct(3, length, disp, type, &ftypeA);
MPI_Type_commit(&ftypeA);

/* create filetype for matrix B */
MPI_Type_vector(Ng, Nb, Ng, MPI_INT, &slice_typeB);
MPI_Type_hvector(P, 1, coext, slice_typeB, &ftypeB);
MPI_Type_commit(&ftypeB);

/* create filetype for matrix C */
MPI_Type_vector(Nb, Nb, Ng, MPI_INT, &block_typeC);
MPI_Type_hvector(P, 1, coext, block_typeC, &slice_typeC);
length[0] = 1;
length[1] = 1;
length[2] = 1;
disp[0] = 0;
disp[1] = sliceext * myrank;
disp[2] = sliceext * P;
type[0] = MPI_LB;
type[1] = slice_typeC;
type[2] = MPI_UB;
MPI_Type_struct(3, length, disp, type, &ftypeC);
MPI_Type_commit(&ftypeC);

/* open files */
mode = MPI_MODE_RDONLY;
MPI_File_open(MPI_COMM_WORLD, fileA, mode, MPI_INFO_NULL, &fh_A);
mode = MPI_MODE_RDONLY;
MPI_File_open(MPI_COMM_WORLD, fileB, mode, MPI_INFO_NULL, &fh_B);
mode = MPI_MODE_CREATE | MPI_MODE_WRONLY;
MPI_File_open(MPI_COMM_WORLD, fileC, mode, MPI_INFO_NULL, &fh_C);

```

**MPI-IO. Fig. 3** (Continued)

tasks accessing the same file explicitly perform appropriate file sync operations collectively among themselves.

### File Hints for Performance

File hints can be specified when a task opens a file in order to take advantage of parallel file system specifics or minimize the use of system resources,

leading to optimized parallel I/O when accessing that file.

Reserved hints are defined by MPI-IO to set the access style to the open file (random, sequential, reverse sequential, read mostly, write mostly, read once, write once), to enable collective buffering (allowing the same buffer to be shared across accessing tasks), to specify an

```

/* set file error-handlers to MPI_ERRORS_ARE_FATAL in order to shorten sample code
 (default file error handler is MPI_ERRORS_RETURN) */
MPI_File_set_errhandler(fh_A, MPI_ERRORS_ARE_FATAL);
MPI_File_set_errhandler(fh_B, MPI_ERRORS_ARE_FATAL);
MPI_File_set_errhandler(fh_C, MPI_ERRORS_ARE_FATAL);

/* set views */
MPI_File_set_view (fh_A, MPI_OFFSET_ZERO, MPI_INT, ftypeA, "native", MPI_INFO_NULL);
MPI_File_set_view(fh_B, MPI_OFFSET_ZERO, MPI_INT, ftypeB, "native", MPI_INFO_NULL);
MPI_File_set_view(fh_C, MPI_OFFSET_ZERO, MPI_INT, ftypeC, "native", MPI_INFO_NULL);

offsetA = MPI_OFFSET_ZERO;
offsetB = MPI_OFFSET_ZERO;
offsetC = MPI_OFFSET_ZERO;

/* read horizontal slice of A */
MPI_File_read_at_all(fh_A, offsetA, val_A, slicelen, MPI_INT, &status);

/* loop on vertical slices */
for (iv = 0; iv < P; iv++) {

 /* read next vertical slice of B */
 MPI_File_read_at_all(fh_B, offsetB, val_B, slicelen, MPI_INT, &status);
 offsetB += slicelen;

 /* compute block product */
 for (i = 0; i < Nb; i++) {
 for (j = 0; j < Nb; j++) {
 val_C[i][j] = 0;
 for (k = 0; k < Ng; k++) {
 val_C[i][j] += val_A[i][k] * val_B[k][j];
 }
 }
 }

 /* write block of C */
 MPI_File_write_at_all(fh_C, offsetC, val_C, Nb * Nb, MPI_INT, &status);
 offsetC += Nb * Nb;
}

/* close files */
MPI_File_close(&fh_A);
MPI_File_close(&fh_B);
MPI_File_close(&fh_C);

/* free filetypes */
MPI_Type_free(&ftypeA);
MPI_Type_free(&slice_typeB);
MPI_Type_free(&ftypeB);
MPI_Type_free(&block_typeC);
MPI_Type_free(&slice_typeC);
MPI_Type_free(&ftypeC);

/* finalize MPI */
MPI_Finalize();
}

```

**MPI-IO. Fig. 3** (Continued)

I/O node list and a striping pattern across these nodes. Each implementation is free to support or not these reserved hints. If they are not supported, they are simply ignored. If they are supported, the implementation

should comply with the semantics defined for them in the MPI-IO specification.

Additional implementation specific hints can be supported by an implementation. The naming of these

hints should be chosen in such a way that conflicting names across implementations be prevented. For instance, IBM defined additional hints in their MPI-IO implementation on top of the IBM General Parallel File System (GPFS) in order to specify a sparse access style to the file, to control GPFS data prefetching, or to benefit from GPFS data shipping feature. This feature partitions a file across a specific number of I/O agents and ships all data accesses from the MPI tasks to these I/O agents over the interconnecting network, hence limiting the number of concurrent tasks accessing the file and grouping several small accesses into fewer larger data blocks to be accessed.

It is clear that file hints are to be used cautiously and only when relevant to the I/O patterns exhibited by the application on a per file basis. Applying inappropriate file hints, when supported by a given implementation, may lead to decreased performance.

### Sample Code: Out-of-Core Matrix Multiplication

In order to illustrate the main MPI-IO concepts, Fig. 2 above depicts the matrix partitioning scheme that is used in order to perform the out-of-core multiplication of two square matrices A and B. Each task is computing a slice of the resulting matrix C.

Figure 3 presents the corresponding code. Matrix A is initially stored in file “fileA,” matrix B in file “fileB,” and the result of the multiplication, matrix C, is stored in file “fileC.” All matrices are 256\*256 square matrices containing integer numbers, stored in row major order.

### Related Entries

- ▶ [Benchmarks](#)
- ▶ [Collective Communication](#)
- ▶ [File Systems](#)
- ▶ [I/O](#)
- ▶ [Metrics](#)
- ▶ [MPI \(Message Passing Interface\)](#)

### Bibliographic Notes and Further Reading

The MPI-IO interface [2] was defined in the late 1990s by the Message Passing Interface Forum [5] in an attempt to provide an interface for portable and efficient parallel I/O operations in distributed memory programming environments.

Implementations have been developed over the years on top of various parallel file systems, such as the General Parallel File System on IBM Scalable Power systems [7] and on the Blue Gene/L supercomputer [12], the Lustre file system [3], the Parallel Virtual File System Version 2 [4], or the NEC SX Global File System [11].

An open source implementation of MPI-IO, referred to as ROMIO [10], is available. It provides portability onto various back-end parallel file systems through an abstract-device I/O interface [9].

Performance studies [1, 6] and benchmarks [8] have been carried out to demonstrate the efficiency that can be achieved for read and write operations through specific I/O strategies [1, 6] and the use of MPI-IO’s file hints [8].

### Bibliography

1. Allsopp NK, Hague JF, Prost JP (2001) Experiences in using MPI-IO on top of GPFS for the IFS weather forecast code. In: Proc Euro-Par, Manchester, UK
2. Corbett P, Feitelson D, Fineberg S, Hsu Y, Nitzberg B, Prost JP, Snir M, Traversat B, Wong P (2001) Overview of the MPI-IO parallel I/O interface. Chapter in: High performance mass storage and parallel I/O: technologies and applications, Jin H, Cortes T, Buyya R (eds). Wiley/IEEE Press, ISBN: 0-471-20809-4
3. Dickens P, Logan J (2008) Towards a high performance implementation of MPI-IO on the Lustre file system. In: Proc GADA’08: grid computing, high-performance and distributed applications, Monterrey, Mexico
4. Latham R, Ross R, Thakur R (2004) The impact of file systems on MPI-IO scalability. In: Proc EuroPVM/MPI, 2004, Budapest
5. Message Passing Interface Forum (1998) MPI-2: a message passing interface standard. HPCA 12(1–2)
6. Prost JP, Treumann R, Hedges R, Jia B, Koniges A (2001) MPI-IO/GPFS, an optimized implementation on top of GPFS. In: Proc Supercomputing, Denver, CO
7. Prost JP, Treumann R, Hedges R, Koniges A, White A (2000) Towards a high-performance implementation of MPI-IO on top of GPFS. In: Proc Euro-Par 2000, Munich, Germany, pp 1253–1262
8. Rabenseifner R, Koniges A, Prost JP, Hedges R (2004) The parallel effective I/O bandwidth benchmark: b\_eff\_io. Chapter in: parallel I/O for cluster computing, Cérin C, Jin H (eds). Kogan Page Science, London, ISBN 1-903996-50-3, pp 107–132
9. Thakur R, Gropp W, Lusk E (1996) An abstract-device interface for implementing portable parallel-I/O interfaces. In: Proc 6th Symposium on the frontiers of massively parallel computation Annapolis, MD, pp 180–187
10. Thakur R, Gropp W, Lusk E (1999) On implementing MPI-IO portably and with high performance. In: Proc 6th Workshop on Input/Output in parallel and distributed systems, Atlanta, GA, pp 23–32

11. Worringen J, Träff JL, Ritzdorf H (2003) Improving generic non-contiguous file access for MPI-IO. In: Proc 10th European PVM/MPI User's Group Meeting, Venice, Italy, Lecture Notes in Computer Science, vol 2840, Springer-Verlag, pp 309–318
12. Yu H, Sahoo RK, Howson C, Almasi G, Castanos JG, Gupta M, Moreira JE, Parker JJ, Engelsiepen TE, Ross R, Thakur R, Latham R, Gropp W (2006) High performance file I/O for the BlueGene/L supercomputer. In: Proc 12th International symposium on high-performance computer architecture (HPCA-12), Austin, TX

## MPP

### Synonyms

[SIMD \(Single Instruction, Multiple Data\) Machines](#)

The Goodyear Massively Parallel Processor (MPP) [1] was an SIMD machine built by Goodyear Aerospace Corporation which was a subsidiary of Goodyear Tire and Rubber Company. The machine was delivered in 1983.

### Related Entries

- [Cray XT3 and Cray XT Series of Supercomputers](#)
- [Flynn's Taxonomy](#)

### Bibliography

1. Batcher KE (1980) Design of a massively parallel processor. *IEEE Trans Comput* C29:836–840

## Mul-T

- [Multilisp](#)

## Multicomputers

- [Clusters](#)
- [Distributed-Memory Multiprocessor](#)
- [Hypercubes and Meshes](#)

## Multicore Networks

- [Interconnection Networks](#)

## Multiflow Computer

GEOFF LOWNEY

Intel Corporation, Hudson, MA, USA

### Definition

The Multiflow Trace was a VLIW computer system, designed and manufactured by Multiflow Computer, Inc. of Branford, CT, USA. The Multiflow Trace was distinguished by its ability to issue as many as 7, 14, or 28 operations in each instruction, whose width could be up to 1,024 bits, depending upon the model. Designed as a target for a *trace scheduling* compiler, users were not required to divide the code into parallel routines. The compiler, sometimes with guidance from the programmer, found instruction-level parallelism in ordinary code, and constructed the very long instructions prior to the running of the program. Design of the Multiflow Trace began in 1984, with the first systems delivered in 1987. The last of the approximately 120 systems sold was delivered in 1990.

### Discussion

#### Introduction

In the late 1970s, instruction-level parallelism (ILP) was a research topic. In fact, Joseph (Josh) Fisher coined the term in 1980, while a professor at Yale University. Dynamic ILP, where the parallelism is discovered by the hardware as the program runs, had been explored in supercomputing. Static ILP, where the microcoder, assembly language programmer, or compiler scheduled multiple operations into a single instruction, existed in microcoded machines and in some specialized accelerators. Several researchers had demonstrated that only limited amounts of instruction-level parallelism could be discovered in a basic block, which seemed a natural boundary for compile-time scheduling.

In his dissertation research at New York University, Fisher demonstrated that extensive instruction-level parallelism could be discovered by scheduling across multiple basic blocks. His compiler technique, called *trace scheduling*, identified paths, or traces, through a program at compile-time, unrolling loops if needed. Operations on the trace were scheduled across basic block boundaries and packed into long instruction words.

Fisher's ideas made it practical to build general-purpose computers designed to exploit instruction-level parallelism. No expensive dynamic hardware would be required. Fisher called these machines VLIW (Very Long Instruction Word) computers, and his research team at Yale developed the concepts behind the VLIW architecture and trace scheduling compilers.

Fisher and two of his colleagues from Yale (John Ruttenberg and John O'Donnell) founded Multiflow in 1984. Compiler and hardware design started immediately, with the compiler written at Yale serving as an initial guide to the hardware design. Three years later, Multiflow delivered the Trace 7/200, followed quickly by the 14/200, and the 7/300, 14/300, and 28/300, all running the Unix operating system, with code compiled by Multiflow's Trace Scheduling Compiler. During the 3 years of their delivery, the Trace systems were consistently at or near the top of the performance/price ratio of scientific computers, despite their relatively modest 65ns cycle time.

## Design Principles

The Trace machines were designed as a target of a trace scheduling compiler. As much as possible, all scheduling decisions and resource allocation checks were made by the compiler, not the hardware. At compile-time, the compiler modeled how the program would behave on a cycle-by-cycle basis, created a schedule of its execution, and then at run-time, the hardware executed the schedule. For events that were not modeled by the compiler, such as an instruction cache miss, the entire execution pipeline was stalled until execution resumes.

Compile-time modeling of program behavior was the essence of the trace scheduling design. This idea is also applicable to modern processors that support more dynamic behavior in hardware, and it has been used successfully in many compilers derived from the Multiflow compiler.

## The Trace Machines

*Implementation.* There were two generations of Multiflow computers. The 200 series, first shipped in January, 1987 with a 65 ns cycle time, was implemented in CMOS gate arrays and TTL logic, using Weitek CMOS floating point chips. The 300 series followed in July, 1988, using the Bipolar Integrated Technologies (BIT) ECL

floating point parts. The cycle time remained at 65ns. A third generation of Multiflow machines was under design when the company closed in March 1990. It was an ECL semi-custom implementation targeting a 15 ns cycle time.

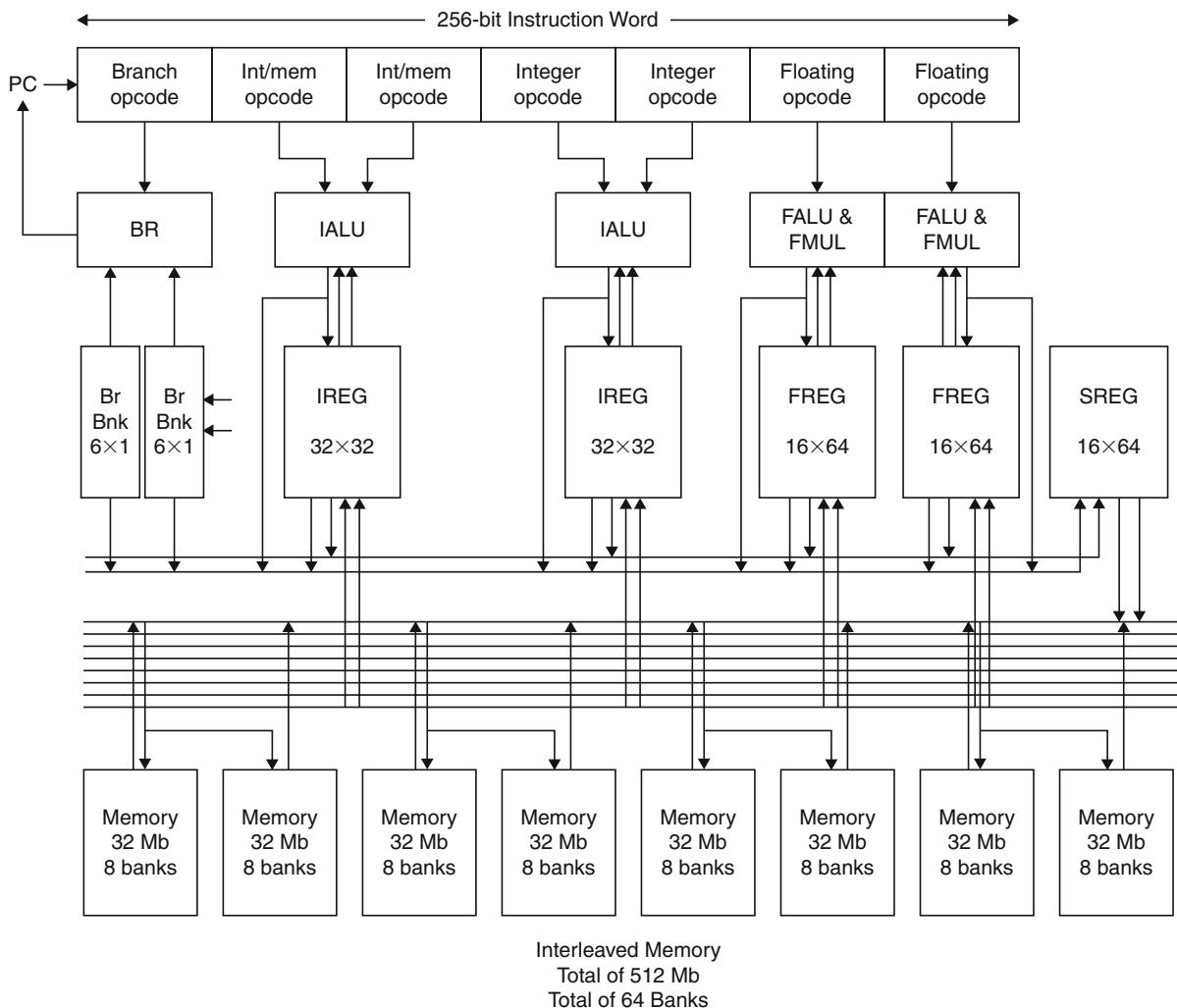
*Basic architecture.* An instruction on the Multiflow Trace machine consisted of several RISC-level operations packed together into a single wide instruction. The wide instructions directly controlled multiple register files, pipelined memory units, branch units, integer ALUs, and pipelined floating point units. Execution was fully synchronous, with a single program counter directing instruction fetch. All operations took a statically predictable number of machine cycles to complete. Pipelining was exposed at the instruction level. The processor was not scoreboarded, and machine resources could be oversubscribed. The memory system was interleaved. The compiler needed to avoid register conflicts, schedule the machine resources, and manage the memory system.

The machines came in three widths: a seven-wide, which had a 256-bit instruction issuing seven operations; a 14-wide with a 512-bit instruction; and a 28-wide with a 1,024-bit instruction. The wider processors were organized as multiple copies of the seven-wide functional units; the seven-wide group of functional units was called a cluster. A block diagram of the 7/300 is below ([Fig. 1](#)).

There were four functional units per cluster: two integer units and two floating units. In addition, each cluster could compute a branch target. An instruction was divided into two 65 ns *beats*. The integer ALUs could issue in the early and late beats of an instruction. The floating ALUs and branch unit could issue only in the early beat. Altogether, the cluster had the resources to issue seven operations for each instruction.

Most integer ALU operations were complete in a single beat. The load pipeline was seven beats. On the 300 series, the floating point pipelines were four beats. Branches issued in the early beat and the branch target was reached on the following instruction, effectively a two beat pipeline.

There are nine register files per cluster. Data going to memory was first moved to a store file. Branch banks were used to control conditional branches and the select operation.



**Multiflow Computer. Fig. 1** Block diagram of the 7/300

The instruction cache held 8K instructions. Each cluster had a slice of the cache, holding the instruction fields which control the cluster. There was no data cache.

The memory system supported 512 MB of physical memory with up to 64 way interleaving. The virtual address space was 4 GB.

There were two IO processors. Each supported a 246 MB/s DMA channel to main memory and two 20 MB/s VME busses.

**Code sample.** The code sample below contains two instructions of 14/300 code, extracted from the compiled inner loop of the  $100 \times 100$  LINPACK benchmark. Each operation is listed on a separate line. The first two fields identify the cluster and the functional unit

to perform the operation, the remainder of the line describes the operation. Note the destination address is qualified with a register-bank name (e.g., sb1.r0); the ALUs could target any register bank in the machine (with some restrictions). There was extra latency in reaching a remote bank.

```

instr c10 ialu0e st.64 sb1.r0,r2,17#144
 c10 ialu1e cgt.s32 l1lbb.r4,r34,6#31
 c10 falu0e add.f64 lsb.r4,r8,r0
 c10 falu1e add.f64 lsb.r6,r40,r32
 c10 ialu01 dld.64 fb1.r4,r2,17#208
 c11 ialu0e dld.64 fb1.r34,r1,17#216

```

```

c11 ialule cgt.s32 lilbb.r3,r32,zero
c11 falu0e add.f64 lsb.r4,r8,r6
c11 falu1e add.f64 lsb.r6,r40,r38
c11 ialu0l st.64 sb1.r2,r1,17#152
c11 ialu1L add.u32 lib.r32,r36,6#32
c11 br true and r3 L23?3
c10 br false or r4 L24?3;

instr c10 ialu0e dld.64 fb0.r0,r2,17#224
c10 ialule cgt.s32 lilbb.r3,r34,6#30
c10 falu0e mpy.f64 lfb.r10,r2,r10
c10 falu1e mpy.f64 lfb.r42,r34,r42
c10 ialu0l st.64 sb0.r4,r2,17#160
c11 ialu0e dld.64 fb0.r32,r1,17#232
c11 ialule cgt.s32 lilbb.r4,r35,6#29
c11 falu0e mpy.f64 lfb.r10,r0,r10
c11 falu1e mpy.f64 lfb.r42,r32,r42
c11 ialu0l st.64 sb0.r6,r1,17#168
c11 ialu1L bor.32 ib0.r32,zero,r32
c11 br false or r4 L25?3
c10 br true and r3 L26?3;

```

*Data types.* The natural data types of the machine were 32-bit signed and unsigned integers, 32-bit pointers, 32-bit IEEE-format single precision, and 64-bit IEEE-format double precision. 16-bit integers and 8-bit characters were supported with extract and merge operations; bit strings with shifts and bitwise logicals; long integers by add with carry; and booleans with normalized logicals. There was no hardware support for extended IEEE precision, denormalized numbers, or gradual underflow.

Loads and stores were either 32 or 64 bits. Natural alignment was required for high performance; misaligned references were supported through trap code, with a substantial performance penalty.

Memory was byte addressed. Byte addressing eased the porting of C programs from byte-addressed processors such as the VAX and the Motorola 68000. The low bits of the address were ignored in a load or a store, but were read by extract and merge operations.

*Memory system and data paths.* The Multiflow Trace had a two-level interleaved memory hierarchy exposed

to the compiler. All memory references went directly to main memory; there was no data cache. There were eight memory cards, each of which contained eight banks. Each bank could hold 8 MB, for a total capacity of 512 MB. Memory was interleaved across the cards and then the banks. The low byte of an address determined its bank: bits 0–1 were ignored, bits 2–4 selected a card, and bits 5–7 selected a bank.

Data was returned from memory on a set of global busses. These busses were shared with moves of data between register files on different clusters; to maintain full memory bandwidth on a 28/30, the number and placement of data moves needed to be planned carefully.

Each level of interleaving had a potential conflict.

- Card/bus conflict. Within a single beat, all references were required to be distinct cards and to use distinct busses. If two references conflicted on a card or a bus, the result was an undefined program error.
- Bank conflicts. A memory bank was busy for four beats from the time it was accessed. If another reference touched the same bank within the four-beat window, the entire machine stalled. To achieve maximum performance, the compiler was required to schedule successive references to distinct banks.

A 28/300 could generate four references per beat, and, if properly scheduled, the full memory bandwidth of the machine could be sustained without stalling.

*Global resources.* In addition to functional units and register banks, the Trace machines had a number of global shared resources that needed to be managed by the compiler.

- Register file write-ports. Each integer and floating register file could accept at most two writes per beat, one of which could come from a local ALU. Each branch bank could accept one write per beat. Each store file could accept two writes per beat.
- Global busses. There were ten global busses; each could hold a distinct 32-bit value per beat. The hardware contained routing logic; the compiler needed only to guarantee that the number of busses was not oversubscribed in a beat.
- Global control. There was one set of global controller resources, which controlled access to link registers used for subroutine calls and indirect branches.

*Integer units.* The integer units executed a set of traditional RISC operations. There were a number of features added to support trace scheduling.

- The most common code optimization in trace scheduling was to move operations above a conditional branch and execute them speculatively, before the branch condition was known. To support speculative execution, there was a set of *dismissible load* operations. These operations performed a normal load and also set a flag to enable the exception handler to know the load was being performed speculatively.
- All operations that computed booleans were invertible. Invertible branches enabled the trace scheduler to lay out a trace as straight-line code by inverting branch conditions as necessary.
- A 3-input, one output *select* operation ( $a = b ? c : d$ ) enabled many short forward branches to be mapped into straight-line code.

A conditional branch was a two-operation sequence. An operation targeted a branch bank register, and then the branch read the register. Separate register files for the branch units relieved pressure on the integer register files and provided additional operand bandwidth to support the simultaneous branch operations.

The two integer ALUs per cluster were asymmetric; only one could issue memory references.

Due to limits on the size of the gate arrays, each integer ALU had its own register file. This fact, coupled with the low latency of integer operations, made it difficult for the instruction scheduler to exploit parallelism in integer code. The cost of moving data between register files often offset the gains of parallelism.

*Floating units.* Each floating unit had register file with 15 64-bit registers available to the compiler. To keep the pipelines full, nine distinct destination registers were required for operations in flight. This left only six registers to hold variables, common sub-expressions, and the results of operations that are not immediately consumed.

There was no pipelined floating move. A move between floating registers took one beat and consumed a register write-port resource. This could prevent another floating point operation issued earlier from

using the same write-port; in some situations, a floating move could lock out two floating point operations.

*Instruction encoding.* The instruction encodings were large for code that did not use all of the functional units in each instruction. A *no-op* operation (NOP) indicated an unused functional unit. To save space in memory and on disk, object code was stored with NOP operations eliminated. Instructions were grouped in blocks of four, and the non-NOP operations were stored with a preceding mask word that indicated which NOPs have been eliminated. When an instruction was loaded into the instruction cache, it was expanded into its full width.

To save space in the instruction cache, an instruction could encode a multi-beat NOP, which instructed the processor to stall for the specified number of beats before executing the following instruction.

The large instruction format provided a generous number of immediate operands. The compiler used immediates heavily. Constants were never loaded from memory; they were constructed from the instruction word. Double precision constants were pieced together out of two immediate fields. The global pointer scheme used by most RISC machines was not required.

M

*Trap code.* Trap hardware and trap code supported virtual memory by trapping on references to unmapped pages and on stores to write-protected pages.

To prevent unwarranted memory faults on speculative loads, the compiler used the dismissible load operation. If a dismissible load trapped, the trap code would not signal an exception. If required, a translation buffer miss or a page fault was serviced, and, if successful, the memory reference was replayed by the trap code and the value was returned. Otherwise, a NAN or integer zero was returned. In all cases, computation continued. NaNs were propagated by the floating units, and checked only when they were written to memory or converted to integers or booleans. Correct programs would run correctly with speculative execution, but an incorrect program may have missed an exception that it would have signaled if compiled without speculative execution.

The hardware optionally supported precise floating exceptions, but the compiler could not move floating operations above conditional branches if this mode was in use. This mode was used when compiling for debugging.

The trap code supported access to misaligned data by piecing together the referenced datum from multiple 32-bit words. In doing so, it placed 8- and 16-bit quantities in the correct place in a 32-bit word so that extracts and merges would work correctly.

## Trace Scheduling Compiler

The Multiflow compiler supported standard C and FORTRAN and discovered instruction-level parallelism in ordinary code. The compiler created a schedule of execution for the program being compiled, modeling the behavior of the program as much as possible at compile-time. It addressed several challenges posed by the Multiflow Trace machines.

- *Find Parallelism.* There were a large number of pipelined functional units, requiring from 10–50 data independent operations in flight to fill the machine. This large amount of instruction-level parallelism required scheduling beyond basic blocks and also a strategy for finding parallelism across loop iterations.
- *Model the machine.* Machine resources could be oversubscribed, and the pipelines used resources in every cycle. The compiler needed to model precisely the cycle-by-cycle state of the machine even in situations where such a model was not critical for performance.
- *Schedule the memory system.* There was an interleaved memory system that was managed by the compiler. Card conflicts caused program error; bank conflicts stalled the program.
- *Schedule data motion.* Each functional unit had its own register file. It cost an operation to move a value between registers, although remote register files could be targeted directly at the cost of an extra beat of latency.

The compiler had a traditional three-phase design: front end, optimizer, and back end. The optimizer focused both on reducing computation and exposing instruction-level parallelism. The back end was organized around the trace scheduling algorithm.

*Front ends.* The compiler supported user-level directives to enable the programmer to pass additional information to the compiler. Loop unrolling directives specified how a loop should be unrolled. Inline directives selected

functions to be inlined. Memory-reference directives asserted facts about addresses to aid memory alias analysis. Trace-picking directives specified branch probabilities and loop trip counts. In addition, the front end could instrument a program to count basic block executions. The instrumentation was saved in a database which could be read back on subsequent compilations. This information was used to guide trace selection.

The compiler supported the Berkeley Unix run-time environment. Care was given to the data structure layout rules to ease porting from the VAX and the Motorola 68000. Despite the unusual architecture of the Multiflow Trace, it was easier to port programs from BSD VAX or Motorola 68000 systems to the Trace than to many contemporary RISC-based systems.

*The optimizer.* The goal of the optimizer was to reduce the amount of computation the program would perform at run-time and to increase the amount of parallelism for the trace scheduler to exploit. Computation was reduced by removing redundant operations or rewriting expensive ones. Parallelism was increased by removing unnecessary control and data dependencies and by unrolling loops to expose parallelism across loop iterations. The Multiflow compiler accomplished these goals with standard Dragon-book-style optimization technology enhanced with a powerful memory-reference disambiguator.

The optimizer was designed as a set of independent, cooperating optimizations that shared a common set of data structures and analysis routines. The analysis routines computed control flow (dominators, loops) and data flow (reaching defs and uses, live variables, reaching copies). In addition, the disambiguator computed symbolic derivations of address expressions. Each optimization recorded what analysis information it needed, and what information it destroyed. The order of optimizations was quite flexible; in fact, it was controlled by a small interpreter. The order of optimization was organized around two invocations of loop unrolling, to expose the maximum amount of parallelism. The order used for full optimization was as follows.

### Basic optimizations

- Expand function entries and returns
- Find register variables
- Expand memory operations
- Eliminate common sub-expressions

Propagate copies  
 Remove dead code  
 Rename temps  
 Transform ifs into selects  
  
 Prepare for first loop unrolling  
   Generate automatic assertions  
   Move loop invariant  
   Find register memory references  
   Eliminate common sub-expressions  
   Transform ifs into selects  
   Find register expressions  
   Remove dead code  
  
 First unroll  
   Unroll and optimize loops  
   Rename temps  
   Propagate copies  
   Simplify induction variables  
   Eliminate common sub-expressions  
   Propagate copies  
   Remove dead code  
  
 Second unroll  
   Unroll and optimize loops  
   Rename temps  
  
 Prepare for Phase3  
   Expand function calls  
   Walk graph and allocate storage  
   Analyze for dead code removal  
   Remove assertions  
   Remove dead code  
   Expand remaining IL operations  
   Propagate copies  
   Remove dead code  
   Rename temporaries

Loops were unrolled by copying the loop body including the exit test. Unlike most machines, the Trace had a large amount of branch resource, and there was no advantage to removing exit branches. By leaving the branches in, the compiler avoided the short-trip count penalty caused by preconditioning. In addition, loops with data dependent loop exits that could not be preconditioned (e.g., while loops) were unrolled and optimized across iterations. For loops with constant trip-count, all but one exit test was removed, and short-trip count loops were unrolled completely.

Loops were unrolled heavily. A loop needed to be unrolled enough to expose sufficient parallelism within the loop body to enable the instruction scheduler to fully utilize the machine. Unrolling was controlled by a set of heuristics which measured the number of operations in the loop, the number of internal branches, and the number of function calls. A loop was unrolled until either the desired unroll amount was reached or one of the heuristic limits was exceeded. The default unrolling in FORTRAN for a Trace 14-wide was 16; at the highest level of optimization, the compiler unrolled by 96.

The compiler did not perform software pipelining, which would have reduced the need for such large unrollings. The large size of the Trace instruction cache limited the benefits of software pipelining.

*The back end.* The output of optimizer was a flow graph of operations lowered to machine level. The back end performed functional-unit assignment, instruction scheduling, and register allocation. The work was divided into four modules: the *trace scheduler*, which managed the flow graph and assured inter-trace correctness; the *instruction scheduler*, which scheduled each trace and assured intra-trace correctness; the *machine model*, which provided a detailed description of the machine resources; and a *disambiguator*, which performed memory-reference analysis.

*Trace scheduler.* The trace scheduler performed the following steps:

*Step 1.* The trace scheduler estimated how many times each basic block in the flow graph would be executed. The execution estimates were calculated from loop trip counts and the probabilities of conditional branches. The estimates were obtained from either a database collected during previous executions of the program, user directives, or heuristics.

*Step 2.* The trace scheduler performed the following loop until the entire flow graph has been scheduled.

- Using the execution estimates as a guide, the trace scheduler picked a trace (a sequence of basic blocks) from the flow graph. The trace scheduler first selected a seed, the yet-to-be-scheduled block with the highest execution estimate. The trace was grown forward (in the direction of the flow graph) and then backward from the seed; in each step, the trace scheduler selected a basic block that satisfied the

current trace-picking heuristic. If no basic block satisfied the current heuristic, the trace ended. Traces always ended when the trace scheduler reached an operation that was already scheduled or an operation that was already on the trace. Also, traces never crossed the back edge of a loop.

- The trace scheduler passed the trace to the instruction scheduler. The instruction scheduler returned a machine language schedule. The instruction scheduler is discussed in more detail below.
- The trace scheduler replaced the trace with the schedule in the flow graph and, if necessary, added copies of operations to compensate for code motions past basic block boundaries. Compensation code was necessary if an operation moved below a branch out of the trace or above the target of a branch into the trace.

Speculative execution, moving an operation above a branch, did not produce compensation code. This was the most common code motion in the Multiflow compiler. High priority operations from late in the trace were moved above branches and scheduled early in the trace. The instruction scheduler would perform such a move only if it was safe: An operation could not move above a branch if it wrote memory or if it set a variable that was live on the off-trace path. The hardware provided special *dismissible* operations and support for suppressing or deferring the exceptions generated by speculative operations.

*Step 3.* The trace scheduler emitted the machine language schedules in depth first order. This ordering gave the effect of profile-guided code layout, since traces were selected along the most frequently traveled paths in the program.

The trace scheduler also provided the instruction scheduler the global context for a trace. The instruction scheduler scheduled one trace at a time, but it required information about neighboring traces for correctness and to optimize the performance on inter-trace transitions. The functional-unit pipelines on the Multiflow computer used machine resources in every cycle; the trace scheduler communicated to the instruction scheduler the state of the machine pipeline on entry to the trace. Similarly, it passed information about memory references in flight, to avoid bank stalls.

The trace scheduler and the instruction scheduler also exchanged the register bindings at each trace entry and exit point, enabling incremental register allocation with a global view.

*Instruction scheduler.* The instruction scheduler transformed a trace of operations into a schedule of wide instructions. For each operation, the instruction scheduler assigned registers for the operands, assigned a functional unit for the operation, and placed the operation in a wide instruction. It used a three-step algorithm.

*Step 1.* The instruction scheduler built a data precedence graph (DPG) from the trace. Scheduling a trace rather than a basic block introduced very little complexity in the instruction scheduler. When building the DPG, edges were added to constrain some operations from moving above branches. After scheduling, the trace scheduler compensated for any global code motions performed by the instruction scheduler.

*Step 2.* The instruction scheduler walked the DPG and assigned operations to functional units and values to register banks. The scheduler made a tradeoff between parallelism and the cost of global data motion. To implement this tradeoff, the scheduler divided the DPG into components that contained a relatively small amount of parallelism and a large amount of shared data. The scheduler did a bottom-up traversal of the DPG, assigning functional units and register banks. The scheduler imposed a high cost for the use of more than one identical resource (e.g., a floating point unit) for a particular component in the DPG. This heuristic kept the members of the same component together in the same subset of the machine and achieved parallelism by spreading the components around the machine.

*Step 3.* The instruction scheduler performed list scheduling, creating the schedule and allocating registers. The scheduler created the schedule in forward order, starting with the first instruction. The scheduler maintained a list of data-ready operations and filled an instruction with the highest priority operations from the data-ready list, and then advanced to the next instruction.

On the Multiflow Trace machines, long pipelines and multiple operations per instruction made floating point registers a critical resource for generating high performance code. The instruction scheduler

performed register allocation, enabling the registers to be scheduled similarly to the other resources. List scheduling was well suited to integration with register allocation. At any point during the scheduling process, each of the registers was either occupied or available. When checking for resources to schedule a particular operation, the scheduler also included a check for a free register for operation's result. If all the resources were available to schedule the operation, the register, along with the other resources, was reserved. When there was no free register, the scheduler would either free a register or delay the current operation. Registers could be freed by spilling, by rematerialization, or by removing an earlier operation from the schedule. Heuristics for this decision were very important to performance.

The scheduler also scheduled the two-level interleaved memory system. The scheduler organized the memory references in a trace into equivalence classes based on the compiler's knowledge of their offsets relative to each other; the classes were formed by querying the disambiguator. Within an equivalence class, the scheduler understood the potential bank and card conflicts; between two different classes, nothing was known. The scheduler would batch memory references by equivalence class to avoid potential stalls. Batching groups of references increased register pressure, and heuristics were important to control this tradeoff.

*Machine model.* The Trace machines had limited hardware resource management and relied on the compiler to allocate machine resources and check for oversubscription. The compiler contained a detailed model of the machine. The model contained the set of operations and the mapping of each operation to functional units, register banks, and immediate fields. The latencies and resources required by each stage of an operation were represented. The model contained a graph of connections between machine elements and the latencies and resources required to traverse the graph. The model was organized to provide efficient access to the scheduler about the resources needed to schedule an operation.

*Disambiguator.* The compiler looked for fine-grained parallelism along a trace and within the body of an unrolled loop. The scheduler typically wanted to know if a load in the current iteration could move above a store in the previous iteration. This is a different question

than the one asked by a parallelizing compiler, which is interested in knowing if there exists a conflict between a load and a store across any of the loop iterations. Memory analysis was performed by a *disambiguator*, which tried to determine if two memory references refer to the same location or whether they refer to the same memory bank.

The disambiguator derived a symbolic equation of the address used in a memory reference; the terminals in the equation were constants and definitions (or sets of definitions) of variables in the program. Each derivation was normalized to a sum of products. To answer a question about two memory references, the disambiguator subtracted the two derivations and performed a GCD test. An assertion facility allowed the programmer pass additional information to improve accuracy.

The disambiguator also knew basic facts about how a variable was allocated and declared, and it used this information in its analysis as well. For example, a pointer cannot reference a variable whose address is never captured by the program.

*Calling sequence.* Multiflow used a pure caller-saves register partition; the values of registers were not preserved across a procedure call. Compiling for parallelism requires the use of many more registers than are used in sequential compilation, and typically register lifetimes do not cross function calls. On a wide machine, the saving and restoring of caller-saved registers could be intermixed with other operations before and after a call, and were frequently scheduled with minimal execution cost. Trace scheduling selected traces in priority order, enabling the saving and restoring of registers to be moved out of the most frequently executed paths in the program.

Multiflow used all of the registers in the first cluster for passing arguments and returning values. A total number of 53 integer values and 30 double precision floating values could be passed or returned in registers. Aggregates (e.g., C structures) could be passed and returned in registers. The large number of registers available for procedure linkage enabled a fast linkage to the 8-and 16-at-a-time math functions.

For C programs, the compiler passed arguments in both memory and registers; the called procedure read the arguments from registers, or from memory if required. The extra cost of passing arguments in both registers and memory was surprisingly small. The access

to arguments on procedure entry was often on the critical path, and the arguments could be read directly from registers. The storing of arguments at a call site was often not on the critical path, and it could be scheduled alongside other operations at little cost.

### Multiflow's Influence on the Industry

The Multiflow compiler technology was purchased by Intel, Hewlett-Packard, Digital Equipment Corporation, Fujitsu, NEC, Hughes, HAL Computer, and Silicon Graphics, as well as by several research institutions, including MIT and The University of Washington. Some purchased it for its highly capable compiler framework, which could be adapted to generate code for non-VLIW systems. Others bought it to aid in the development of VLIW architectures or to generate production compilers to accompany VLIW products. In the past two decades, several notable compiler product lines have been derived from the Multiflow compiler technology, in particular the compiler products from Intel, NEC, and Fujitsu. These compilers have received significant ongoing investment, and the original Multiflow implementation has been largely rewritten.

Multiflow was an important proof-of-concept of VLIW technology. The successful implementation of Multiflow's systems demonstrated that VLIW technology was viable, something that was not widely believed before Multiflow's engineering success. Texas Instruments had written of their skepticism before Multiflow; eventually, they were to develop VLIW processors that have had a strong position in cellular communications. VLIWs from various manufacturers are now commonplace in digital cameras, multifunction printers, network communication devices, computer graphics accelerators, televisions, digital video recorders, and other set-top boxes.

The Trace machines were a direct influence on the STMicroelectronics ST-231's architecture, developed by Fisher's team at HP. By 2010, nearly 100 million ST-231 parts were reported to be deployed in printers and video devices.

### Related Entries

- [Instruction-Level Parallelism](#)
- [Trace Scheduling](#)
- [VLIW Processors](#)

### Bibliographic Notes and Further Reading

Multiflow technology was derived from Josh Fisher's research. As a graduate student at New York University, he developed the idea of trace scheduling [4, 5]. As a professor at Yale, he extended the trace scheduling idea, designing a VLIW architecture [6] and a practical compiler [7]. The dissertations of John Ellis [3] and Alex Nicolau [13] describe the compiler in detail. Ellis's dissertation won the ACM Doctoral Dissertation Award, and his research compiler was a starting point of the work at Multiflow.

The first paper on Multiflow was published at ASPLOS in 1987[1], the same year the first Trace machines shipped. A retrospective on the never-completed 500 series is presented in [2]. A thorough description of the Multiflow compiler was published in a special issue of the Journal of Supercomputing devoted to instruction-level parallelism [11]. An experimental evaluation of the Multiflow compiler is presented in [10].

### Bibliography

1. Colwell RP, Nix RP, O'Donnell JJ, Papworth DB, Rodman PK (1988) A VLIW architecture for a trace scheduling compiler. *ACM SIGARCH Computer Architecture News* 15(5), October 1987, pp 180–192. A revised version was published in *IEEE Trans Comput* 37(8):967–979
2. Colwell RP, Hall WE, Joshi CS, Papworth DB, Rodman PK (1990) Architecture and implementation of a VLIW supercomputer. In: *Proceedings, IEEE/ACM supercomputing 90*, October 1990. IEEE Computer Society Press, Los Alamitos, pp 910–919
3. Ellis JR (1986) Bulldog: a compiler for VLIW architectures. MIT Press, Cambridge, MA. Also Ph.D. Thesis, Yale University, New Haven, Conn., February 1985
4. Fisher JA (1979) The optimization of horizontal microcode within and beyond basic blocks: An application of processor scheduling with resources. Ph.D. Thesis, New York University, New York
5. Fisher JA (1981) Trace scheduling: a technique for global microcode compaction. *IEEE Trans Comput* C-30(7):478–490
6. Fisher JA (1983) Very long instruction word architectures and the ELI-512. In: *Proceedings of the 10th annual international symposium on computer architecture*, Stockholm, Sweden. ACM, New York, pp 140–150
7. Fisher JA, Ellis JR, Ruttenberg JC, Nicolau A (1984) Parallel processing: a smart compiler and a dumb machine. In: *Proceedings of ACM SIGPLAN '84 symposium compiler construction*, Montreal, June 1984. ACM, New York, pp 37–47
8. Fisher JA, Faraboschi P, Young C (2004) Embedded computing a VLIW approach to architecture, compilers, and tools. Morang Kaufman, San Francisco

9. Freudenberger S, Ruttenberg J (1992) J phase ordering of register allocation and instruction scheduling. In: Giergerich R, Graham SL (eds) Code generation – concepts, tools, techniques. Springer, London, pp 146–172
10. Freudenberger SM, Gross TR, Lowney PG (1994) Avoidance and suppression of compensation code in a trace scheduling compiler. *Trans Prog Lang Syst* 16(4):1156–1214
11. Lowney PG, Freudenberger S, Karzes T, Lichtenstein W, Nix R, O'Donnell J, Ruttenberg J (1993) The multiflow trace scheduling compiler. *J Supercomput* 7(1–2):51–142
12. Nicolau A, Fisher J (1981) Using an oracle to measure parallelism in single instruction stream programs. The 14th annual microprogramming workshop, October 1981. IEEE, New York, pp 171–182
13. Nicolau A (1984) Parallelism, memory anti-aliasing and correctness for trace scheduling compilers. Ph.D. Thesis, Yale University, New Haven, CT

## Multifrontal Method

PATRICK AMESTOY<sup>1</sup>, ALFREDO BUTTARI<sup>1</sup>, IAIN DUFF<sup>2</sup>,  
ABDOU GUERMOUCHE<sup>3</sup>, JEAN-YVES L'EXCELLENT<sup>4</sup>,  
BORA UÇAR<sup>4</sup>

<sup>1</sup>Université de Toulouse ENSEEIHT-IRIT, Toulouse cedex 7, France

<sup>2</sup>Science & Technology Facilities Council, Didcot, Oxfordshire, UK

<sup>3</sup>Université de Bordeaux, Talence, France

<sup>4</sup>ENS Lyon, Lyon, France

## Synonyms

Linear equations solvers

## Definition

The multifrontal method is a direct method for solving systems of linear equations  $Ax = b$ , when  $A$  is a sparse matrix and  $x$  and  $b$  are vectors or matrices. The multifrontal method organizes the operations that take place during the factorization of sparse matrices in such a way that the entire factorization is performed through partial factorizations of a sequence of dense and small submatrices. It is guided by a tree that represents the dependencies between those partial factorizations. In the following, the multifrontal method is formulated first for finite-element analysis and later generalized to assembled sparse matrices.

## The Multifrontal Method

The multifrontal method is a direct method so that if the aim is to solve the equations

$$Ax = b, \quad (1)$$

where  $A$  is sparse, then this is done by performing the matrix factorization

$$PAQ = LU, \quad (2)$$

where  $P$  and  $Q$  are permutation matrices,  $L$  is a lower triangular matrix, and  $U$  is an upper triangular matrix. The solution to the Eq. 1 is then obtained through a forward elimination step

$$Ly = Pb,$$

followed by the backsubstitution

$$UQ^T x = y.$$

If the matrix  $A$  is symmetric, then the factorization

$$PAP^T = LDL^T \quad (3)$$

is used, where  $D$  is a block diagonal matrix with blocks of order 1 or 2 on the diagonal. The blocks of order 2 are required for stability when factorizing symmetric indefinite matrices. In the positive definite case,  $D$  can be diagonal and the Cholesky factorization

$$PAP^T = L_1 L_1^T$$

can be used where the diagonal entries of  $L_1$  are the square root of the entries in  $D$ . However, the Cholesky factorization is usually not used even in the positive definite case. This is more to avoid problems when the matrix is close to indefinite rather than to avoid taking square roots.

As mentioned later, the multifrontal method can also be used to perform a matrix factorization as a product of an orthogonal and upper triangular matrix.

## Finite-Element Analysis-Based Formulation

The multifrontal method can be regarded as a generalization of the frontal method. The frontal method was developed for the solution of sparse linear systems arising from finite-element problems. In such problems, the system matrix is defined as the sum

$$A = \sum_{\ell} A^{[\ell]} \quad (4)$$

of elemental matrices  $A^{[\ell]}$  that have nonzeros only at the intersection of the rows and columns corresponding to the variables associated with the element  $\ell$ . It is normal

to store the matrices  $A^{[\ell]}$  as small dense matrices whose rows and columns are indexed by the corresponding rows and columns of the matrix  $A$ . Thus, the summation in Eq. 4 should be thought of as an “expanded sum.” The operation in (4) is commonly referred to as the “assembly” and involves elementary summations

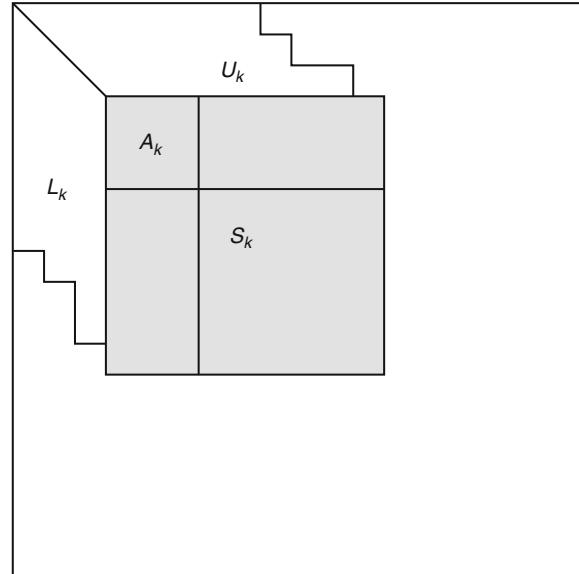
$$a_{ij} = a_{ij} + a_{ij}^{[\ell]}. \quad (5)$$

A value  $a_{ij}$  is said to be *fully summed* (or *fully assembled*) if all the operations (5) have been performed, and a variable is said to be fully summed if all the values in its row and its column are fully summed. The basic Gaussian elimination step

$$a_{ij} = a_{ij} - \frac{a_{ik}a_{kj}}{a_{kk}} \quad (6)$$

only requires that variable  $k$  is fully summed. The frontal method proceeds by successive steps where, at each step, a new element is assembled and variables are eliminated as soon as they become fully assembled. The main characteristic of the frontal method is that, at each step, all the operations are performed on a dense matrix, called a *frontal matrix*, whose rows and columns correspond to variables in the elements that have been assembled (commonly referred to as active variables). There are several benefits from using this approach. First, the elimination operations on a frontal matrix can be performed using efficient dense matrix kernels (Level-3 BLAS subroutines). Second, the stability of the factorization can be improved by applying pivoting techniques within the fully assembled variables of a frontal matrix. Third, the memory required for each step of the frontal method need only be enough to hold the frontal matrix, lending the method gracefully to limited memory environments. Fourth, it is relatively easy to use out-of-core techniques so that neither the factors nor the matrix need be held in memory.

Assuming that  $A$  is decomposed using an  $LU$  factorization (as in Eq. 2), Fig. 1 illustrates one step of the frontal method, as element  $k$  is assembled. The dark-shaded area represents the frontal matrix whose variables are split into assembled (in  $A_k$ ) and non-assembled variables. As the fully assembled variables are eliminated, the corresponding rows and columns end up in the  $U$  and the  $L$  factors, respectively, to form  $U_{k+1}$  and  $L_{k+1}$ . The next element  $k+1$  is then assembled with  $S_k$  (commonly referred to as a *contribution block*) which



**Multifrontal Method. Fig. 1** One step of the frontal method

corresponds to the Schur complement resulting from the elimination of the fully assembled variables.

The name “frontal” stems from the observation that the set of active variables forms a front that sweeps the problem domain element by element. Figure 2b shows the assembly and elimination process of the frontal method when applied to the regular finite-element problem depicted in Fig. 2a, with the element ordering  $a, b, c, d$ .

The  $A^{[\ell]}$  submatrix associated with an element  $\ell$  may contain fully assembled variables before being summed as in Eq. 4; this corresponds to the case where a variable only appears in one element, for example, a variable corresponding to an interior node of a finite element. In this case, it is possible to eliminate the fully assembled variables before assembling the element, the  $A^{[\ell]}$  term in Eq. 4 can be replaced by the resulting Schur complement. This is known as *static condensation*.

The multifrontal method takes this observation to a higher level in order to achieve a better exploitation of the sparsity structure of the matrix. Building upon the idea that variables that are internal to different subdomains (by which we mean elements or groups of elements) can be assembled and eliminated independently, this method basically amounts to having multiple fronts so that the set of variables that are fully assembled within a front does not overlap with the set

of active variables in any other front. In other words, if the frontal method corresponds to the left-to-right evaluation of the summation (4), where the order of the elements is critical for the performance, the multifrontal method corresponds to any valid bracketing of the same summation.

The multifrontal method is guided by a tree structure commonly referred to as an *assembly tree*, which expresses the way elements are assembled into the fronts. Considering the simple finite-elements problem in Fig. 2a, the assembly tree corresponding to the bracketing

$$(((A^{[a]} + A^{[b]}) + A^{[c]}) + A^{[d]})$$

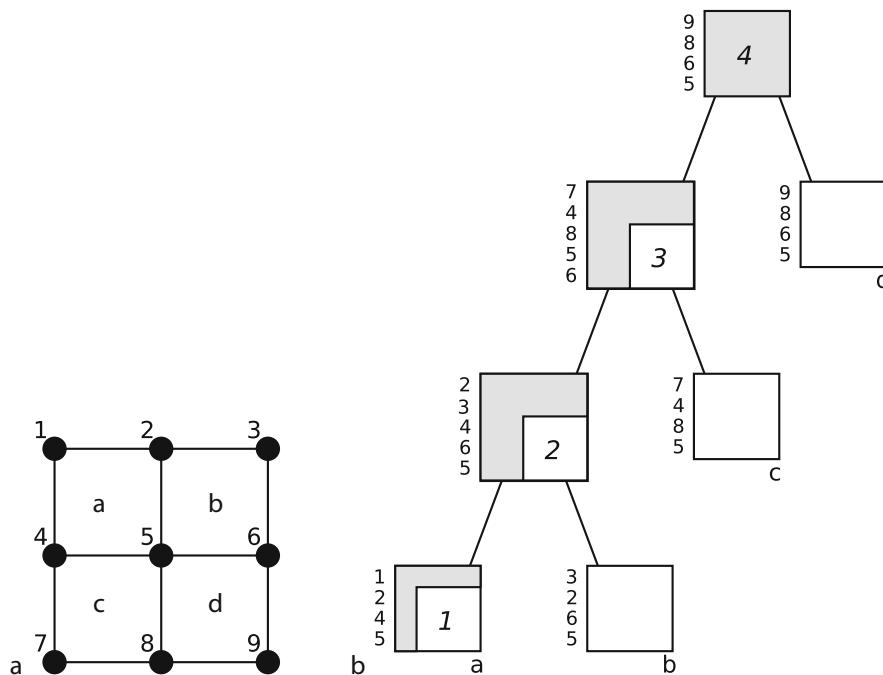
is shown in Fig. 3a; note that in this case the multifrontal method is equivalent to a frontal method (as shown in Fig. 2b) with static condensation. However, thanks to the higher flexibility of the multifrontal method, different bracketing schemes can be used to improve the memory required and the execution time. For example, if a nested dissection ordering is used on the simple problem in Fig. 2a the bracketing

$$((A^{[a]} + A^{[b]}) + (A^{[c]} + A^{[d]}))$$

would result, and thus the assembly tree in Fig. 3b would be obtained.

### Assembled Matrix Point of View

The multifrontal method can be used for factorizing quite general matrices, but first consider the case when  $A$  is symmetric and numerical pivoting is not required (e.g., when  $A$  is positive definite). Assume  $A$  is of order  $n$  and a sparsity preserving ordering is known. Consider a tree data structure on  $n$  nodes, each node corresponding to a column of  $L$  (say from the factorization (2)), built by setting a parent pointer for each node  $j$  as follows:  $p$  is the parent of  $j$  if and only if  $p = \min\{i > j : \ell_{ij} \neq 0\}$ . If no such  $p$  exists for a node  $j$ , then  $j$  is a root. The resulting tree structure is called the *elimination tree*. If  $A$  is irreducible, then the data structure is indeed a tree, otherwise it is a forest. A frontal matrix can be associated with each node of the elimination tree. As in the finite-element analysis case, the frontal matrices are permuted into a  $2 \times 2$  block structure where all variables from the (1,1)-blocks can be eliminated (i.e., they are *fully summed*) but the Schur complements (formed by the elimination of these fully summed entries on the



**Multifrontal Method. Fig. 2** The assembly and elimination process in the frontal method on a regular grid with four elements

(2,2)-block) cannot be eliminated until later in the factorization. The Schur complement computed at a node corresponds to the *contribution block* introduced in the finite-element analysis section. The variable at a node  $j$  becomes fully summed after all of the descendants of the node in the elimination tree have been eliminated.

By definition of the elimination tree, however, there is only one fully assembled variable in each frontal matrix. This prevents the use of efficient Level-3 BLAS routines for performing the elimination and update of the Schur complement and may result in fronts whose size is too small to achieve a good exploitation of the memory hierarchy available in modern processors. It is thus advantageous to combine or amalgamate nodes of the elimination tree. In the case where two or more columns of the  $L$  factor have the same structure below the diagonal (modulo a diagonal block), the corresponding variables can be amalgamated and thus eliminated within a single frontal matrix without introducing any additional fill-in. In the general case where columns of the  $L$  factor do not have the same structure, it is possible to perform amalgamation at the cost of additional fill-in; threshold-based strategies can be used in this case to perform amalgamation so that the cost of the introduced fill-in is overcome by the advantages of more efficient computations.

As a result of the amalgamation procedure, the elimination tree is transformed into an *assembly tree* whose nodes define the steps of the multifrontal method. At each step, values of the problem matrix are assembled together with the contribution blocks from the children nodes into a frontal matrix, the fully summed variables are eliminated and the corresponding Schur complement is formed in order to be assembled at the parent node (if any). The “Achilles heel” of sparse direct methods applied to general systems is that indirect addressing will be required. The multifrontal method cannot thus avoid this but the indirect addressing only occurs in the assemblies while all the arithmetic is performed using direct addressing as in the dense case, as in the case of frontal methods.

## Discussion

### Phases of Solution

Similar to other sparse direct methods, the solution of a linear system by means of a multifrontal method is

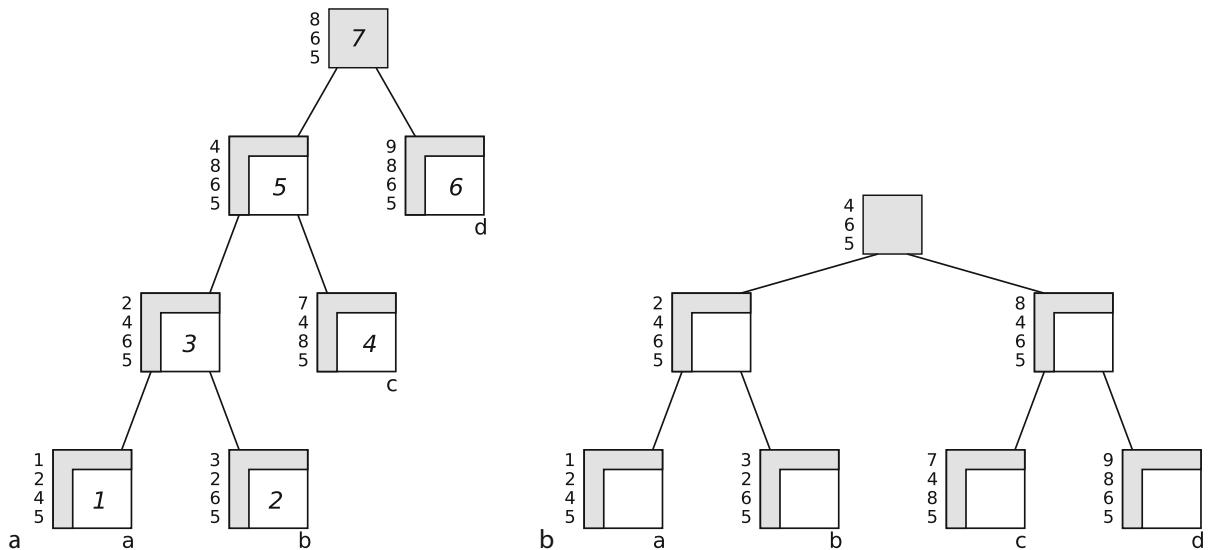
commonly achieved through three main phases: analysis, factorization, and solve. This subdivision enables the analysis and factorization phases to be performed only once when several linear systems with the same coefficients and differing right-hand side vectors are to be solved successively; similarly if several matrices with the same sparsity pattern are to be factorized, the analysis phase can be performed only once.

### Analysis

This step mostly consists of symbolic preprocessing operations prior to the actual factorization. These operations include the computation of a fill reducing pivotal order and a symbolic elimination process to determine the size of the factors and the frontal matrices. The ordering and matrix pattern will define a unique elimination tree but the important feature of the tree is that it only defines a partial ordering for the subsequent matrix factorization and this freedom can be used to great effect to reduce memory costs and to enable the efficient exploitation of parallelism.

### Factorization

The actual numerical factorization takes place in this step. From the memory point of view, implementations of the multifrontal factorization commonly use three “logical” areas of storage: one for the factors, one to stack the contribution blocks, and one for the current frontal matrix. The idea of using a stack stems very naturally from the fact that, in a sequential environment, the nodes of the elimination tree can be processed using a *postorder* (all nodes within any given subtree are numbered consecutively). During the tree traversal, the storage required by the factors always grows while the stack memory (containing the contribution blocks) can both increase and decrease. When the partial factorization of a frontal matrix is completed, a contribution block is pushed on to the stack, increasing the stack size; on the other hand, when a frontal matrix is formed and assembled, the contribution blocks of its children are popped out of the stack, decreasing the stack size. This pattern of memory use permits a natural extension to out-of-core execution. First, the computed factors of the frontal matrices can be moved to secondary storage after they are computed (they are not needed until the solve phase). Second, the stack memory has a natural locality, thanks to the postorder.



**Multifrontal Method. Fig. 3** Finite-element problem and examples of associated assembly trees. Fully assembled variables are shown with a dark-shaded area within each frontal matrix

Because the order in which the tree is traversed can have a significant impact on both the number of contribution blocks stored simultaneously on the stack and the working memory, great care must be taken to sort the siblings at each level of the tree. The rule of thumb followed in performing this operation can be illustrated in Fig. 3a (where the memory associated with factors that have already been computed is not taken into account). If the nodes of the tree are processed in the order 1-2-3-4-5-6-7, at most two contribution blocks and one frontal matrix must be stored. If, on the other hand, the nodes are processed in the order 6-4-2-1-3-5-7, then the peak memory requirement is for four contribution blocks and one frontal matrix.

During the factorization phase, pivoting may be performed to improve numerical stability. The normal numerical control for pivoting is the same for multifrontal methods as for most sparse direct methods, that is threshold pivoting. In the general case, the entry  $a_{ij}$  can be used as a pivot only if

$$\frac{|a_{ij}|}{\max_k |a_{kj}|} \geq u,$$

where  $u$  is a threshold parameter ( $0 < u \leq 1.0$ ). Clearly, in a frontal matrix, the pivot can only be chosen from the fully summed block so it is possible that not all

the fully summed block (the (1,1) block of the frontal matrix) can be eliminated at a particular node because of relatively large entries in the (2,1) block. The rows and columns corresponding to the fully summed variables which remain uneliminated are then added to the contribution block and assembled in the frontal matrix of the parent node. This gives rise to *delayed pivoting*. The elimination of delayed pivots may become possible at the parent node because the large entry could now be in the (1,1) block of the new frontal matrix, for example. It must be noted, however, that in case of delayed pivots, the size of the frontal matrix at the parent node is increased as well as that of the contribution block. For such a reason, the use of delayed pivoting requires complex dynamic data structures. In parallel implementations, dynamic scheduling strategies may be necessary to automatically adapt to such situations and this is done in the code ►MUMPS. A more recent alternative is to avoid delayed pivoting by using *static pivoting* where a pivot is still chosen even if it does not satisfy the threshold criterion but may be augmented in value to avoid instability. In this case, which is available in MUMPS as well as the other codes (SuperLU and PARDISO) described in this encyclopedia, iterative refinement or perhaps more sophisticated iterative techniques must be used to obtain an accurate solution.

## Solve

During this step, the triangular factors computed at factorization time are used together with the right-hand side vector or matrix to compute the solution of the problem by means of forward and backward substitutions. More precisely, the forward step uses a bottom-up traversal of the tree similar to the factorization, and the backward step uses a top-down traversal of the tree: starting from the root, each node computes the components of the solution corresponding to its fully summed variables and informs its children of the already computed components of the solution.

## Unsymmetric Matrices

The multifrontal method can be applied to unsymmetric matrices as well. If the matrix  $A$  is pattern symmetric, or nearly so (in which case the pattern of  $M = |A| + |A|^T$  can be used), then  $A$  can be treated as before. By generalizing the partial factorization of the frontal matrices to the rectangular case, special methods for unsymmetric matrices can be developed. This approach necessitates the use of directed acyclic graphs to represent the computational dependencies. Because of asymmetry, it can be shown that in general the compact representation of this acyclic graph is not a tree. It can still be beneficial to use the simplicity of the elimination tree structure (instead of the directed acyclic graphs) in the unsymmetric case, with special attention in order to detect and make use of asymmetry in the frontal factors.

Another approach is to generalize the notion of the elimination tree to unsymmetric matrices. This generalization provides a general framework to describe all unsymmetric multifrontal approaches.

The multifrontal method also extends to sparse QR factorization, typically used for least-square problems where the objective is to find an  $x$  that minimizes  $\|Ax - b\|$ .

## Parallel Execution

The multifrontal method presents many features that make it suitable for multiprocessor environments. Parallelism can be exploited at two different levels. First, two disjoint subtrees of the elimination tree can be processed in parallel, as there is no computational dependency between such subtrees. Second, the frontal matrices that are large enough can be factorized in

parallel using algorithms for dense linear algebra operations – such frontal matrices occur usually toward the end of the factorization process where nodes of the elimination tree that are close to the root are processed.

Effective scheduling and mapping strategies are necessary for exploiting the two sources of parallelism mentioned above. In a distributed-memory environment (see MUMPS), much attention has been paid to static mapping methods to map the tasks associated with the nodes of the tree to the processes. The *subtree-to-subcube* mapping is used to reduce the communication overhead in parallel sparse Cholesky factorization on hypercubes. This mapping mostly addresses parallelization of the factorization of matrices arising from a nested dissection-based ordering of regular meshes. The basic idea of the algorithm is to build a layer of  $p$  subtrees (where  $p$  is the number of processors) and then to assign to each processor  $p_i$  a subtree  $T_i$ . The nodes of the upper parts of the tree are then mapped using a simple rule: when two subtrees merge together into their parent subtree, their set of processors are merged and assigned to the root of that parent subtree. The method reduces the communication cost but can lead to load imbalance on irregular assembly trees. A bin-packing-based approach has then been introduced to improve the load balance on irregular trees. Given an arbitrary tree, the improvement is achieved by finding the smallest set of subtrees which can be partitioned among the processors while attaining a given load balance threshold. In this scheme, starting from the root node, the assembly tree is explored until a layer of subtrees, whose corresponding mapping on the processors produce a “good” balance, is obtained (the balance criterion is computed using a first-fit decreasing bin-packing heuristic). Once the mapping of the layer of subtrees is done, all the processors are assigned to each remaining node (i.e., the nodes of the upper parts of the tree). This method takes into account the workload to distribute processors to subtrees. However, it does not minimize communication in the sense that all processors are assigned to all the nodes of the upper parts of the tree. The *proportional mapping* algorithm improves upon the two above algorithms. It uses the same “local” mapping of the processors as that produced by the *subtree-to-subcube* algorithm with the difference that it takes into account workload information to build the layer of subtrees. To be more precise, starting from the root node to which all the processors

are assigned, the algorithm splits the set of processors recursively among the branches of the tree according to their relative workload until all the processors are assigned to at least one subtree. This algorithm is characterized by both a good workload balance and a good reduction in communication.

In a parallel computing context, memory use is much more difficult to control and limit for two related reasons. First, it may not be possible to guarantee a postorder traversal of the tree while exploiting the tree parallelism since processes may work on different branches of the tree at the same time. Thus the use of a simple stack to hold the contribution blocks as in the sequential case is not possible. Second, as many processes are active during the factorization, the global memory requirement may increase compared to a sequential execution. The memory per process should decrease but the overall memory requirement may increase.

The solve step can also be executed in parallel by following the same computational pattern of the factorization phase, that is, the pattern induced by the elimination tree.

## Bibliographic Notes and Further Reading

Based on techniques developed for finite-element analysis by Irons [15] (and a technical report by Speelpenning dated 1978), Duff and Reid proposed the multifrontal method for symmetric [7] and unsymmetric [8] systems of equations. Eisenstat et al. [10] proposed another method with the same basic features as the multifrontal method. More recently, Eisenstat and Liu [9] have provided a new generalization of elimination trees for unsymmetric matrices using the notion of paths instead of edges to define a parent node in the tree. This generalized elimination tree leads to a general framework to describe how all existing approaches take the asymmetry into account. This includes the work of: Duff and Reid [8] and Amestoy, Duff, and l'Excellent [2] who apply the multifrontal method to unsymmetric matrices with symmetrized pattern; Davis and Duff [6] and Gupta [14] who do not use the assembly tree but use a directed acyclic graph to represent the computational dependencies; Amestoy and Puglisi [5] who work on a “symmetrized” assembly tree by making use of the structural asymmetry of the frontal matrices during the factorization phase.

Liu [17] provides an overview of the multifrontal method for symmetric positive definite matrices.

Liu [16] and Guermouche et al. [13] have explored the impact of the tree traversal on the memory behavior and proposed tree traversals that minimize the storage requirements of the method.

The mapping algorithms discussed in Sect. Parallel Execution are by George et al. [12], Geist and Ng [11], and Pothen and Sun [19]. Amestoy et al. [4] generalize and improve the heuristics in [11, 19] by taking memory scalability issues into account and by incorporating dynamic load balancing decisions for which some preprocessing is done in the analysis phase.

Memory management issues in the implementation of the multifrontal method are discussed by Amestoy and Duff [1]. Matstoms [18] and Amestoy, Duff, and Puglisi [3] discuss the multifrontal QR method in a multiprocessor environment.

## Related Entries

- [BLAS \(Basic Linear Algebra Subprograms\)](#)
- [Dense Linear System Solvers](#)
- [Mumps](#)
- [Reordering](#)

## Bibliography

1. Amestoy PR, Duff IS (1993) Memory management issues in sparse multifrontal methods on multiprocessors. *Int J Supercomput Appl* 7:64–82
2. Amestoy PR, Duff IS, L'Excellent J-Y (2000) Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput Methods Appl Mech Eng* 184:501–520
3. Amestoy PR, Duff IS, Puglisi C (1996) Multifrontal QR factorization in a multiprocessor environment. *Numer Linear Algebra Appl* 3(4):275–300
4. Amestoy PR, Guermouche A, L'Excellent J-Y, Pralet S (2006) Hybrid scheduling for the parallel solution of linear systems. *Parallel Comput* 32:136–156
5. Amestoy PR, Puglisi C (2002) An unsymmetrized multifrontal LU factorization. *SIAM J Matrix Anal Appl* 24:553–569
6. Davis TA, Duff IS (1997) An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM J Matrix Anal Appl* 18:140–158
7. Duff IS, Reid JK (1983) The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans Math Softw* 9:302–325
8. Duff IS, Reid JK (1984) The multifrontal solution of unsymmetric sets of linear equations. *SIAM J Sci Stat Comput* 5:633–641
9. Eisenstat SC, Liu JWH (2005) The theory of elimination trees for sparse unsymmetric matrices. *SIAM J Matrix Anal Appl* 26: 686–705

10. Eisenstat SC, Schultz MH, Sherman AH (1976) Applications of an element model for Gaussian elimination. In: Bunch JR, Rose DJ (eds) Sparse matrix computations. Academic, New York/London, pp 85–96
11. Geist GA, Ng EG (1989) Task scheduling for parallel sparse Cholesky factorization. *Int J Parallel Program* 18:291–314
12. George A, Liu JWH, Ng E (1989) Communication results for parallel sparse Cholesky factorization on a hypercube. *Parallel Comput* 10:287–298
13. Guermouche A, L'Excellent J-Y (2006) Constructing memory-minimizing schedules for multifrontal methods. *ACM Trans Math Softw* 32:17–32
14. Gupta A (2002) Recent advances in direct methods for solving unsymmetric sparse systems of linear equations. *ACM Trans Math Softw* 28:301–324
15. Irons BM (1970) A frontal solution program for finite-element analysis. *Int J Numer Methods Eng* 2:5–32
16. Liu JWH (1986) On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Trans Math Softw* 12:249–264
17. Liu JWH (1992) The multifrontal method for sparse matrix solution: theory and practice. *SIAM Rev* 34:82–109
18. Matstoms P (1995) Parallel sparse QR factorization on shared memory architectures. *Parallel Comput* 21:473–486
19. Pothen A, Sun C (1993) A mapping algorithm for parallel sparse Cholesky factorization. *SIAM J Sci Comput* 14:1253–1257

## Multi-Level Transactions

► [Transactions, Nested](#)

## Multilisp

ROBERT H. HALSTEAD, JR  
Curl Inc., Cambridge, MA, USA

### Synonyms

[Mul-T](#); [MultiScheme](#)

### Definition

Multilisp is a parallel programming language, focused particularly on parallel symbolic computing, developed as a research project at MIT during the 1980's. The Multilisp language design and implementation projects served as a vehicle for exploring various ideas related to concurrency constructs, scheduling, parallel garbage

collection, speculative computation, and debugging tools for parallel programs.

## Discussion

### Introduction

The 1980s saw much experimentation with parallel programming models, as supercomputer performance and performance/price ratios both reached new heights amid a growing variety of SIMD and MIMD parallel machines, using both shared- and distributed-memory architectures, from various vendors. It became increasingly apparent that scaling laws for hardware would eventually favor parallel computing as the most cost-effective way to achieve high performance.

A growing recognition of the challenges of programming parallel machines also led to many parallel programming research projects targeting different hardware architectures and different application areas. Two key characteristics of an application that influence the choice of a parallel computing approach are the amount of parallelism available in the application (often related to the size of the data set) and the regularity of the available parallelism: whether the set of operations to be performed, and the dependences between them, have a simple structure largely independent of the data values, or are complex and data-dependent. “Regular” applications include various signal-processing problems as well as partial differential equation solutions over regular grids, while “irregular” applications include sparse matrix computations, mesh generation, computer algebra, discrete event simulation, and various heuristic search and optimization applications. Many irregular applications are termed “symbolic” because they emphasize navigating and creating complex data structures over performing arithmetic.

Some parallel programming projects focused on massively parallel computing for large numeric problems with a fairly regular structure, often a good match for vector or SIMD architectures. Multilisp was among a group of other projects that focused on exploiting parallelism in problems with a less regular structure. Multilisp itself was aimed at shared-memory MIMD architectures and oriented especially toward symbolic applications. The Multilisp team believed that the irregularity of these applications makes it too difficult to

preschedule operations or predetermine a good distribution of application data across the nodes of a distributed-memory machine, so a shared-memory MIMD architecture would be the best implementation target.

Most applications that were investigated for Multilisp offered moderate opportunities for parallel speedup, more in the tens than in the thousands. Such speedups were often not exciting enough in the era of the 1980s and 1990s, when sequential processor speeds were doubling every couple of years. In the present environment, clock speeds are increasing only slowly and multicore processors are the new vehicle for increasing computation throughput, so there may be a renaissance of serious interest in concepts first developed for Multilisp and its contemporary projects. Multicore processors are precisely the shared-memory MIMD model at which Multilisp and similar projects were targeted.

## Multilisp Semantics

The design of Multilisp is based on several major hypotheses:

1. The best route to a high degree of parallelism is to identify a very large number of concurrent computations and then inexpensively and efficiently schedule them across the available processors.
2. Functional programming is good for parallel computing because it eliminates bugs caused by unintended nondeterminacy, but controlled nondeterminacy is desirable in some applications.
3. For symbolic computations that work by traversing and building linked data structures, synchronization closely tied to the data structures is often the most natural.
4. Parallel programming is hard enough, so a parallel programming language implementation should automate clerical tasks that we know how to automate, notably heap storage allocation and garbage collection.

Point (1) creates a strong bias in favor of simple, lightweight concurrency and synchronization constructs. Point (2) drove the Multilisp team toward a design strongly influenced by the Scheme language [10], which includes side-effecting primitives but can be used very naturally in a “mostly functional” programming style, where side-effecting operations are used only at

the comparatively infrequent points where an imperative state change on an object is exactly what is needed. Point (4) seems uncontroversial from today’s vantage point, after the rise of Java in the 1990s popularized the notion that serious software can be written on a platform with garbage-collected heap storage, but in the 1980s, there was still great debate about whether garbage-collected systems like Lisp and Scheme could be used for “real work” like C and C++ could.

Scheme, a member of the Lisp family of languages, was an attractive starting point not only because it is consistent with point (2) above, but also because it is built on garbage-collected heap storage (point 4) and is widely used for building symbolic applications. Multilisp implementations differed somewhat in how faithfully they adhered to the details of the Scheme definition, but all of them were philosophically quite close to Scheme.

## Basic Concurrency Constructs

On top of the Scheme foundation, Multilisp adds two basic concurrency constructs: `pcall` and `future`, described in detail by Halstead [3]. `pcall` provides classic fork-join concurrency while `future` provides a form of data-flow concurrency. A Multilisp expression such as

$$(pcall F A B C \dots)$$

allows concurrent evaluation of the expressions  $F$ ,  $A$ ,  $B$ ,  $C$ , .... After waiting for all of these evaluations to return a value, the procedure value of the expression  $F$  is applied to the argument values resulting from evaluating  $A$ ,  $B$ ,  $C$ , .... The result of evaluating the `pcall` expression is thus equivalent to that of the sequential procedure call

$$(F A B C \dots)$$

except that the expressions  $F$ ,  $A$ ,  $B$ ,  $C$ , ... are evaluated concurrently.

`future` is a more radical construct that permits concurrency structures not possible with simple fork-join concurrency. A Multilisp expression

$$(future X)$$

permits a task to be spawned to evaluate the expression  $X$ , but immediately returns a placeholder for the result of that evaluation. When the evaluation of  $X$  yields a value, the `future` object essentially becomes that value. The `future` is said to *resolve* to the value of  $X$ .

The evaluation of  $X$  can proceed concurrently with the use of the placeholder value, until the point where the user actually needs some information about the value. An operation that actually needs information about a value is said to *touch* the value. Such an operation will have to block until the future resolves to a value, but especially in symbolic applications, often quite some time passes while a value is passed into or out of procedure calls, or is built into data structures and then retrieved, before the value is actually touched. Since the point where a value is touched can be remote in the program code, as well as in time, from the point where an evaluation was spawned, the `future` construct enables rich and complex concurrency structures to be easily specified that could not be specified using fork-join concurrency alone, fulfilling point (3) in the above list of design goals (closely linking synchronization with data structures). The `future` concept has its origins in the Actor systems of Baker and Hewitt [1].

Both `pcall` and `future` have the property that in a fully functional program (no side effects), inserting these constructs does not change the meaning of the program, so a programmer can feel free to insert them in whatever way best balances the exposure of concurrency with the concurrency management overhead costs.

In addition to the `pcall` and `future` constructs, which combine task spawning and synchronization, Multilisp also includes `replace` and `replace-if-eq` operations, similar to the classic atomic swap and compare-and-swap instructions, as building blocks for higher-level synchronization operators [3].

## Speculative Computing and Sponsors

The simplest use of the Multilisp constructs is just to use the concurrency primitives to relax the precedence constraints in a sequential algorithm, enabling the exact same set of operations to be performed with some parallelism on a parallel machine. A more aggressive approach is to use parallelism to speculatively launch new lines of inquiry, whose results may or may not be used depending on the results of other computations. Opportunities for such *speculative computing* are especially prevalent in search problems.

Speculative computing brings with it some new challenges. Usually, some speculative tasks are consid-

ered more valuable than others, and it is important to ensure that low-value speculative tasks do not consume resources to the extent of starving out higher-value speculative tasks or “mandatory tasks” that are not speculative. Also, estimates of the value of speculative tasks can change as a computation progresses and more information is learned about the problem being solved. An important special case of this is when the computation performed by a speculative task becomes moot because of information developed by other ongoing computations. Such *irrelevant* speculative tasks should be terminated gracefully so they will no longer consume resources. Conversely, a task initially thought to be speculative may become recognized as mandatory and deserving of the highest priority.

Managing speculation in a language like Multilisp is complex, however, because `future` can induce dependence relationships within a computation that are much more varied and tangled than the dependences that result from simple fork/join concurrency. This challenge can be addressed by introducing a higher-level construct to represent the relationships between various speculatively launched subcomputations of an overall computation. Osborne [14] investigated a *sponsor* model inspired by earlier work by Kornfeld and Hewitt [11]. In this model, each computational task is associated with one or more *sponsors* that represent the reason, or goal, for performing the computation and provide a place to store and update information about the urgency or importance of that goal.

Sponsors provide a natural framework for solving various problems with priority management. For example, a task that is blocked waiting for a future can add its own sponsor to the sponsorship of the task that is computing a value for the future. Similarly, a task waiting to enter a critical region can temporarily add its sponsor to the sponsorship of a task currently in the critical region. Both of these strategies prevent priority inversions (where higher-priority tasks are blocked by lower-priority tasks) by ensuring that any tasks whose results are awaited get the same claim on resources as the tasks that are waiting. Speculative subcomputations within a computation that is itself speculative can be represented using a tree structure of sponsors, where some sponsors are subordinated to other sponsors. Finally, a sponsor can be declared irrelevant if its associated speculative task has become moot, and this provides a natural way

to know which tasks should no longer receive execution resources.

While considerable progress was made in developing the capabilities of the sponsor model, the efficient implementation of sponsor-based speculative scheduling on a parallel machine remains an underexplored subject. However, the sponsor model seems useful not just for speculative computing but also as a model for “system programming” on a parallel machine, where multiple computational threads that share the machine’s resources execute on behalf of different users or different computations initiated by the same user, who may change his/her mind from time to time about the priority of the different computations or may even decide to abandon some of them completely.

### Futures and Continuations

The Multilisp project provided a framework for investigating various higher-level questions about how to carry familiar sequential programming capabilities into the world of parallel programming. One challenging area of investigation was how to reconcile the `future` construct with the “first-class continuation” mechanism embodied in Scheme’s `call/cc` construct [10]. The name of this construct is a contraction of “call with current continuation.” An expression such as `(call/cc f)` captures the continuation to which the value of this expression will be returned, reifies it as a procedure, and passes the resulting procedure as an argument to `f`. Subsequent computations, within `f` or elsewhere, can then cause a value `V` to be returned from the `call/cc` expression by passing `V` as an argument to the continuation procedure.

Many useful program structures can be built using `call/cc`, but it can also be used to produce various paradoxical behaviors, such as procedure calls that return more than once, possibly returning different values each time. Such behaviors conflict with the restriction that a `future` can only resolve once, to one value. Halstead [5] reviews various ideas for dealing with these conflicts, but in the end there was no definitive resolution.

### Errors and Exceptions

The handling of errors and exceptions is another challenge: what should happen if the computation `X` in `(future X)` ultimately attempts a division by zero, an

out-of-bounds array access, or even explicitly raises an exception? Modern languages for sequential programming provide mechanisms for catching such exceptions so execution can proceed down another path that deals with the condition. Typically these mechanisms work by unwinding the call stack back to a caller that catches the exception and specifies an action to take in response. With `future`, this idea is problematic because a caller that contained the `(future X)` expression may already have returned before the exception was raised.

Two solutions to this problem were investigated. One solution, described by Halstead and Loaiza [8], is for the exception in `X` to cause the `future` to resolve to an *exception value*. When an operation touches an exception value, the exception is raised again in that operation. Thus, exceptions propagate back through the data-flow structure of the program rather than through the control structure. An alternative possibility, outlined by Halstead [5] but never fully worked out, would use continuations (for example via a specified “error continuation” which would be called to signal an exception) in combination with the sponsor model (for abandoning or suspending computations rendered moot by the raising of the exception).

M

### Implementations

Over the course of the Multilisp project, three implementations were built. The first implementation, Concert Multilisp [3], ran on the Concert multiprocessor at MIT. While Concert Multilisp was a versatile testbed for many implementation ideas, it was based on a bytecode interpreter and therefore was not fast in absolute terms. The second implementation, MultiScheme [13], was based on the MIT Scheme system and ran on the BBN Butterfly multiprocessor. It also used a bytecode interpreter. The third implementation, Mul-T [12], was based on the T system from Yale and used T’s ORBIT compiler to generate native machine code. Mul-T ran on an Encore Multimax multiprocessor.

### Storage Management

Concert Multilisp included a copying, incremental garbage collector intended to allow as much concurrency as possible between garbage collection operations and computational operations [3]. To minimize synchronization and improve locality, each processor had

its own heap in which to allocate objects. Each processor's heap was divided into two *semispaces*: *oldspace* and *newspace*. During each garbage-collection cycle, all live objects were copied from *oldspace* to some processor's *newspace*. At the beginning of each cycle, all processors needed to synchronize to swap the roles of the two semispaces, but after a short period to copy objects in the root set, all processors could proceed independently, interleaving computational operations with housekeeping intervals for moving live heap objects from *oldspace* to *newspace*. This approach has the advantage of allowing all processors to be fully utilized without requiring the garbage-collection load to be spread evenly across them, but the cost on stock hardware of checking pointers loaded from the heap to ensure that they do not point into *oldspace* is significant and might be too great to tolerate in an implementation based on compiled code.

To avoid the cost of checking pointers loaded from the heap, both MultiScheme and Mul-T employed non-incremental garbage collectors that required all processors to suspend computational activities and synchronize, then focus exclusively on garbage collection until the garbage collection pass is done, at which point they can resume computation. This approach streamlines the execution of computational tasks but puts severe requirements on the evenness with which garbage-collection work can be distributed among the processors. The parallelism achieved during garbage collection in these implementations was not carefully tuned and is not conclusively stated in the published reports, but there was evidence that speedups of 10 or more were difficult to achieve.

The garbage collector was also called on to perform other housekeeping tasks. All Multilisp implementations used the garbage collector to splice out resolved futures, replacing pointers to the future object with pointers to the future's value, eliminating any subsequent costs for indirection through the future's placeholder object. MultiScheme also used the garbage collector to kill tasks that are working on resolving futures that have become inaccessible.

## Scheduling

In programs with large amounts of concurrency, unfolding the task-spawning tree breadth-first can require exponentially more resources than unfolding

the task tree depth-first. Therefore, depth-first scheduling is strongly preferred, once enough concurrency has been exposed to keep all processors busy. In practice, this means that when processing the expression (*future X*), execution of the parent task should be suspended and the evaluation of *X* should begin immediately. If there is already plenty of work to keep all processors busy, the parent task will still be suspended when the evaluation of *X* completes, and the execution of the parent task can be resumed at that point. This was the scheduling policy used in Concert Multilisp [3].

In the design of Mul-T it was observed that the implementation of this policy could be speeded up by skipping the creation of a child task, allowing the "parent" task to just continue into the evaluation of *X*. If another processor needs work before the evaluation of *X* completes, it can break into this task's stack and steal away the continuation of the computation beyond the *future* construct. This policy of *lazy task creation* [14] greatly reduces the cost of task management, since most potential task-spawning points never actually result in the creation of a new task. This policy was subsequently adopted, and put on a sound theoretical footing, by the MIT Cilk project [2]. It is also related to the "evaluate-and-die" mechanism of Glasgow Parallel Haskell [16].

The informally stated goal of the Mul-T implementation was to make *future* "no more expensive than a procedure call." With lazy task creation, this cost was reduced to 9-12 machine instructions in most cases [14], coming close to this goal.

## Debugging Tools

In sequential programming, it is important that programs both be correct and have the needed performance, and this is no less true in parallel programming. Multilisp's mostly functional nature is intended to assist in developing correct programs by reducing opportunities for nondeterminacy and supporting a development style where testing a program on a sequential machine will go a long way toward assuring its correct behavior on a parallel machine. Beyond that, it is useful to be able to execute programs in parallel and replay the same execution path later on, if a problem is noticed. The Multilisp team's MulTVision project [7] included a log-and-replay technique, implemented in Mul-T, which reduced logging costs substantially by logging only task

creations and side effects on shared data structures, which are not common in mostly functional programs.

While performance optimization is often important for sequential programs, it is critical for parallel programs, since performance is the main reason to target a program to a parallel machine in the first place. In a large program, however, it can be quite hard to understand the reasons for the observed performance on a parallel machine. MulTVision also included a logger for events such as task creation, blocking, and unblocking, and a visualizer for viewing the resulting execution traces. The project was disbanded before the value of these tools could be clearly demonstrated, but this is likely to become an urgent problem in software development for modern multicore architectures.

## Performance

Mul-T is the most significant of the Multilisp implementations from the standpoint of assessing Multilisp's actual potential to deliver real parallel speedup over the best sequential implementation. The extra costs in the Mul-T implementation, compared with the sequential T implementation, come principally from two sources: synchronization and task scheduling. The major component of the synchronization costs is the checks that must be included in every operation that touches its operands, to determine whether any of the operands is a future and, if so, to take the needed actions. The cost for this depends considerably on the nature of the program, but after some optimizations, an increase of about 65% in execution time due to touch checks was fairly typical [12]. It would be interesting to know how this measurement would look on today's machines. When combined with lazy task creation, Mul-T was measured as delivering performance for a range of benchmarks, on 16 processors, of up to about 13 times the performance of a sequential program running in the unmodified T system [14].

## Related Work and Influences on Other Projects

Several other parallel Lisp languages and implementations of the 1980s are reviewed by Halstead [5] and are briefly mentioned here. Qlisp was based on Common Lisp augmented with futures as well as several other concurrency constructs and was implemented at

Stanford University on an Alliant FX-8 multiprocessor. Butterfly Lisp (closely related to MultiScheme) and Parallel Portable Standard Lisp were implemented on the BBN Butterfly multiprocessor. Both rely primarily on futures for concurrency. Farther afield, Concurrent Scheme at the University of Utah was targeted to a distributed-memory MIMD multiprocessor, while Connection Machine Lisp was designed for the SIMD architecture of the Connection Machine.

In addition to influencing contemporaneous parallel Lisp systems that also used futures, ideas from the Multilisp project, particularly in the areas of synchronization, task scheduling, and garbage collection, have found their way into various other systems. Especially notable among these systems is the work-stealing scheduler of the MIT Cilk system [2], which was developed in the mid-1990s and continues in use to this day. Cilk has been used to build very large parallel applications, such as the competition-quality CilkChess program. Most recently, these ideas have been commercialized in Cilk++, a tool for programming present-day multicore processors.

M

## Related Entries

- [Actors](#)
- [Cilk](#)
- [Debugging](#)
- [Futures](#)
- [Glasgow Parallel Haskell \(GpH\)](#)
- [Processes, Tasks, and Threads](#)
- [Speculation, Thread-Level](#)

## Bibliographic Notes and Further Reading

The basic principles behind Multilisp and Mul-T are outlined in [3] and [12], respectively. Lazy task creation, the essential idea that has been picked up in subsequent "work-stealing" schedulers, is described in [14]. Halstead [5] gives an extended overview of the entire parallel Lisp field as of 1990. Many members of the parallel Lisp and parallel symbolic computing communities gathered at a sequence of workshops held in 1989, 1992, and 1995. The proceedings of the first [9] and second [6] of these workshops include several papers discussing various aspects of the Multilisp project as well as papers about other contemporaneous projects mentioned in

this article. These volumes are thus a good starting point for further exploration of the literature.

## Bibliography

1. Baker H, Hewitt C (1977) The incremental garbage collection of processes. MIT Artificial Intelligence Laboratory Memo 454, Cambridge
2. Blumofe R et al (1995) Cilk: an efficient multithreaded runtime system. In: Proceedings of the fifth ACM symposium on principles and practice of parallel programming (PPoPP). July 1995, pp 207–216
3. Halstead R (1985) Multilisp: a language for concurrent symbolic computation. *ACM Trans Prog Lang Syst* 7(4):501–538
4. Halstead R (1986) Parallel symbolic computing. *IEEE Comput Mag* vol 19(8):35–43
5. Halstead R (1990) New ideas in parallel lisp: language design, implementation, and programming tools. In: Proceedings of US/Japan workshop on parallel lisp, Lecture Notes in Computer Science 441. Springer, Heidelberg, pp 2–57
6. Halstead R, Ito T (eds) (1993) Parallel symbolic computing: languages, systems, and applications. In: Proceedings of US/Japan workshop on parallel symbolic computing, Lecture Notes in Computer Science 748. Springer, Heidelberg
7. Halstead R, Kranz D, Sobalvarro P (1993) MulTVision: a tool for visualizing parallel program executions. In: Proceedings of US/Japan workshop on parallel symbolic computing, Lecture Notes in Computer Science 748. Springer, Heidelberg, pp 183–204
8. Halstead R, Loaiza J (1985) Exception handling in multilisp. In: Proceedings of the 1985 international conference on parallel processing, St Charles, IL, August 1985, pp 822–830
9. Ito T, Halstead R (eds) (1990) Parallel lisp: languages and systems. In: Proceedings of US/Japan workshop on parallel lisp, Lecture Notes in Computer Science 441. Springer, Heidelberg
10. Kelsey R, Clinger W, Rees J (eds) (1998) Revised<sup>5</sup> report on the algorithmic language scheme. *ACM SIGPLAN Not* 33(9):26–76
11. Kornfeld W, Hewitt C (1981) The scientific community metaphor. *IEEE Trans Syst Man Cybern*, pp 24–33
12. Kranz D, Halstead R, Mohr E (1989) Mul-T: a high-performance parallel lisp. In: Proceedings of ACM SIGPLAN '89 Conference on programming language design and implementation. Portland, Oregon, pp 81–90
13. Miller J (1987) MultiScheme: a parallel processing system based on MIT scheme. MIT Laboratory for Computer Science Technical Report TR-402, Cambridge
14. Mohr E, Kranz D, Halstead R (1991) Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE Trans Parallel Dist Syst* 2(3):264–280
15. Osborne R (1990) Speculative computation in multilisp. In: Proceedings of US/Japan workshop on parallel lisp, Lecture Notes in Computer Science 441. Springer, Heidelberg, pp 103–137
16. Peyton Jones S, Clack C, Salkild J (1989) High-performance parallel graph reduction. In: PARLE '89 Proceedings, Lecture Notes in Computer Science 365. Springer, Berkley, pp 193–206

## Multimedia Extensions

- [Vector Extensions, Instruction-Set Architecture \(ISA\)](#)

## Multiple-Instruction Issue

- [Superscalar Processors](#)

## Multiprocessor Networks

- [Hypercubes and Meshes](#)
- [Interconnection Networks](#)

## Multiprocessor Synchronization

- [Synchronization](#)
- [Transactional Memory](#)

## Multiprocessors

- [Distributed-Memory Multiprocessor](#)
- [Shared-Memory Multiprocessors](#)

## Multiprocessors, Symmetric

- [Shared-Memory Multiprocessors](#)

## MultiScheme

- [Multilisp](#)

## Multistage Interconnection Networks

- [Networks, Multistage](#)

## Multi-Streamed Processors

► Multi-Threaded Processors

## Multi-Threaded Processors

MARIO NEMIROVSKY

Barcelona Supercomputer Center, Barcelona, Spain

### Synonyms

Multi-streamed processors

### Definition

Opposed to standard single-threaded processors, in which only one software thread can be present and executed at any given time, Multithreaded Processors can hold multiple software threads which can be running simultaneously. The amount of information needed to be held per software thread is defined as the context. Therefore, a processor that can hold multiple contexts concurrently is defined as a Multithreaded Processor.

### Discussion

#### Introduction

In the continuing search for a higher level of performance, computer architects have been consistently pushing the speed of the system and the microarchitecture design. For instance, microprocessor speeds in the last 40 years have grown by 4,000 times while the instructions per clock have increased from 0.125 on the Intel 4004 to at least 3 per core on the Intel Xeons which have a 64-bit data width compared to the 4-bit 4004. Yet, the semiconductor technology and microarchitecture innovations that led to these processor performance gains have started to provide diminishing returns. As pulling out substantial performance benefits from these techniques starts turning increasingly tedious, higher processor efficiency becomes attainable through the exploitation of higher-level parallelism. It is important to take into account all the different levels at which parallelism can play a significant role so looking beyond Instruction Level Parallelism (ILP) is Thread Level Parallelism (TLP). A thread is a process with its

own instruction and data. These processes can be totally independent or have different levels of sharing.

A multithreaded processor is able to concurrently execute instructions of different threads within a single pipeline. Multithreading could be applied either to increase performance of multiprogramming or multithreaded workloads or to increase performance of a single program.

Multithreaded processors interleave the execution of instructions of different user-defined threads in the same pipeline. This requires that multiple program counters and contexts be present, often stored in different register sets. The execution units are multiplexed between the running threads. Latencies that occur in the computation of a single instruction stream (also known as bubbles) are filled by the instructions of another thread. Context switching within the running threads of the hardware is automatically assigned.

A multithreaded processor capable of executing multiple instructions from different instruction streams (i.e., threads running within the hardware) simultaneously is called simultaneous multithreaded (SMT) processor. SMT is the confluence of multithreading techniques on a wide superscalar processor so that the hardware pipeline is used more efficiently.

M

#### Classification of Multithreading

##### Techniques

Multithreaded processors execute different instructions from different threads simultaneously within a same pipeline. At the early stages, multithreading techniques were developed for use in scalar processors and consisted of two types commonly known as Fine Grain Multithreading (FGMT) and Coarse Grain Multithreading.

In fine grain multithreading, every new instruction issue derives from a different thread so that no two instructions from the same thread are present in the pipeline at any given time. Furthermore, the overall efficiency of the processor is increased over standard scalar processors because the bubbles normally caused by high latency events get used by other threads. In order to have a high efficiency, at least as many threads as pipe stages are needed but if only one single thread is active, then the performance is  $1/n$  ( $n$  being the number of pipe stages) of a scalar processor. The hardware complexity on one

hand increases due to the need of replicating the context registers, however, on the other hand, the pipeline complexity becomes simplified because dependencies between instructions is almost nonexistent, since every instruction is from a different thread, so bypass logic, branch prediction, and out of order execution are not needed.

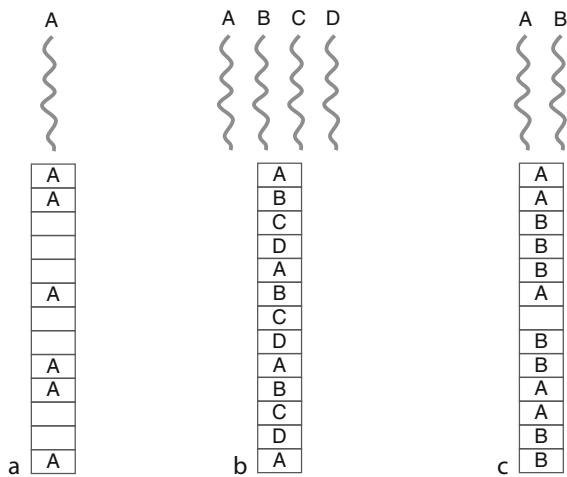
Coarse grain multithreading (CGMT), similarly can handle various threads in a single pipeline but every new issued instruction can derive from any thread independent of what thread instructions are currently in the pipeline. This implies that if a single thread is active, then the performance of that thread is the same as one would have achieved in a single-threaded processor. Coarse grain is beneficial in cases that require high performance similar to a single-threaded processor or systems where there are fewer software threads available. Although the complexity of a CGMT processor is larger than the FGMT, since it includes both multithreading hardware support plus the dependency checks and bypasses of a single-threaded processor, a CGMT requires less threads to remain efficient than the fine grain and is more efficient than a single-threaded processor when more than one thread is running.

The advent of superscalar processors created the possibility of executing several instructions at a time that could, by applying multithreading techniques, be derived from several threads leading to what is commonly known as simultaneous multithreaded (SMT). The amount of instruction bubbles that can be covered by SMT is generally greater than fine or coarse grain multithreading; however, the hardware cost complexity is also greater.

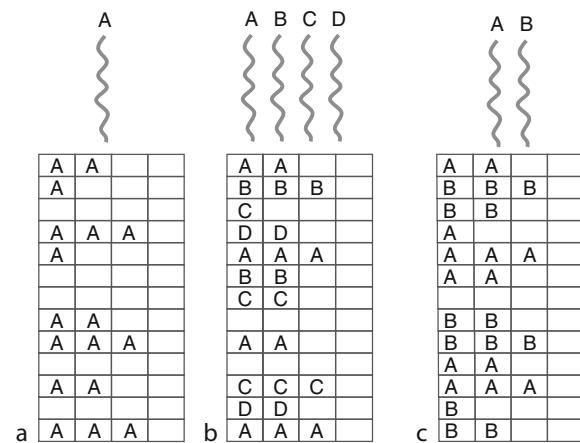
### Benefit of Multithreaded Processors

Multithreaded Processors (MTP) achieve higher performance than single-threaded processors by taking advantage of the thread level parallelism (TLP) and executing various threads to cover bubbles that get generated in the pipelines. These bubbles are the result of long latency instructions, data and control dependencies, cache misses, exceptions, etc. By covering these bubbles by executing instructions from other ready threads, MTPs can achieve much greater efficiency than their single-threaded counterparts.

In Fig. 1, it can be seen how the bubbles that get generated on a single-threaded scalar gets utilized by



**Multi-Threaded Processors. Fig. 1** (a) Single-threaded scalar, (b) fine grain multithreading scalar, and (c) coarse grain multithreading scalar



**Multi-Threaded Processors. Fig. 2** (a) Single-threaded super scalar, (b) fine grain multithreading super scalar, and (c) coarse grain multithreading super scalar

a fine grain multithreading (1b) and a coarse grain multithreaded processor (1c), whereas the superscalar version are represented in Fig. 2.

Nowadays, there is a significant amount of applications that feature a large quantity of thread level parallelism (TLP) making multithreaded processors a valued practical solution for achieving significant improvements in performance and power. Examples of applications with large TLP include video, gaming,

modeling, simulations, web searching, as well as web 2.0 applications.

There are two forms of attaining thread level parallelism. One way is to extract TLP from an application by splitting it into separate tasks that can be run in parallel, for example, splitting a video screen into sections and having them be executed in parallel. Additionally, thread level parallelism exists when several independent tasks arrive almost simultaneously, for instance with network processors, packets arrive very close to each other (i.e., packet bursts) and each packet can be processed in its own thread.

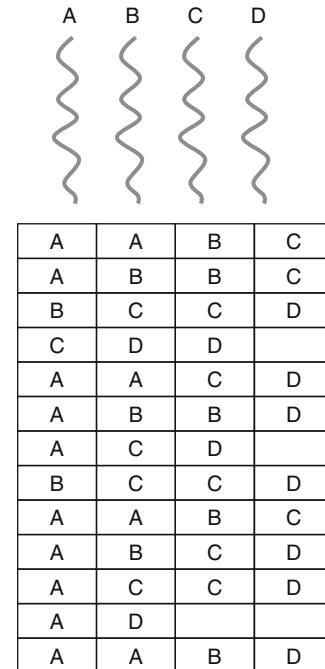
Multithreading can also be used to make a single thread run faster. Even if a thread cannot be separated into further parallel tasks or there are not many threads available, multithreading techniques such as multipath speculative execution and helping threads (explained in section Simultaneous Multithreading) can be used to accelerate the execution of single threads.

It is important to differentiate the use of multithreading from chip multiprocessors (CMPs). Multicores are made up of independent processors that normally only share a L2 cache while a multithreaded processor shares the processor pipeline yet has separate context registers for the threads. Furthermore, multithreading is used to augment the processor utilization by filling pipeline bubbles while multicores are just adding processors to the overall chip. In terms of resources needed to build multithreaded processors compared to multicore processors, the cost of adding a thread is much lower than adding a whole separate core. However, there is a performance saturation point for how many threads can be put into a single core after which the more threads added can actually cause performance degradation. Therefore what is commonly used is a blending of multithreading and multicore techniques which is discussed below.

### Fine Grain Multithreading (FGMT)

In an FGMT processor, in every cycle, instructions from a unique thread get issued and only when those instructions retire can new instructions from that same thread be issued.

For example, when a thread accesses memory, the thread does not get scheduled again until the memory transaction gets completed. This model requires at least as many threads as pipeline stages in the processor, and



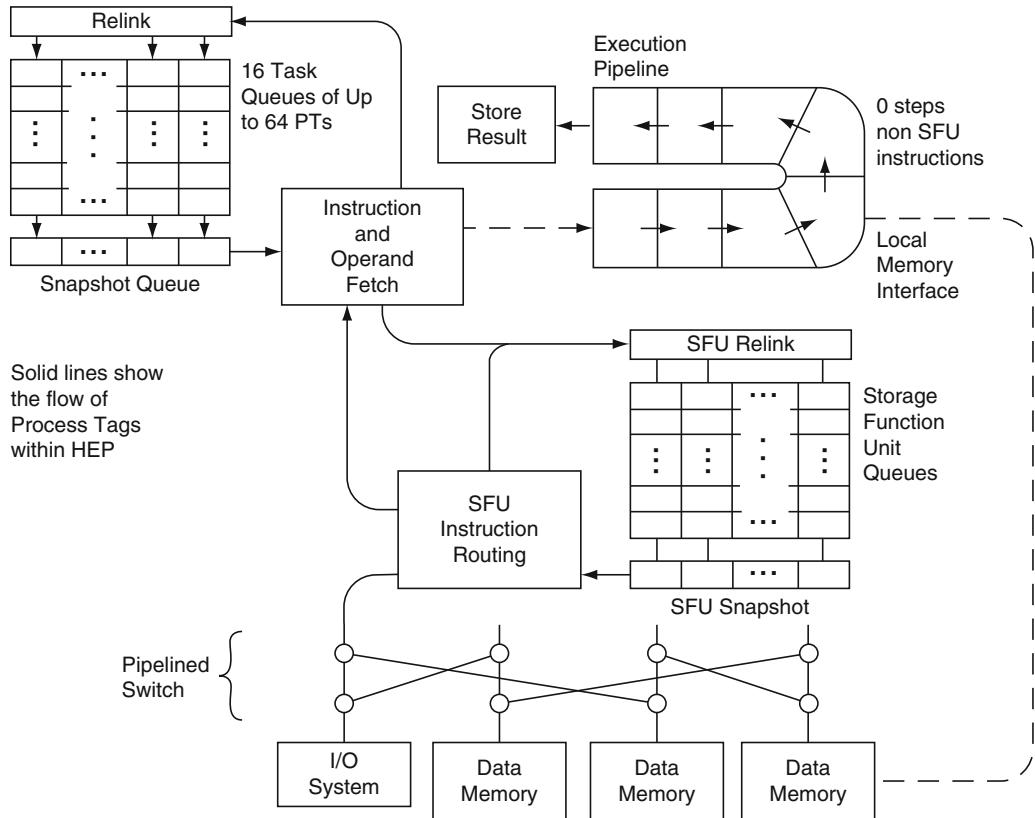
**Multi-Threaded Processors. Fig. 3** Simultaneous multithreading

M

adding a few more threads will allow the processor to tolerate higher latencies.

The first commercial FGMT computer was the Denelcor HEP [14, 21, 22] in 1978; however, in the 1960s, Control Data Corporation in their Peripheral Processors (PPs) of the CDC6600 [25] incorporated 10 threads (or 10 PP) that executed in a round robin fashion meaning that the execution units would execute one instruction cycle from the first PP, then one instruction cycle from the second PP, etc. This was done both to reduce costs and because accesses to memory required 10 PP clock cycles: when a PP accessed memory, the data was available next time the PP received its slot time.

The HEP system shown in Fig. 4 could have upto 16 processors with 128 threads per processor. It had an eight-stage pipeline that matched the minimum latency delay from memory fetches. Therefore, upto eight threads were in execution at any given time within a single processor. Furthermore, only a single instruction from a given process was permitted to be in the pipeline at any point in time.



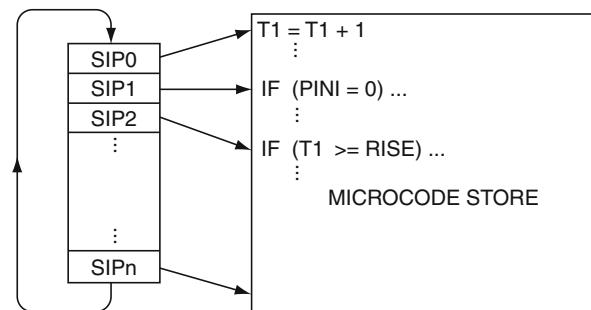
**Multi-Threaded Processors. Fig. 4** HEP-1 processor pipeline

In the late 1980s, Delco Electronics, a division of General Motors, introduced the first real-time fine grain multithreaded processor, and perhaps the most successful FGMT processor deployed since 1987, known as the Timer Input Output (TIO) [18]. It was designed as a two-stage pipeline with two levels of fine grain multithreading. The first level used a round robin method of choosing between the M. Instruction Processor (MIP) and the S. Instruction Processor (SIP). On the second level, the SIP on its own could hold up to 32 separate threads. The issue pattern resembled: MIM, SIM\_1, MIM, SIM\_2, MIM, SIM\_3, ... MIM, SIM\_32, MIM, SIM\_1. Figure 5 shows the SIP scheduler.

Other popular FGMT processors were the Horizon [24], the Cray MultiThreaded Architecture (MTA) [2], the Multilisp Architecture for Symbolic Applications (MASA) [8], and M\_Machine [15].

### Coarse Grain Multithreading

In Course Grain MultiThreading (CGMT) a single thread is executed until it encounters a high latency



**Multi-Threaded Processors. Fig. 5** The single instruction processor of the TIO

event at which point the hardware automatically performs a context switch to another thread. Examples of long latency events are page faults and L2 cache misses. In situations in which context switches can be performed very quickly, even small bubbles generated by such events as branches can be used to trigger a context swap. Even with a small number of threads high processor utilization can be achieved while if only

a single thread is being executed, then the processor performance is like that of a single-threaded processor.

CGMT is very useful in situations that many different processes need to be run not all of which have the same priority. A particular example of using CGMT for real-time processing was the DISC [19]. DISC was a CGMT processor that performed context switching in order to distribute processing power between the active threads. Also the MSpard [17] was targeting real-time systems.

Several processors use the CGMT technique: the MIT Sparcle [17], the MIT J-Machina [20], and the Rhamma processor [7].

### Simultaneously Multithreading

A superscalar processor where instructions can be issued from multiple threads in the same cycle is called Simultaneous Multithreaded (SMT). Figures 2 and 3 compare a superscalar, a CGMT, an SGMT, and an SMT. It is clear to visualize how these diverse techniques can take advantages of the bubbles that gets generated in a processor. SMT is the technique that has more flexibility to cover the inefficiencies of the single-threaded architectures. An SMT processor has more flexibility than CGMT or FGMT processors because it allows an instruction from any thread to be executed at any time while in parallel, an instruction from a different thread may also be executed.

The first work where SMT was proposed on top of a standard RISC processor (IBM RS6000 Power) was presented by Yamamoto in [29, 30]. This work extended the concept of FG and CG multithreading (then called *multistreaming*) technique to general-purpose superscalar processor architecture. In their work, an IBM Power processor was modified to produce an SMT processor capable of running up to 8 threads simultaneously. They also presented an analytical model of the processor. Another pioneering approach to SMT was the multithreaded processor created in the Media Research Laboratory of Matsushita Electric Ind. (Japan) [9]. However, it was the SMT process or architecture, proposed by D. Tullsen, S. Eggers, and H. Levy [26] at the University of Washington, Seattle, WA, which has given the name to the technique.

Another way to use multithreading is to increase the performance of sequential programs. A processor can spawn threads from a single thread and execute

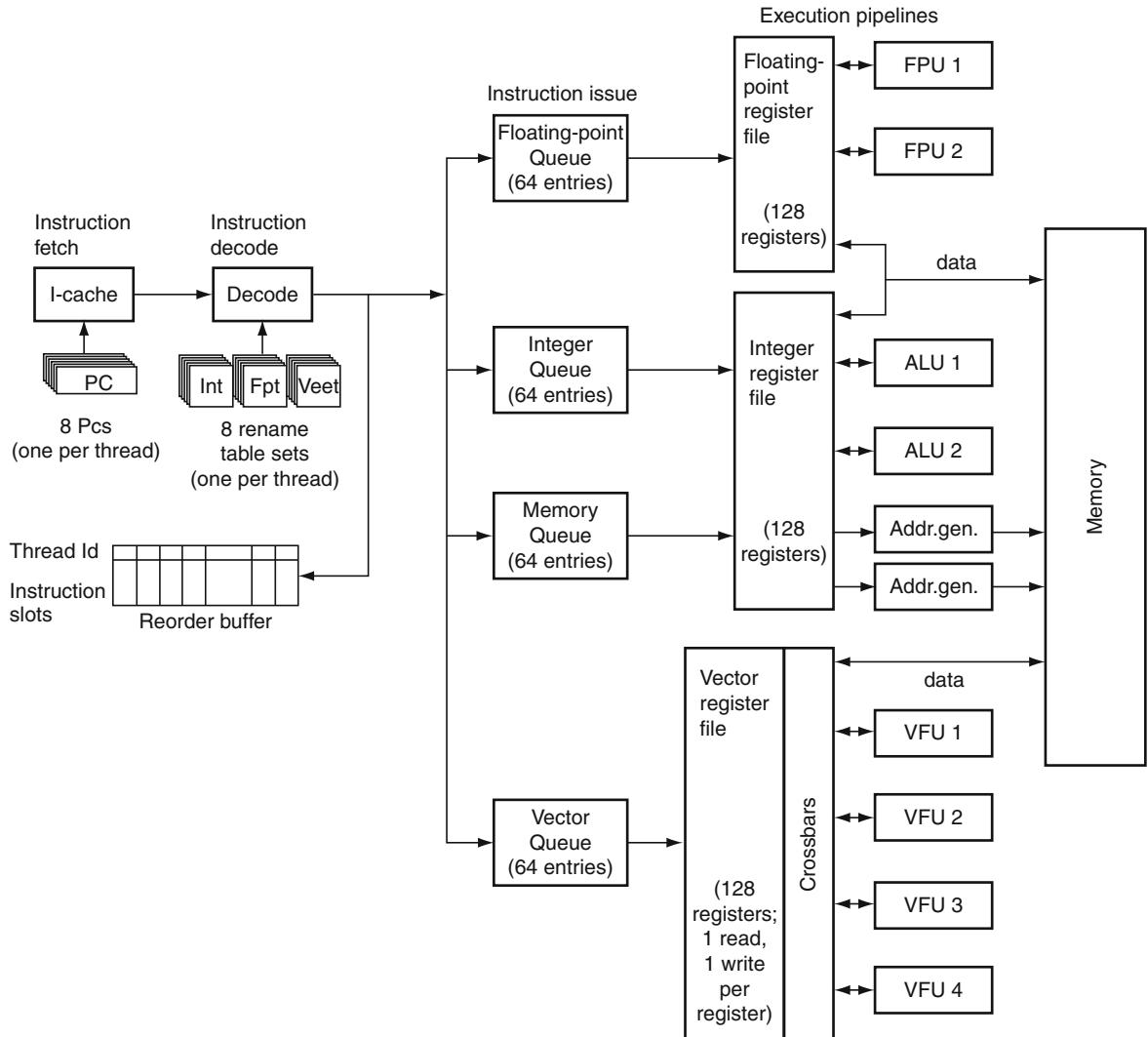
them speculatively concurrently with the main thread in order to accelerate the main program. Several SMT approaches fall in this area such as simultaneous subordinate microthreading [3], multiple path execution [28], the multiscalar processor [23], and the single-program speculative multithreading architecture [4].

Another interesting work on SMTs is the Simultaneous Multithreaded Vector (SMV) architecture [6], which combines simultaneous multithreaded execution and out-of-order execution with an integrated vector unit and vector instructions.

### Simultaneous Multithreading Commercial Processors

*Alpha21464.* Compaq unveiled its Alpha EV821464 in 1999 [12], a four-threaded eight-issue SMT processor. The out-of-order execution processor had a large on-chip L2 cache, a direct RAMBUS interface, and an on-chip router for system interconnect of a directory-based, cache-coherent NUMA (nonuniform memory access) multiprocessor. This 250 million transistor chip was planned for the year 2003. The project was abandoned, as Compaq sold the Alpha processor technology to Intel.

In 2001, Clearwater networks unveiled the CNP810SP processor [12] shown in Fig. 7. This core incorporates Simultaneous Multithreading (SMT) capabilities, whereby eight threads execute simultaneously, utilizing a variable number of resources on a cycle by cycle basis. In each cycle, anywhere from zero to three instructions can be executed from each of the threads depending on instruction dependencies and availability of resources. The maximum instructions per cycle (IPC) of the entire core is ten. In each cycle, two threads are selected for fetch and their respective program counters (PCs) are supplied to the dual-ported instruction cache. Each port supplies eight instructions, so there is a maximum fetch bandwidth of 16 instructions. Each of the eight threads has its own instruction queue (IQ) which can hold up to 16 instructions. The two threads chosen for fetch in each cycle are the two that have the fewest number of instructions in their respective IQs. Each of the eight threads has its own 31 entry register file (RF). Since there is no sharing of registers between threads, the only communication between threads occurs through memory. The eight threads are



**Multi-Threaded Processors. Fig. 6** Simultaneous multithreaded vector architecture

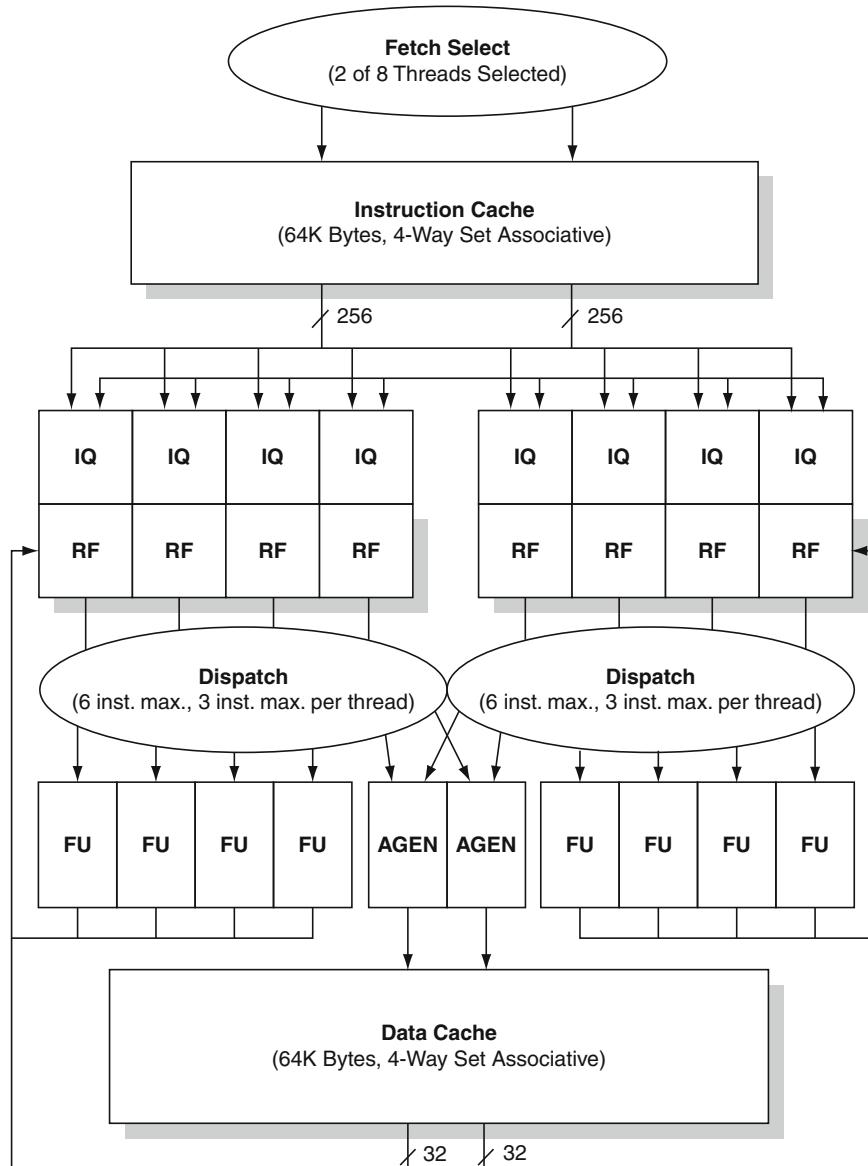
divided into two clusters of four for ease of implementation. Thus the dispatch logic is split into two groups where each group dispatches up to six instructions from four different threads. Eight function units are grouped into two sets of four, each set dedicated to a single cluster. There are also two ports to the data cache that are shared by both clusters. A maximum of ten instructions can be dispatched in each cycle. The function units are fully bypassed so that dependent instructions can be dispatched in successive cycles. Clearwater was out of business in 2001 by tape out date.

The UltraSPARCV, which exploited SMT, was able to switch between two different modes depending on the

type of work – one mode for heavy duty calculations and the other for business transactions such as database operations [11].

UltraSPARC T1 [10] is a microprocessor that is *both* multicore and multithreaded. The processor is available with four, six, or eight CPU cores, each core able to handle four threads concurrently. Thus the processor is capable of processing up to 32 threads concurrently.

UltraSPARC T2 [10] is a member of the SPARC family, and the successor to the UltraSPARC T1. The processor, manufactured in 65 nm, is available with eight CPU cores, and each core is able to handle eight



Copyright © 2002, Stephen W. Melvin

Multi-Threaded Processors. Fig. 7 The CNP810SP processor

threads concurrently. Thus the processor is capable of processing up to 64 concurrent threads.

Hyper-Threading Technology [16] proposed SMT for the Pentium 4-based Intel Xeon processor family to be used in dual and multiprocessor servers. Hyper-Threading Technology makes a single physical processor appear as two or more logical processors by applying the SMT approach. Each logical processor maintains a complete set of the architecture state, which

consists of the general-purpose registers, the control registers, the *advanced programmable interrupt controller* (APIC) registers, and some machine state registers. Logical processors share nearly all other resources on the physical processor, such as caches, execution units, branch predictors, control logic, and buses. Each logical processor has its own APIC. The interrupts sent to a specific logical processor are handled only by that logical processor.

## Multithreading on Power Efficiency

An important issue to discuss is power efficiency on multithreaded computers. As shown in the previous sections multithreaded processors fill these bubbles that get generated in the pipeline with useful instructions thus making the power per instruction overall more efficient. Speculation, used to achieve high performance on single-threaded machines, normally consumes not only power at the moment of computation but also at the moment of squashing if a misprediction occurs. Additionally, the hardware complexity of the predictors drain energy but multithreaded processors can make significant energy efficiency strides by not needing to implement as many predictors to achieve as high performance as single-threaded processors. Furthermore, a significant amount of logic used in superscalar design can be eliminated, such as out of order. In the case of FGMT, most of these logics which in turn represent additional power saving. On the other hand, the additional required context cost in multithreaded processors demands more power but in general this amount of power is not significant. Therefore, the overall power efficiency of multithreaded processors is much better than with single-threaded processors and explains why the technique is in use in the top-of-the-line power-efficient processors today.

## Pros

The major benefit of multithreading is the improvement of processor and memory utilization. Ideally as you add threads to the processor, the performance of the running thread will remain constant while the total system performance will be the sum of the performance of each thread. In real systems, as you add threads the performance of the running ones will degrade. In order to achieve performance increase throughout multithreading, it is important that the performance obtained by adding a thread should be larger than the performance degradation caused to the other threads. In many cases, threads could share code and/or data resulting in positive interference in the caches. In other words, as a thread needs a new piece of code it finds it in the instruction cache because a co-running thread had already requested it. Additionally, what will normally be a bubble in the execution pipeline can be easily used by a co-running thread to execute instructions without

causing penalties to the main running thread. The hardware complexity of the system can be reduced since it is the total system performance that is sought and not just the single thread performance, making complex microarchitecture designs (e.g., OOO execution, branch predictors, etc.) unnecessary.

## Cons

There are also a few issues to take into account before using multithreaded processors. If the performance of a single running thread is what has been sought, then the best way to use a multithreaded architecture is to use the “helping thread or speculative multithreading” [guri]. However, in a system where multithreading is used to achieve higher levels of system performance, it is important to notice that the individual performance of each thread will be, in general, lower than a thread will have when running alone. This is due to the fact that interference between threads will be mostly negative, causing trashing on the caches and TLBs, therefore increasing the misses and thread running time. It is essential to take advantage of thread interference by co-running threads with very low negative interference and high positive interference.

## Future Directions

Today, multithreaded multicore is pervasive in the industry. However, there are many areas that require further study, for instance, how to program these processors, how to map threads to streams, and how to perform run time debugging. These interesting challenges are the scope of work of various professionals and academics today, ensuring the extension of multithreading throughout the computing world which will branch out to also cover most areas of processing such as graphics, DSP, high performance, and low power.

In the future, asymmetric multithreaded processors, in which every thread could be specialized for a particular task, will start appearing.

## Related Entries

- ▶ [Computer Graphics](#)
- ▶ [Control Data 6600](#)
- ▶ [DEC Alpha](#)
- ▶ [Denelcor HEP](#)
- ▶ [Instruction-Level Parallelism](#)
- ▶ [POSIX Threads \(Pthreads\)](#)

- ▶ Parallelization, Automatic
- ▶ Processes, Tasks, and Threads
- ▶ Shared-Memory Multiprocessors
- ▶ Speculation, Thread-Level
- ▶ Superscalar Processors
- ▶ Tera MTA

## Bibliographic Notes and Further Reading

As stated earlier, multithreaded work began in the 1960s on the Control Data Computer (CDC600) Peripheral Control Processor; a description can be found in Design of a Computer – The Control Data 6600 [25]. The first FGMT paper by B. J. Smith, “A Pipelined Shared Resource MIMD Computer,” [22] in 1978 where he introduced the HEP machine followed a few years later in 1988 by the Horizon machine in “A Processor Architecture for Horizon,” Mark R. Thistle and B.J. Smith [24]. The DISC architecture which was a pioneer of SMT can be read in “DISC: dynamic instruction stream computer,” proceedings of the 24th annual international symposium on Microarchitecture [19]. There have been several valued reviews on multithreaded processors. A very complete one can be found in the book *Multithreaded Computer Architecture: A Summary of the State of the Art* by Robert A. Iannucci, Guang R. Gao, Robert H. Halstead Jr, and Burton Smith [13]. The book is divided in four parts covering from basic definitions up to compilation techniques. Additionally, Theo Ungerer, in 2003, has done a good examination of multithreaded processors in his paper “A survey of processors with explicit multithreading” [27].

## Bibliography

1. Agarwal A et al (1993) Sparcle: an evolutionary processor design for large-scale multiprocessors. IEEE Micro 13:48–61
2. Anderson W (2005) Experiences using the cray multi-thread architecture (MTA 2). 2003 user group conference (DoD UGC'03), Bellevue, ISBN: 0-7695-1953-9
3. Chapell R et al (1999) Simultaneous subordinate microthreading (SSMT). In: Proceedings of the 26th annual international symposium on computer architecture, Atlanta, GA. IEEE Computer Society Washington, DC, pp 186–195
4. Dubey P et al (1995) Single-program speculative multithreading (SPSM) architecture: compiler-assisted fine-grain multithreading. Tech. Rep. RC 19928. IBM, Yorktown Heights, NY
5. Emer J (1999) Simultaneous multithreading: multiplying aspha's performance. In: Proceedings of the microprocessor forum, San Jose, CA
6. Espasa R, Valero M (1997) Exploiting instruction- and data-level parallelism. IEEE MICRO 17(5):20–27
7. Grunewald W, Ungerer T (1997) A multithreaded processor designed for distributed shared memory systems. In: Proceedings of the international conference on advances in parallel and distributed computing, APDS-97, Shanghai
8. Halstead Jr RH (1988) MASA: a multithreaded processor architecture for parallel symbolic computing. ACM SIGARCH Computer Architecture News 16(2):43–451
9. Hirata H et al (1992) An elementary processor architecture with simultaneous instruction issuing from multiple threads. In: Proceedings of the 19th international symposium of computer architecture, Gold Coast, Australia. ACM Press, New York, pp 135–145
10. <http://www.opensparc.net>
11. [http://www.pcworld.com/article/105126/sun\\_hints\\_at\\_ultrasparc\\_v\\_and\\_beyond.html](http://www.pcworld.com/article/105126/sun_hints_at_ultrasparc_v_and_beyond.html)
12. <http://www.zytek.com/~melvin/clearwater.html>
13. Iannucci R, Gao G, Halstead R Jr, Smith B (1994) Multithreaded computer architecture: a summary of the state of the art. The Springer International Series in Engineering and Computer Science. Kluwer, Norwell, ISBN 0-7923-9477-1
14. Jordan HF (1983) Performance measurements on HEP – a pipelined MIMD computer. In: Proceedings of the 10th annual international symposium on computer architecture. IEEE, New York, ISBN 0-89791-101-6
15. Keckler SW, Dally WJ et al (1995) The M-machine multicomputer. In: Proceedings of the 28th annual international symposium on computer architecture, Ann Arbor, MI. ACM Press, New York, pp 146–156
16. Marr D et al (2002) Hyper-threading technology architecture and microarchitecture: a hypertext history. Intel Technol J 6(1)
17. Metzner A, Niehaus J (2000) MSparc: multithreading in real-time architectures. J Univ Comput Sci 6(10):1034–1051
18. Nemirovsky et al (1992) Microprogrammed timer processor. US patent 5117387
19. Nemirovsky M, Brewer F, Wood R (1991) DISC: dynamic instruction stream computer. In: Proceedings of the 24th annual international symposium on microarchitecture. Albuquerque, New Mexico
20. Noakes MD, Wallach DA, Dally WJ (1993) The J-machine multicomputer: an architectural evaluation. In: Proceedings of the 20th annual international symposium on computer architecture, San Diego, CA. ACM Press, New York, pp 224–235
21. Smith B (1985) On parallel MIMD computation: HEP supercomputer and its applications. Massachusetts Institute of Technology, Cambridge, MA, ISBN 0-262-11101-2
22. Smith BJ (1978) A pipelined, shared resource MIMD computer. In: Proceedings of the 1978 international conference on parallel processing. IEEE press, Piscataway, pp 6–8
23. Sohi G, Breach S, Vijaykumar T (1995) Multiscalar processor. In: Proceedings of the 22nd annual international symposium on computer architecture, Santa Margherita Ligure, Italy, June 1995. Kluwer, Norwell, pp 414–425

24. Thistleton MR, Smith BJ (1988) A processor architecture for horizon. In: Proceedings of the 1988 ACM/IEEE conference on supercomputing, Orlando, FL. IEEE Computer Society Press, Los Alamitos, pp 35–41
25. Thornton JE (1970) Design of a computer – the control data 6600. Scott, Foresman and Co, Glenview
26. Tullsen D, Eggers S, Levy H (1995) Simultaneous multithreading: maximizing on-chip parallelism. In: Proceedings of the 22nd annual international symposium on computer architecture, Santa Margherita Ligure, Italy, June 1995. Kluwer, Norwell, pp 392–403
27. Ungerer T, Bobic B, Silc J (2003) A survey of processor with Explicit Multithreading. ACM Comput Surv (CSUR) 35(1):29–63
28. Wallace S, Calder B, Tullsen D (1998) Threaded multiple path execution. In: Proceedings of the 25th annual international symposium on computer architecture, Barcelona, Spain. IEEE Computer Society Washington, DC, pp 238–249
29. Yamamoto W, Nemirovsky M (1995) Increasing superscalar performance through multistreaming. In: Proceedings of the international conference on parallel architectures and compilation techniques, Limassol, Cyprus. IEEE Computer Society Washington, DC, pp 49–58
30. Yamamoto W, Serrano M, Wood R, Nemirovsky M (1994) Performance estimation of multistreamed superscalar processors. In: Proceedings of the Hawaii conference on systems and science, Hawaii, January 1994. IEEE Computer Society, Los Alamitos, pp 195–204

## Mumps

PATRICK AMESTOY<sup>1</sup>, ALFREDO BUTTARI<sup>1</sup>, IAIN DUFF<sup>2</sup>,  
ABDOU GUERMOUCHE<sup>3</sup>, JEAN-YVES L'EXCELLENT<sup>4</sup>,

BORA UÇAR<sup>4</sup>

<sup>1</sup>Université de Toulouse ENSEEIHT-IRIT,  
Toulouse, France

<sup>2</sup>Science & Technology Facilities Council, Didcot,  
Oxfordshire, UK

<sup>3</sup>Université de Bordeaux, Talence, France

<sup>4</sup>ENS Lyon, Lyon, France

## Synonyms

MUMPS

## Definition

MUMPS (MULTifrontal Massively Parallel Solver) is a parallel library for the solution of sparse linear equations. It primarily targets parallel platforms with distributed memory, where the message passing paradigm MPI is used. MUMPS is a direct code, based on Gaussian elimination. It will solve sparse linear systems with

a real unsymmetric, symmetric positive definite, or symmetric indefinite coefficient matrix and will solve complex systems where the matrix is unsymmetric or complex symmetric. MUMPS has a large number of options, some to enhance functionality and some to improve performance or core memory usage. Whereas most direct solvers for distributed memory environments rely on static approaches where the computational tasks are known and assigned to the processors in advance, one of the main originalities of MUMPS is its ability to perform dynamic pivoting in order to guarantee numerical stability, leading to dynamic data structures and non-fully predictable tasks graphs. In order to handle these dynamic aspects, the management of parallelism is based on a fully asynchronous approach, and the computational tasks are defined and scheduled dynamically at runtime.

## Discussion

### Introduction

MUMPS (MULTifrontal Massively Parallel Solver) is a direct code, based on Gaussian elimination, for the solution of sparse linear equations on distributed memory machines, although a serial version is available.

With a direct code, the first step in solving a linear system of equations of the form

$$Ax = b \quad (1)$$

is to perform the matrix factorization

$$PAQ = LU, \quad (2)$$

where  $P$  and  $Q$  are permutation matrices,  $L$  is a lower triangular matrix, and  $U$  is an upper triangular matrix. The solution to the Eq. 1 is then obtained through a forward elimination step

$$Ly = Pb,$$

followed by the backsubstitution

$$UQ^T x = y.$$

When  $A$  is sparse, that is, when there are many zero entries in  $A$ , the computation will be organized so that  $L$  and  $U$  are also sparse, but there are in general significantly more nonzeros in  $L$  and  $U$  than in  $A$ .

If the matrix  $A$  is symmetric, then the factorization

$$PAP^T = LDL^T \quad (3)$$

is used, where  $D$  is a block diagonal matrix with blocks of order 1 or 2 on the diagonal. The blocks of order two are required for stability when factorizing symmetric indefinite matrices.

MUMPS uses the factorization (2) when the matrix is unsymmetric and the factorization (3) when the matrix is symmetric.

Before discussing the MUMPS code in more detail in the following sections, it is worth recalling how the steps given above for solving Eq. 1 are implemented within sparse direct codes. In most sparse direct codes, this is done in three steps, namely:

*Analysis:* In this step of the computation, the sparsity pattern of  $A$  (or, when the matrix is unsymmetric, of the pattern of  $|A| + |A|^T$ , where  $|A|$  is the matrix whose entries are the modulus of the corresponding entries in  $A$ ) is used to determine an ordering to maintain sparsity in the triangular factors,  $L$  and  $U$ . This is called the ordering phase. The analysis step then continues by organizing data structures so that the subsequent numerical factorization can be computed efficiently. The main part of this phase is often called *symbolic factorization*.

*Factorization:* In the next step, the factors  $L$  and  $D$  (or  $L$  and  $U$  in the unsymmetric case) are computed using the information from the analysis step. This is the most computationally intensive step and generally takes the most time to execute, although much work has been done to run this efficiently on parallel computers. If the matrix is positive definite, it is usually possible just to follow exactly the pivoting sequence obtained from the ordering phase. If, however, the matrix is unsymmetric or symmetric indefinite, then care must be taken to avoid numerical instability.

*Solve:* In this last step, the factors from the factorization step are used to solve the system through the forward and back substitutions shown above. While this step is computationally cheap, involving effectively only two arithmetic operations for each entry in the factors, it suffers greatly in performance because of the amount of data movement relative to this small amount of arithmetic.

Sparse direct codes differ in the way these steps are performed. MUMPS uses a multifrontal approach to effect these steps. The details of this approach are

discussed in another entry of this Encyclopedia. Later in this entry, a short history of the multifrontal developments that led to the first version of MUMPS is given at the beginning of ►section Origins of MUMPS. In the remainder of this section, the way MUMPS realizes the three steps indicated above will be discussed. MUMPS is as much a research project as a code, and this aspect is discussed in ►section Further Comments.

## General Comments

As said in the definition, MUMPS is a parallel code that uses MPI for message passing. It will solve linear systems with a real unsymmetric, symmetric positive definite, or a symmetric indefinite coefficient matrix and will solve complex systems where the matrix is unsymmetric or complex symmetric. Over the years, a large number of options have been developed, some to enhance functionality and some to improve performance. The range of functionalities of MUMPS is far greater than that of any of other sparse direct code at the present time. In addition to handling the very wide range of matrices listed above, MUMPS accepts the input matrix as a distributed assembled matrix or in elemental format, that is to say the matrix is represented as an expanded sum of small dense matrices each based on a single finite element or on a group of such elements.

The parallel model used by MUMPS relies on a dynamic and distributed management of the computational tasks (dynamic scheduling): each process is responsible for providing work to some other processes while at the same time acting as a worker for others. One outcome of such a distributed management is that the work is performed asynchronously so that computation and communication are very much overlapped. Since the main pivoting algorithms in the symmetric indefinite and unsymmetric cases will alter the order of computation of the factors and will usually change the data structures, MUMPS has from its early days always used dynamic scheduling [8]. While initially required to accommodate numerical pivoting, the dynamic scheduling approach is also a powerful tool for the control of memory usage and for efficiency in a multiuser environment where the load on the processors can have a significant influence on performance. In fact, much of the research activities related to MUMPS (summarized in ►section Further Comments) have been concerned with this aspect of the package.

Options exist to run the analysis step in parallel, including a priori scaling of the matrix, and both the factorization and solution phases can be run in parallel although uniprocessor versions for all phases are also available.

## Analysis Step

As discussed in ►section [Introduction](#), the analysis step consists of two main phases: the selection of a pivot ordering to maintain sparsity and the creation of data structures for the subsequent factorization and solution steps (the symbolic factorization). In MUMPS, the symbolic factorization phase produces the assembly tree that is used in the multifrontal factorization. In the analysis step, MUMPS does a static scheduling phase that distributes the nodes of the tree to the processes. This static phase attempts to balance arithmetic and storage over the processors but, for the reasons mentioned in the previous section, will need to be followed by dynamic scheduling in the factorization step. In particular, because of limited tree parallelism near the top of the tree, more than one process is required for each node that appears in that area: the process in charge of such a node will select dynamically other processes that will participate in the computations at that node, possibly among a set of candidate processors [9] chosen at analysis. Because one of the features of MUMPS is that the execution should fit within a memory area allocated at the beginning of the factorization, an accurate estimate of the memory required for the numerical factorization is required. Such an estimate is computed by each process at the analysis phase through a “symbolic” traversal of the tree where the algorithm mimics the memory requirements that will occur at the factorization step. This feature is available for both parallel and sequential execution. The accuracy of the estimate degrades depending on the amount of numerical pivoting which may happen during the numerical factorization, and on the dynamic behavior of the parallel factorization (which is by nature difficult to predict). In practice, allowing a small percentage increase to the estimate is sufficient.

For the ordering phase, there are several sparsity ordering methods available to MUMPS users, and there is an option for automatically choosing an ordering method based on the size and density of the matrix, the ordering packages available to MUMPS at installation time, and the number of processors being used. The

user can also input his or her own pivotal order. Note that the ordering option can have a significant impact on the shape of the assembly tree and on the memory requirements during the numerical phases [18, 19]. A range of ordering options is supported natively by MUMPS including the AMD (Approximate Minimum Degree) method and some variants of it, AMF (Approximate Minimum Fill), QAMD (Approximate Minimum Degree with detection of quasi dense rows), and PORD (Paderborn ORDering). Graph partitioning-based orderings are also supported by means of external software packages such as METIS or SCOTCH; these tools are usually capable of providing better orderings (based on nested dissection) for large-scale, three-dimensional problems. They are also recommended in the parallel case because they lead to reasonably wide trees with more parallelism than the ordering options natively supported by MUMPS.

Note that after the assembly tree is built, there is still room to improve the memory behavior of the solver by computing and using a tree-traversal that is optimal in terms of memory usage. This is done within MUMPS by using a variant of the algorithm proposed by Liu [20].

In the early days of the development of parallel sparse direct codes, little attention was paid to the analysis step because of its low cost relative to the factorization and solve phases. However, for two reasons this has become a more major concern in recent years. One reason is that the efficient parallelization of the factorize step means that if the analysis is still run on one processor then the elapsed time for this can be significant or even dominant if many processors are used for the factorization. The second reason is that on some multiprocessor machines there is insufficient memory on a single processor to hold even the integer information for the whole matrix.

Thus MUMPS now offers a parallel option for the analysis step [6]. After an initial redistribution of the matrix, a parallel, nested-dissection-based tool, such as PT-SCOTCH or ParMETIS, is used to compute a fill-reducing pivotal order. According to the partitioning induced by the separators computed during the ordering step, the subsequent symbolic factorization is executed in parallel through an algorithm based on the use of distributed quotient graphs. This approach proved to be very effective in reducing the memory requirements and in improving the elapsed time for the analysis phase.

Most of the analysis step requires only the pattern of the matrix. However, scaling is a preprocessing for improving some of the numerical properties of the matrix, and this requires also the numerical values. Among a number of scaling alternatives available in MUMPS, the following two are the most powerful ones. The first uses linear programming duality where the scaling operators are obtained from weighted bipartite matching algorithms. The computation of this scaling alternative is performed sequentially on a central processor. A particular example of this is in the indefinite case where scaling can be used to identify potential two by two pivots allowing the analysis to be continued on a reduced graph obtained by collapsing the nodes corresponding to those pivots [12, 13]. The second scaling is an iterative algorithm. At each iteration of this algorithm, each row and column of the current matrix is scaled so that it is normalized according to a predefined norm, where the scaling of the rows and the scaling of the columns are independent. The scaling for this transformation is then combined with the scaling at the previous iteration in order to obtain a scaling for the original matrix. This algorithm is implemented in MUMPS for distributed memory environments in such a way that the communication and computation requirements of an iteration are equivalent to the sum of those for sparse matrix-vector and sparse matrix transpose-vector multiplication operations with the same matrix.

### Factorization Step

The factorization step uses the tree and subsidiary information from the analysis to effect the numerical factorization. As mentioned earlier, a main feature of this phase is the dynamic scheduling that is used to balance work and storage at execution time when the initial mapping is not appropriate because of numerical or load balance considerations [8, 10]. So that each processor can take appropriate dynamic scheduling decisions, an up to date view of the load and current memory usage of all the other processors must be maintained. This is done using asynchronous messages, as described in [17]. The nodes of the tree can be processed on a single processor or can be split between several processors. In MUMPS, a one-dimensional splitting and mapping can be used within intermediate nodes and a two-dimensional splitting can be used at the root node. The root node is by default factorized using ScaLAPACK.

In the factorization step, before the actual numerical factorization, there are a range of options available to scale the matrix including a version which runs in parallel. The most important scaling options were described when discussing the analysis step, but it is interesting to perform scaling at the factorization step rather than at the analysis step when successive factorizations are performed on matrices that share the same structure.

In addition to using standard threshold pivoting (see article on the ►Multifrontal Method in this Encyclopedia) for indefinite and unsymmetric systems, there is an option to use a static pivoting approach where pivots that are too small to satisfy a threshold criterion are augmented to avoid instability. In this case, one can avoid the perturbations to the original analysis ordering caused by numerical pivoting, making the performance closer to that obtained for positive definite matrices. However, the factorization will be that of a perturbed matrix and so an iterative refinement method or other more powerful iterative method must later be used to obtain a satisfactory solution.

The use of threshold partial pivoting may lead to a dynamic modification of the size of the tasks that have to be processed. Indeed, pivots which, for reasons of numerical stability, cannot be eliminated from the frontal matrix at a given node of the assembly tree are delayed to the parent node inducing an increase in its size and cost. This issue requires complex memory management mechanisms dealing with dynamic data structures. In addition, the fact that the cost of the tasks may differ from the predictions made during the analysis phase requires the scheduling mechanisms to dynamically adapt the mapping. This is done within MUMPS by using dynamic schedulers which complete the partial scheduling produced at the analysis phase (see [10] for more details). This enables the code to react to load variations due to delayed pivots.

In addition to a standard factorization of the matrix as in Eqs. 2 or 3, other options are available in MUMPS. It is possible to request a partial factorization of the matrix in order to compute the Schur complement. Consider a matrix ordered and partitioned as

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

and assume that the elimination is restricted to choose the pivots from the (1,1) block. Then the factorization

will proceed to compute the  $LU$  (or  $LDL^T$ ) factorization of  $A_{11}$  and to compute the Schur complement matrix  $A_{22} - A_{21}A_{11}^{-1}A_{12}$  which can then be returned to the user, either centralized on one processor, or distributed over the processors. This partial factorization feature is particularly powerful for problems where partitions of the above form are used, for example when using domain decomposition. Options exist to solve the whole system after the Schur complement system has been solved.

There is also an option to detect null or almost null pivots and compute the numerical rank of the matrix. In cases where the matrix is rank deficient, vectors from the null space can be returned at the solve step.

One of the main issues that commonly arises during the factorization of large scale problems concerns the amount of main memory required. Although parallelism represents a natural way of reducing the local memory requirements, this may still not be enough. Operating systems provide mechanisms that can handle transparently cases where an application requires more memory than available on the system. These techniques, commonly referred to as *virtual-memory* techniques, consist in extending the main memory using disk storage. Portions of data in the main memory, so-called pages, are swapped out of main memory to the disk when they are unlikely to be used soon (for example, based on a least recently used policy). They are brought back to main memory if later access to them is required. Although these techniques apply transparently to any application, relying on them normally results in considerable performance loss, since the swapping of pages is done according to general policies that cannot fully exploit the access pattern to data in an application.

Instead of this mechanism, MUMPS provides an *out-of-core* feature [2] that greatly reduces main memory consumption by saving the factors on disks. Although not available in the current version of MUMPS, specific strategies for the traversal of the elimination tree could be used in order to limit the amount of data moved to and from the disk [3]. Also because of a clever use of asynchronous I/O operations which allow an overlap of computations with disk accesses, the out-of-core feature only has a relatively small impact on the performance of the factorization phase while delivering great benefits in terms of memory consumption. The numerical robustness provided by partial pivoting

used in MUMPS is fully guaranteed even when the out-of-core feature is enabled.

### Solve Step

The solution phase, that is the forward and back substitution using the matrix factors, also exploits parallelism relying on the computational pattern defined by the elimination tree. Performance gains can also be achieved when there are multiple right-hand sides. The resulting solution vector(s) can be left as a distributed vector or can be returned as a centralized vector. If the matrix is unsymmetric, the solution of the transpose equations can be obtained using the same factorization.

Considerable performance gains can be achieved in MUMPS when sparse (perhaps multiple) right-hand sides are supplied since the solution phase only requires parts of the factors. In fact, nonzero values in the right-hand sides induce paths in the elimination tree which identify the portions of the factors involved in the solution phase. In the case of multiple, sparse right-hand sides, MUMPS uses grouping strategies in order to optimize the exploitation of such paths. In an out-of-core context, the grouping of sparse right-hand sides aims to maximize the overlap of these paths which is equivalent to minimizing the amount of data loaded from disks, and thus the number of expensive I/O operations [4].

As mentioned above, the solve step of MUMPS also provides an option for returning either all or specific vectors from the null-space basis.

### Postprocessing Step

In common with many of its precursors, MUMPS offers further information on the solution after it has been computed. Options exist to return information on the residuals, the condition number of the matrix, and an estimate of the error and iterative refinement can be used to improve the solution by solving equations of the form

$$A\delta x = r$$

where  $r$  is the residual

$$r = b - Ax$$

and then correcting the solution by  $\delta x$ .

### Origins of MUMPS

MUMPS started in 1996 with the PARASOL Project which was an Esprit IV European project. MUMPS

implements a multifrontal method (see the ► [Related Entry](#)) and was inspired by an experimental prototype of an unsymmetric multifrontal code for distributed-memory machines using PVM [15]. That experimental prototype was itself inspired by the code MA41, developed by Amestoy during his Ph.D. thesis [5] at CERFACS under the supervision of Duff. MA41 exploits vector computers and shared-memory multiprocessor environments to solve unsymmetric linear systems. Going back further in the past, MA41 started from the work of Duff, who designed a parallel version of the unsymmetric code from Duff and Reid (HSL code MA37 [14] developed at the Harwell Laboratory). This parallel code was run primarily on the Alliant FX/8 [11].

The PARASOL Project consisted of a consortium of six software providers that included teams at CERFACS, INPT(ENSEEIHT)-IRIT, and RAL, and four application-based users of the software. MUMPS, so named because of the target of massive parallelism using the MPI message passing protocol, was the sole direct code in the project which also included linear solvers based on domain decomposition (including a FETI implementation) and multigrid. The first functionalities of MUMPS were developed to meet the requirements of the industrial partners most of whom worked with finite-element models [7]. All the software produced by this project was in the public domain although by the end of the project, in September 1999, most were still almost at a prototype level.

Since the first prototype of MUMPS at the end of the PARASOL project (1999), MUMPS has been supported by research institutions (CERFACS, CNRS, INPT-IRIT and INRIA) and projects (academic or industrial), that have helped to develop new functionalities over the years. This development of the MUMPS package (including development of new features, maintenance, integration of research work, support to users and project management) has been led by the teams of Patrick Amestoy (INPT) at IRIT Toulouse and Jean-Yves L'Excellent (INRIA) at LIP-ENS Lyon. Some developments also involve members of the LaBri laboratory in Bordeaux, France, and of the French research institute CNRS, while CERFACS has contributed by financing research related activities through Ph.D. grants. Both postdoctoral and engineer positions have been directly financed by CNRS and INRIA, and MUMPS

has also been supported by a large number of contracts, projects and grants including

- International projects like the France-Berkeley Fund in collaboration with the University of California, Berkeley or the Egide-Aurora project in collaboration with the University of Bergen, Norway or the Multicomputing project in collaboration with the University of Tel Aviv, Israel
- Projects financed by research institutions like CNRS, INRIA or ANR like the ANR-SOLSTICE project
- Contracts financed by private institutions and industries such as Samtech

### Further Comments

MUMPS is both a software platform and a research project that has developed many new and innovative techniques for the parallel solution of large sparse linear systems. Several Ph.D. theses have been obtained by students directly connected to MUMPS [1, 16, 21, 22]. Some of the major current research interests include further work on out-of-core implementation, on memory scalability, on sparse right-hand sides, and on combinatorial aspects related to preprocessing and analysis. One way in which this stimulating symbiosis between code development and research can be seen is in the restricted availability of prototype versions of the code and the use of experimental parameters which both help the research efforts in addition to acting as a beta-test for new features.

MUMPS is coded in Fortran 90 and C and uses MPI for message passing. It calls standard codes from the BLAS, BLACS, and ScaLAPACK so that some additional fine multi-threading can be obtained by using appropriate versions of the BLAS.

Single and double precision versions are supported for both real and complex systems.

MUMPS supports all variants of UNIX systems including Linux, and Microsoft Windows thanks to a port developed by MUMPS users.

There are interfaces to MUMPS from Fortran 90, C, C++, MATLAB and SCILAB. The code will run on 64-bit architectures and exploits long integers that are becoming increasingly needed as the size of problems being solved by direct solvers continues to increase.

The MUMPS software is continually updated both to correct any bugs and also to add new functionality.

The current version (as of November 2009) is Release 4.9.2, which is in the public domain. Up-to-date information including the MUMPS Users' Guide and information on downloading the package can be obtained through the web pages <http://graal.ens-lyon.fr/MUMPS> or <http://mumps.enseeih.t.fr/>. MUMPS is currently downloaded by approximately a 1,000 users per year from these web sites. MUMPS is interfaced or included (and redistributed) within various commercial and academic packages such as Samcef from Samtech, FEM-Town from Free Field Technologies, *Code\_Aster* from EDF, IPOPT, PETSc (see the ►PETSc (Portable, Extensible Toolkit for Scientific Computation) entry in this Encyclopedia).

## Related Entries

- [Dense Linear System Solvers](#)
- [Linear Algebra Software](#)
- [Linear Algebra, Numerical](#)
- [Multifrontal Method](#)
- [PARDISO](#)
- [PETSc \(Portable, Extensible Toolkit for Scientific Computation\)](#)
- [SuperLU](#)

## Bibliography

1. Agullo E (2008) On the out-of-core factorization of large sparse matrices. PhD thesis, École Normale Supérieure de Lyon, France, November 2008
2. Agullo E, Guermouche A, L'Excellent J-Y (2008) A parallel out-of-core multifrontal method: storage of factors on disk and analysis of models for an out-of-core active memory. *Parallel Comput* 34(6–8):296–317
3. Agullo E, Guermouche A, L'Excellent J-Y (2009) Reducing the I/O volume in sparse out-of-core multifrontal methods. *SIAM J Sci Comput*, to appear
4. Amestoy P, Duff I, Guermouche A, Slavova T (2009) Analysis of the solution phase of a parallel multifrontal approach. *Parallel Comput* 2009. doi: 10.1016/j.parco.2009.06.001
5. Amestoy PR (1991) Factorization of large sparse matrices based on a multifrontal approach in a multiprocessor environment. INPT PhD thesis TH/PA/91/2, CERFACS, Toulouse, France
6. Amestoy PR, Buttari A, L'Excellent J-Y (2008) Towards a parallel analysis phase for a multifrontal sparse solver. June 2008. Presentation at the 5th International workshop on Parallel Matrix Algorithms and Applications (PMAA'08)
7. Amestoy PR, Duff IS, L'Excellent J-Y (1998) Multifrontal solvers within the PARASOL environment. In: Kågström B, Dongarra J, Elmroth E, Waśniewski J (eds) *Applied parallel computing, PARA'98*, Lecture Notes in Computer Science, No. 1541, pp 7–11, 1998. Springer, Berlin
8. Amestoy PR, Duff IS, L'Excellent J-Y, Koster J (2001) A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J Matrix Anal A* 23(1):15–41
9. Amestoy PR, Duff IS, Vömel C (2005) Task scheduling in an asynchronous distributed memory multifrontal solver. *SIAM J Matrix Anal A* 26:544–565
10. Amestoy PR, Guermouche A, L'Excellent J-Y, Pralet S (2006) Hybrid scheduling for the parallel solution of linear systems. *Parallel Comput* 32(2):136–156
11. Duff IS (1986) Parallel implementation of multifrontal schemes. *Parallel Comput* 3:193–204
12. Duff IS, Pralet S (2007) Strategies for scaling and pivoting for sparse symmetric indefinite problems. *SIAM J Matrix Anal A* 27(2):313–340
13. Duff IS, Pralet S (2007) Towards stable mixed pivoting strategies for the sequential and parallel solution of sparse symmetric indefinite systems. *SIAM J Matrix Anal A* 29(3): 1007–1024
14. Duff IS, Reid JK (1984) The multifrontal solution of unsymmetric sets of linear systems. *SIAM J Sci Stat Comput* 5:633–641
15. Espirat V (1996) Développement d'une approche multifrontale pour machines à mémoire distribuée et réseau hétérogène de stations de travail. Technical Report Master thesis, ENSEEIHT-IRIT
16. Guermouche A (2004) Études et optimisation du comportement mémoire dans les méthodes parallèles de factorisation de matrices creuses. PhD thesis, ENS Lyon, France
17. Guermouche A, L'Excellent J-Y (2005) A study of various load information exchange mechanisms for a distributed application using dynamic scheduling. In: 19th International Parallel and Distributed Processing Symposium (IPDPS'05)
18. Guermouche A, L'Excellent J-Y (2006) Constructing memoryminimizing schedules for multifrontal methods. *ACM T Math Software* 32(1):17–32
19. Guermouche A, L'Excellent J-Y, Utard G (2003) Impact of reordering on the memory of a multifrontal solver. *Parallel Comput* 29(9):1191–1218
20. Liu JWH (1986) On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM T Math Software* 12(3):249–264
21. Pralet S (2004) Constrained orderings and scheduling for parallel sparse linear algebra. Phd thesis, Institut National Polytechnique de Toulouse, September 2004. CERFACS Technical Report, TH/PA/04/105
22. Slavova Tz (2009) Parallel triangular solution in the out-of-core multifrontal approach for solving large sparse linear system. PhD thesis, Institut National Polytechnique de Toulouse, 2009. CERFACS Technical Report, TH/PA/09/59

## Mutual Exclusion

- ▶ [Path Expressions](#)
- ▶ [Synchronization](#)

## Myri-10G

- ▶ [Myrinet](#)

## Myricom

- ▶ [Myrinet](#)

## Myrinet

SCOTT PAKIN  
Los Alamos National Laboratory, Los Alamos, NM, USA

### Synonyms

[Interconnection network](#); [LANai](#); [Myri-10G](#); [Myricom](#); [Network architecture](#)

### Definition

Myrinet is a family of high-performance network interface cards (NICs) and network switches produced by Myricom, Inc. (Myrinet® and Myricom® are registered trademarks of Myricom, Inc.) Some salient features of Myrinet are that it utilizes wormhole switching and emphasizes fully programmable NICs and simple switches.

### Discussion

#### History

The Myrinet network [3] is a direct descendent of Caltech's Mosaic C multicomputer [11] and USC/ISI's ATOMIC local-area network (LAN) [6]. The Mosaic C was based on custom processors that integrated a network interface, router, and simple CPU onto a single die. Although the Mosaic C processor was intended to be used as the building block for a massively

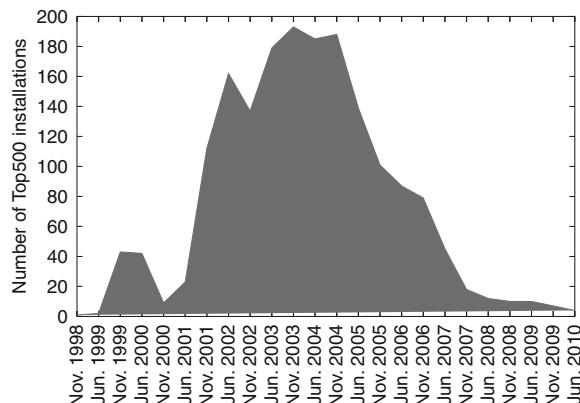
parallel processor, the ATOMIC project investigated using Mosaic C processors in a LAN setting as intelligent NICs – protocol processors – as a means for providing high-speed communication to nodes with more powerful CPUs. The project was deemed a success, and members of the Caltech and USC/ISI teams formed Myricom, Inc. to enhance and commercialize the technology.

The first Myrinet product was released in 1994 and utilized separate NICs and switches, in contrast to the Mosaic C's integrated system. The NICs initially comprised a 16-bit microcoded processor called the LANai and 128 KiB of static random-access memory (SRAM). As in ATOMIC, the NICs resided on an I/O bus. The switches were based on 4- or 8-port crossbars. Connectivity was via copper cables, and the link data rate was 640 Mb/s (76.3 MiB/s) in each direction.

The second-generation Myrinet network (1996) replaced the 16-bit microcoded LANai 2.3 with a 32-bit pipelined LANai 4, increased the NIC's on-board SRAM to 4 MiB, made some physical changes to the cabling (supporting both "LAN" and "SAN" configurations and both copper and optical cables) and to the signaling and encoding mechanisms, doubled the link data rate to 1,280 Mb/s (1.2 GiB/s) in each direction, and added support for the PCI bus. 16-port Myrinet switches became available. During Myrinet's second generation, the LANai processor went through numerous performance enhancements and major revisions, ending with the LANai 9, whose core is still in use in recent Myrinet products.

The third-generation product, Myrinet 2000, appeared in 2000 and further increased the data rate to 2,000 Mb/s (1.9 GiB/s) in each direction. Within that same generation, Myricom began supplying a "converged" architecture that continued to support the extant Myrinet cables and protocols while adding support for Gigabit Ethernet cables and protocols. This required altering most parts of the LANai (most noteworthy excluding the processor proper) to support multiple interfaces, resulting in the LANai XM, XP, and 2XP, each with a different tally of host and external interfaces. Support for the PCI-X I/O bus was also added.

Finally, the fourth-generation Myrinet product, Myri-10G, increased the data rate to 10,000 Mb/s (9.3 GiB/s), added support for the PCI Express bus,



**Myrinet. Fig. 1** Myrinet's popularity in the HPC community

and upgraded the convergence products from Gigabit Ethernet to 10 Gigabit Ethernet via new LANai Z8E, 2ZS, and Z8ES implementations.

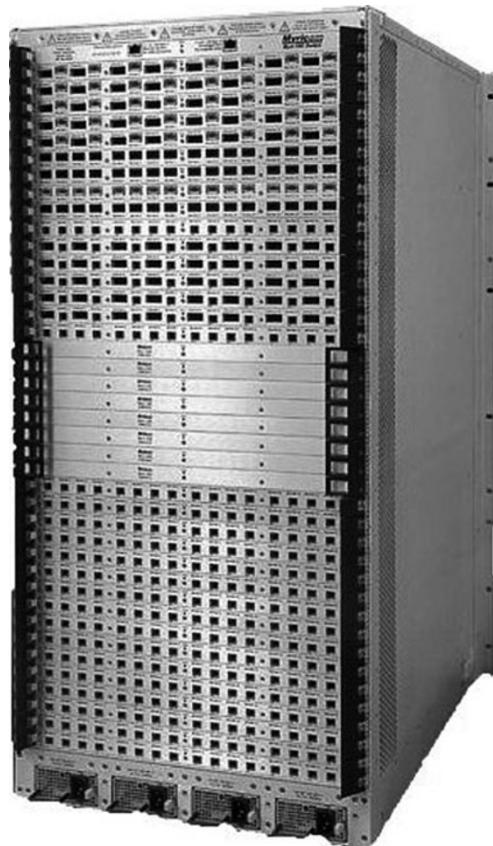
According to the Top500 list (<http://www.top500.org/>), Myrinet's popularity in the high-performance computing (HPC) community peaked in November 2003 with 193 of the world's 500 fastest supercomputers using a Myrinet network. However, as the Top500 data show, Myrinet's presence in the Top500 list dropped rapidly starting in 2005 (Fig. 1). At the time of this writing (June 2010), InfiniBand and Gigabit Ethernet have largely cornered the HPC interconnect market.

## Hardware

A Myrinet network consists of NICs and switches. Each computer on the network contains a NIC (possibly more than one). NICs are responsible for transferring data (bidirectionally) between main memory and the nearest switch. Switches are responsible for transferring data from their input ports to their output ports, where a port can be connected to either a NIC or another switch. Within a chassis, switches are connected to each other via a backplane, and between chassis, switches are connected via cables.

## Switches

Each individual switch contains some routing logic and a crossbar (a 32-port crossbar in the Myri-10G product). In principle, switches support arbitrary network topologies. That is, as far as the switch hardware is concerned, any switch can be connected to any other switch or NIC. This was originally convenient for local area network (LAN) settings, in which desktop computers



**Myrinet. Fig. 2** A fully populated 512-port Myri-10G enclosure (reproduced with permission from [http://www.myri.com/Myri-10G/switch/configuration\\_guide.html](http://www.myri.com/Myri-10G/switch/configuration_guide.html))

could simply be connected to the nearest switch without concern for topology. Today, Myrinet switches come in integrated chassis pre-wired into a regular, folded-Clos topology. In an HPC environment, multiple Myrinet chassis are usually also networked into a folded-Clos topology to provide high bisection bandwidth. Figure 2 shows a fully populated 512-port Myri-10G enclosure, which measures 21U (approximately 37 in. or 93 cm) in height. This unit contains a total of 48 switches, with 32 exposing half of their ports externally and routing the other half to a row of 16 all-internal “spine” switches.

Myrinet switches employ wormhole switching, which is fairly uncommon in cluster networks, although the technique had previously been used in a number of massively parallel processors (MPPs) such as the nCUBE3, Intel Paragon, and Cray T3D. Wormhole switching means that switches do not need to buffer an entire incoming packet before sending it out. Rather, as

each flit (*flow control digit* – one 8-bit byte in Myrinet) arrives at a switch, it can immediately be sent downstream. In fact, a packet’s first flit can reach the destination NIC before the source NIC has transmitted the packet’s final flit.

## Network Protocols

ANSI/VITA 26-1998 [4], the ANSI standard that defines Myrinet, specifies the network’s behavior at various layers of the Open System Interconnection (OSI) communication model. At the OSI data-link layer, Myrinet employs link-level flow control to prevent upstream NICs or switches from overwhelming a downstream NIC or switch. For example, if two packets arriving at a switch target the same output port, one packet will be allowed to proceed while the other is delayed. The switch sends a STOP symbol to the sender to instruct it to stop sending data. When the target port is free, the switch then sends a Go symbol to allow the sender to resume transmission. The other data-link-layer symbols are GAP and BEAT. GAP indicates that no data is being sent and that the link is between packets. The next data byte to arrive after a GAP is therefore the first byte of a new packet. GAP symbols obviate the need to specify a maximum transmission unit (MTU) size in hardware. Unlike most other networks, Myrinet allows packet frames to be arbitrarily large, although ANSI/VITA 26-1998 allows implementations to limit MTU sizes – but to no less than 4 MiB [4]. BEAT is a heartbeat symbol sent every 10 µs to indicate that data are being transmitted correctly (e.g., that the sender is not stuck repeatedly sending the same value).

At the OSI network layer, Myrinet packets are defined fairly simply. Each packet contains a number of routing bytes followed by a number of data bytes, followed by an 8-bit cyclic redundancy check (CRC-8) byte. Myrinet is a source-routed network, which means that the NIC that initiates a packet specifies the complete route to the destination NIC. Each packet begins with one routing byte per intermediate switch. Because there can be any number of switches between NICs, the number of routing bytes in a packet varies accordingly. In early versions of Myrinet, each routing byte specified an absolute port number. For example, a routing byte of {3} means that the packet should exit the subsequent switch from port 3. The network-layer protocol was later modified to use relative port numbers, which can be positive or negative. For example, a routing byte of {3}

means that if the packet enters a switch on port 4, it should exit on port 7. The advantage of relative addressing is that reverse routes are easy to calculate; a NIC needs only to invert the signs and reverse the order of the bytes. For example, if NIC A sends a packet to NIC B through three intermediate switches using routing bytes {+2, -4, +3}, then NIC B can send a packet to NIC A through the same three switches using routing bytes {-3, +4, -2}. Each switch discards the routing byte at the start of the packet and forwards the remaining bytes downstream. When the packet reaches its destination NIC it is not preceded by any routing bytes.

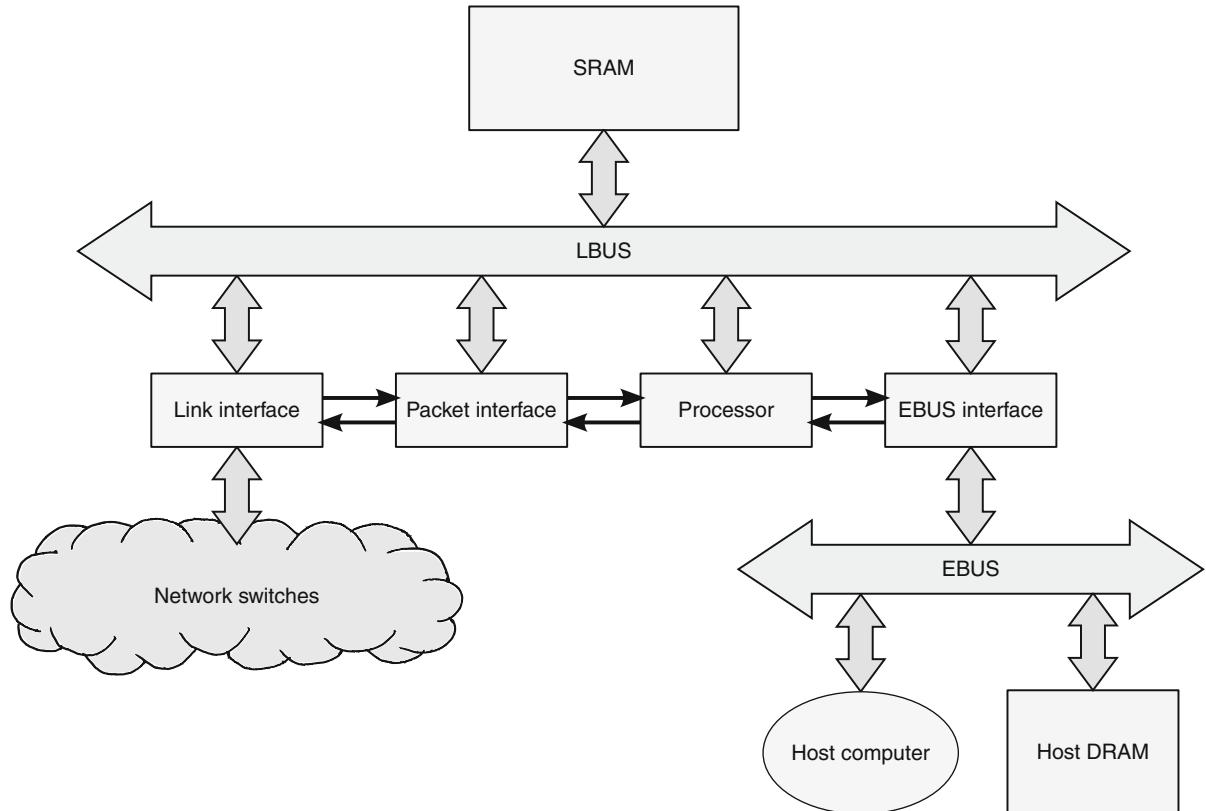
The packet’s data payload follows the routing bytes. The data begin with a four-byte packet type (two bytes for the primary type, two bytes for the secondary type) to indicate to the destination NIC what software layer is expected to handle the packet. For example, packet type 700F<sub>H</sub> indicates a topology-mapping packet used by the MX messaging layer [9], and packet types 0200<sub>H</sub>–0207<sub>H</sub> are used for various purposes by the Illinois Fast Messages messaging layer [10]. While the use of packet types is technically not required by the hardware, ANSI/VITA 26-1998 dictates its use to prevent software from mistaking one higher-level protocol’s packets for another’s. Myricom maintains a registry of packet types, and customers can request values dedicated to their own messaging layers.

The final byte of a Myrinet packet is a CRC-8 that covers the entire packet (routing bytes and data payload). It is inserted by the NIC that initiates the packet and is checked by each switch on the routing path and by the destination NIC. Because each switch modifies the packet by removing a header byte, each switch replaces the incoming CRC-8 byte with a new CRC-8 byte while sending the packet downstream.

## Network Interface Cards

One of Myrinet’s design philosophies is to keep the switches simple and fast and put all of the communication “smarts” into the endpoints. Hence, a Myrinet NIC is not a passive device controlled by the host computer’s operating system, as was typical for local area networks when Myrinet was first introduced, but rather an active entity that can autonomously manage data transfers between the network and host memory.

The basic NIC architecture is illustrated in Fig. 3. The main units shown in that figure are a link interface, which connects the NIC to an external switch; a



**Myrinet. Fig. 3** Block diagram of a Myrinet NIC

packet interface, which manages the transfer of packets to and from the network; a custom processor, which controls all aspects of the NIC; an EBUS (External BUS) interface, which manages the transfer of data to and from the host's I/O bus (e.g., PCI Express); and a small quantity of high-speed, multiported SRAM connected directly via an LBUS (Local BUS). The link interface, packet interface, processor, and EBUS interface are integrated onto a single chip called the LANai. Throughout the generations of Myrinet, the link interface adapted to Myrinet's ever-faster link speeds; the packet interface began including support for network types other than Myrinet (viz. Ethernet and InfiniBand) and in some instances doubled to two packet interfaces per NIC; the processor changed from being 16-bit microcoded to being 32-bit pipelined and underwent various modifications to the instruction set and clocking; the EBUS interface tracked each new I/O bus technology; and the SRAM switched from asynchronous

to zero bus turnaround (ZBT) technology for a speed boost and also increased in size from 128 KiB (with 16 bits/byte) to 4 MiB (with 8 bits/byte) with a maximum address space of 8 MiB. All of those implementation changes notwithstanding, the basic NIC architecture represented by Fig. 3 has remained largely unchanged across all versions of Myrinet.

The LANai processor utilizes a typical reduced instruction set computer (RISC) instruction set with a few additional features to support its role in managing communication. There is no floating-point unit. Because the processor pipeline is non-interlocked and because all memory is constant-latency SRAM, it is relatively easy to reason about LANai performance: Exactly one instruction is issued every cycle.

The software running on the LANai, known as the Myrinet Control Program (MCP), generally works by initiating direct memory access (DMA) transfers between local SRAM and either host DRAM or the

network. LANai SRAM can be mapped into the host's address space, so programmed I/O – word-by-word writes from the host to LANai SRAM or word-by-word reads from LANai SRAM to the host – is also possible. The LANai provides various features in hardware to perform these tasks with minimal overhead. DMA transfers between host memory and LANai SRAM are performed by writing source and destination addresses into a set of special memory-mapped registers. These registers are also mapped into host memory, so the host can control parts of the NIC hardware independently from the LANai. Dedicated hardware in the EBUS interface can generate CRC-32 values of data as it is copied between host memory and LANai SRAM. DMA transfers between the network and LANai SRAM are performed by writing starting and ending addresses into special send and receive registers. Messages being sent can comprise multiple DMA requests; a “final ending address” register designates the end of a message. CRC-32 values can be computed on both incoming and outgoing data and can be injected into an outgoing message at a specified location.

As a simple example of data transmission, the following code is all that is needed to send a msglen-byte message from buffer message:

#### C code

```
SMP = (void *)message;
SMLT = (void *)&message[msglen];
```

#### LANai assembly code

```
! Put msglen's value in r7.
 ld [_msglen],%r7
! Put message's address in r6.
 mov _message,%r6
! r6 is the DMA's starting address.
 st %r6,[0xFFFFFEF0]
! Put message's address in r8.
 mov _message,%r8
! Point r6 just pass the message.
 add %r7,%r8,%r6
! Initiate the send.
 st %r6,[0xFFFFFFF08]
```

Address FFFFFEF0<sub>H</sub> corresponds to special register Send-Message Pointer (SMP), which points to the start of the message. Address FFFFFF08<sub>H</sub> corresponds to special register Send-Message Limit, with the Tail (SMLT), which points to one byte past the last byte of the message. Storing a value in the SMLT register triggers the DMA from LANai SRAM into the network. Note that the preceding code assumes that the message begins with the requisite routing and packet-type bytes. Waiting for send completion is simply a matter of polling the interrupt status register, ISR, until the SEND\_INT bit is set:

#### C code

```
while ((ISR & SEND_INT_BIT) == 0)
;
```

#### LANai assembly code

```
! Store SEND_INT's bit offset in r13.
 mov 8,%r13
! Load ISR into r6.
L3: ld [0xFFFFFE50],%r6
! Delay until r6 is ready.
 nop
! Delay until r6 is ready.
 nop
! Test r6's SEND_INT bit.
 and.f %r6,%r13,%r0
! Try again if not set.
 beq L3
```

The nop instructions are necessary because the LANai has a non-interlocked processor pipeline.

The LANai supports two execution contexts, “user” and “system,” each with its own program counter and other registers. Context switching is extremely fast and can be performed either synchronously via an explicit machine instruction (punt) or asynchronously from user to system context in response to a hardware interrupt. The LANai signifies DMA completion, message availability, timer wraparound, and various other conditions by setting a bit in the ISR. The MCP can choose to poll the ISR for status changes or request that

modifications to certain bits trigger an interrupt, which the system context can subsequently handle.

## Software

### Communication-Layer Design Issues

Because the LANai is fully programmable and development tools are readily available, it is possible to implement any number of low-level communication layers. These can provide different feature sets, different semantics, and make different performance tradeoffs. In fact, there have been a wide variety of low-level, software communication layers developed over the years not only by Myricom – the so-called Myrinet API followed by GM (Glenn’s Messaging) followed by MX (Myrinet Express) – but also by researchers at numerous institutions. Research communication layers targeting Myrinet include Active Messages (AM), the Basic Interface for Parallelism (BIP), the BullDog Myrinet Control Program (BDM), Fast Messages (FM), Hamlyn, Link-level Flow Control (LFC), ParaStation2, PM, Trapeze, U-Net, Virtual Memory-Mapped Communication (VMMC), as well as the Message-Passing Interface (MPI) implemented atop various Myrinet-specific MPICH channel devices. All of those are designed to be user-level communication layers, meaning that the operating system has essentially no involvement in the critical path of communication. However, traditional, operating-system-based Internet Protocol (IP) communication can run atop Myricom’s low-level communication layers. Each of the communication layers listed above must address issues of memory pinning, data transfer, reliability, and route discovery, among others.

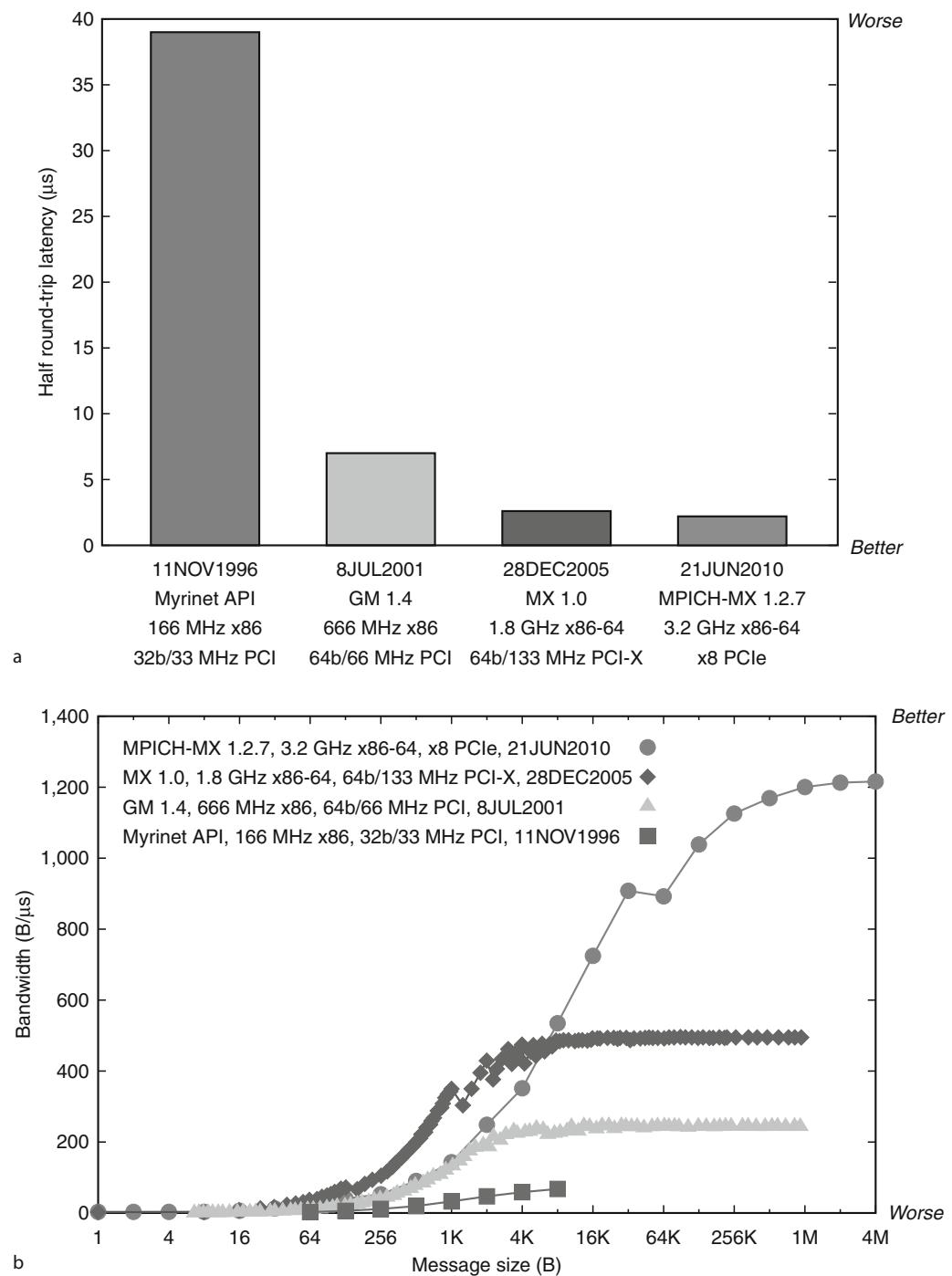
The LANai can transfer data only to and from physical addresses in the host. Consequently, the host operating system must ensure that memory pages that may be accessed by the LANai are “pinned,” making them ineligible for being swapped out to disk and replaced by another page. Some communication layers have the operating system pin a fixed region of addresses at initialization time, let the LANai transfer data only to/from that region, and use host software to copy data to/from that region into arbitrary (non-pinned) buffers provided by the application. Other communication layers modify the operating system to respond to LANai interrupts, pin pages on demand, and notify the MCP of the physical addresses to use.

LANai-initiated data transfers to/from the host must be performed using the LANai’s DMA mechanism. However, host-initiated data transfers can use either DMA or programmed I/O (word-by-word reads or writes). DMA-based transfers tend to have a higher startup cost but lower per-byte cost while programmed I/O-based transfers tend to have a lower startup cost but a higher per-byte cost. Programmed I/O-based transfers can address arbitrary host memory and can therefore avoid copying data to/from a pinned region or asking the operating system to pin a region of memory before the transfer begins.

Related to data transfers is event notification: Is data ready to be processed? The LANai and the host can each signal the other with interrupts, which are asynchronous but, particularly in the case of the LANai interrupting the host, exhibit high overhead due to operating-system involvement and costly context switches. The alternative is to poll memory for state changes. While a single poll is faster than a single interrupt, polling must be performed frequently to avoid delaying communication progress.

Myrinet provides reliable data transmission in the sense that it never intentionally discards data. However, software communication layers must ensure that there is always a place to store incoming data. Most communication layers implement some form of flow control to prevent data buffers from overflowing. However, some simply discard incoming data if there is no place left to store it and let higher layers of software or the application itself detect the data loss and induce a retransmission.

Because Myrinet is a source-routed network, an MCP needs to know the precise path to every NIC in the cluster so it can inject the correct routing bytes into the network. Some communication layers simply read a configuration file that enumerates these paths and instruct the MCP accordingly. This works well for smaller networks with fairly static topologies. Other communication layers implement route discovery. In Myrinet, route discovery has to be performed entirely by software, either by the MCP or by the host with the MCP’s assistance. Route discovery works essentially by performing a breadth-first search of the network: First, all possible one-hop destinations are queried for existence, followed by all possible two-hop destinations, and so forth. Because this can be a time-consuming



**Myrinet.** Fig. 4 Myrinet historical performance (a) latency, (b) bandwidth

operation for large networks, route discovery is performed only occasionally.

## Performance

[Figure 4](#) aggregates historical performance data that have appeared on [Myricom's Web site](#) over a span of 14 years. [Figure 4a](#) presents messaging latency, measured as half the average time for a minimal-sized message to be sent back and forth between a host process on each of two computers. [Figure 4b](#) presents the messaging bandwidth, measured as the average rate at which a message of a given size can be sent back and forth between a host process on each of two computers.

The data represent multiple generations of low-level Myrinet software, Myrinet NIC hardware, Myrinet switch hardware, host CPUs, and I/O buses. Consequently, performance improvements cannot be attributed to the evolution of a single component. Rather, [Fig. 4](#) should be interpreted as portraying the performance that one could have expected from a Myrinet cluster with typical hardware and typical vendor-supplied software at four points in time from November 1996 to June 2010. As [Fig. 4](#) shows, a cluster running a recent version of MPICH-MX on recent Myrinet hardware can expect to see a minimum of 2.2 µs one-way message latency and a maximum of 1,215 MB/s of communication bandwidth – both a factor of 18 improvement over a state-of-the-art Myrinet cluster from 14 years earlier.

The data presented in [Fig. 4](#) represent only Myricom's software. Myrinet software from third-party research institutions often outperformed the corresponding offering from Myricom, generally by utilizing novel data structures and by removing nonessential, high-overhead features. As a point of comparison, contemporaneously with the Myrinet API observing a minimum latency of 39 µs, Fast Messages, Active Messages, and PM all observed a minimum latency of 10 µs or less. In fact, some of Myrinet's uniqueness from a software perspective is that the LANai's programmability makes Myrinet a suitable platform for experimentation with communication-layer design.

## Related Entries

- [Cray XT4 and Seastar 3-D Torus Interconnect](#)
- [Ethernet](#)
- [Flow Control](#)

- [InfiniBand](#)
- [Interconnection Networks](#)
- [Network Interfaces](#)
- [Network of Workstations](#)
- [Networks, Direct](#)
- [Networks, Multistage](#)
- [Quadratics](#)
- [SCI \(Scalable Coherent Interface\)](#)
- [Switch Architecture](#)
- [Switching Techniques](#)

## Bibliographic Notes and Further Reading

One of Myricom's few peer-reviewed publications is an overview of Myrinet that appeared in a 1995 issue of IEEE Micro [3]. Although the paper describes the first commercial implementation of Myrinet, much of what it says is still applicable to more recent incarnations of Myrinet. Details of Myrinet's OSI data-link layer and OSI network layer appear in the Myrinet standardization document, ANSI/VITA 26-1998 [4]. Various aspects of the LANai architecture and physical characteristics are covered by Myricom white papers [7, 8].

The LANai's origins are in Caltech's Mosaic C multicomputer [11]. Both contain a simple processor, a memory interface, and an integrated network interface with instruction-set support. However, the Mosaic C targets large-scale direct networks and therefore additionally integrates a dimension-order router for 2-D mesh networks onto the die. In contrast, the LANai targets indirect networks of arbitrary topology and therefore delegates routing to an external switching fabric. USC/ISI saw the potential for utilizing Mosaic C components in a LAN setting and constructed their ATOMIC LAN as a proof of the concept [6]. In a sense, Myrinet is a commercial implementation of the ATOMIC network architecture. An interesting part of Felderman et al.'s ATOMIC paper [6] is its exposition of the challenges of route discovery on a network topology that is not guaranteed to be either symmetric or consistent. (Myrinet networks are symmetric but not necessarily consistent.)

Myrinet has been the target platform for a wealth of software communication layers such as Fast Messages [10], Active Messages II [5], and Myrinet Express [9]. Two papers published in November 1998 contrast many of the Myrinet communication layers that

were current at that time. Bhoedjang et al. discuss the design decisions underlying user-level communication layers, step through a simple communication layer for Myrinet, and show how numerous Myrinet communication layers address various implementation alternatives [2]. Araki et al. take a more performance-oriented approach, presenting a LogP (latency, overhead, gap, processes) analysis of multiple Myrinet communication layers and drawing conclusions about how various communication-layer features impact communication time [1].

## Bibliography

1. Araki S, Bilas A, Dubnicki C, Edler J, Konishi K, Philbin J (1998) User-space communication: a quantitative study. In: Proceedings of the 1998 IEEE/ACM conference on supercomputing (SC'98), Orlando, Florida, 7–13 Nov 1998. ISBN: 0-8186-8707-X. doi: 10.1109/SC.1998.10038
2. Bhoedjang RAF, Rühl T, Bal HE (Novemebr 1998) User-level network interface protocols. *IEEE Comput* 31(11):53–60. ISSN: 0018-9162. doi: 10.1109/2.730737
3. Boden NJ, Cohen D, Felderman RE, Kulawik AE, Seitz CL, Seizovic JM, Su W-K (Februar 1995) Myrinet: a gigabit-per-second local area network. *IEEE Micro* 15(1):29–36. ISSN: 0272-1732. doi: 10.1109/40.342015
4. Chin G, Parsons E, Dumont JJ, Wright D, Sherfinski H, Robak D, Jaenicke R, Vadasz I, Thompson M, Blake M, Morgan R, Bratton J, Cohen D, Waggett J, McKee R, Munroe M, Peterson W, Endo D, Kwok J, Ryneanson J, Alderman R, Andreas H, Bedard J, Lavelly T, Patterson B (1998) Myrinet on VME protocol specification. ANSI Standard ANSI/VITA 26-1998, VITA Standards Organization, Scottsdale. November 2, 1998. A draft is available from <http://www.myri.com/open-specs/myri-vme-d11.pdf>
5. Chun BN, Mainwaring AM, Culler DE (January/February 1998) Virtual network transport protocols for Myrinet. *IEEE Micro* 18(1):53–63. ISSN: 0272-1732. doi: 10.1109/40.653035
6. Felderman R, DeSchon A, Cohen D, Finn G (1994) ATOMIC: a high speed local communication architecture. *J High Speed Netw* 3(1):1–28. ISSN: 0926-6801
7. Myricom, Inc. (2000) LANai 9. June 26, 2000. <http://www.myri.com/vlsi/LANai9.pdf>
8. Myricom, Inc. (2003) Lanai X. July 28, 2003. <http://www.myri.com/vlsi/LanaiX.Rev1.1.pdf>. Revision 1.1
9. Myricom, Inc. (2006) Myrinet Express (MX): a high-performance, low-level message-passing interface for Myrinet. October 1, 2006. <http://www.myri.com/scs/MX/doc/mx.pdf>. Version 1.2
10. Pakin S, Karamcheti V, Chien AA (April–June 1997) Fast messages: efficient, portable communication for workstation clusters and MPPs. *IEEE Concurr* 5(2):60–73. ISSN: 1092-3063. doi: 10.1109/4434.588295
11. Seitz CL, Boden NJ, Seizovic J, Su W-K (1993) The design of the Caltech Mosaic C multicomputer. In: Borriello G, Ebeling C (eds) Proceedings of the 1993 symposium on research on integrated systems, Seattle, Washington, 14–16 Mar 1993. MIT Press, pp 1–22. ISBN: 0-262-02357-1

# N

## NAMD (NAnoscale Molecular Dynamics)

LAXMIKANT V. KALÉ, ABHINAV BHATELE, ERIC J. BOHM,  
JAMES C. PHILLIPS  
University of Illinois at Urbana-Champaign, Urbana,  
IL, USA

### Definition

NAMD is a parallel molecular dynamics software for biomolecular simulations.

### Discussion

#### Introduction

NAMD (NAnoscale Molecular Dynamics, <http://www.ks.uiuc.edu/Research/namd>) is a parallel molecular dynamics (MD) code designed for high-performance simulation of large biomolecular systems [1–4]. Typical NAMD simulations include all-atom models of proteins, lipids, and/or nucleic acids as well as explicit solvent (water and ions) and range in size from 10,000 to 10,000,000 atoms.

NAMD employs the prioritized message-driven execution capabilities of the Charm++/Converse parallel runtime system (<http://charm.cs.illinois.edu>), allowing excellent parallel scaling on both massively parallel supercomputers and commodity workstation clusters. NAMD is distributed free of charge as both source code and pre-compiled binaries by the Theoretical and Computational Biophysics Group (<http://www.ks.uiuc.edu>) of the University of Illinois Beckman Institute. NAMD development is primarily funded by the NIH through the Resource for Macromolecular Modeling and Bioinformatics. NAMD has been downloaded by over 36,000 registered users, over 8,000 of whom have downloaded multiple releases.

#### Biomolecular Simulation

The interactions between atoms in a biomolecular system are modeled with a combination of classical (non-quantum) terms. Each atom is assigned a static partial charge (a fraction of an electronic charge) based on the difference between its nuclear charge and the nearby electron density given its molecular configuration. Standard  $1/r$  electrostatic interactions between these partial charges are added to a Lennard-Jones potential, which combines a  $1/r^6$  van der Waals attractive potential with a short-range  $1/r^{12}$  repulsive term. The effects of covalent bonds are represented by harmonic terms applied to the distance between directly bonded atoms, angles formed by a pair of bonds to a common atom, and dihedral angles formed by chains of four bonded angles. Dihedral terms may also be sums of cosines, and harmonic terms are applied to enforce planarity of four atoms in the case of so-called improper dihedrals. Parameters for the Lennard-Jones and bonded terms are based on the element type of each atom, as well as its bonds to hydrogen or other heavy atoms.

In preparing to simulate a biomolecular system with NAMD, atomic coordinates of the proteins, nucleic acids, and/or lipids of interest are obtained from known crystallographic or other structures. Missing parts of the structure may be modeled based on homologous molecules of known structure. Any aggregates must then be assembled, such as embedding a protein in a membrane or binding a hormone receptor to DNA. A periodic block of water is replicated to cover the entire structure up to the boundaries of the desired periodic cell, water molecules within a few Angstroms of the structure are removed, and the remaining water is merged into the structure. In order to neutralize the electrical charge of the system and obtain a proper salt concentration, some water molecules are replaced with  $\text{Na}^+$  and  $\text{Cl}^-$  ions in a manner that minimizes electrostatic energy.

Using the potential function, the initial atomic coordinates are adjusted to eliminate initial close contacts

and large forces that would destabilize the system. The Newtonian equations of motion, with modifications to control temperature and pressure, are then integrated by symplectic and reversible methods using a time step of 1 femtosecond (or 2 fs if water molecules and bonds to hydrogen atoms are held rigid). A typical simulation may be 1–10 ns (1–10 million steps), and long simulations may be a microsecond or more.

Although the symplectic property of the standard explicit Verlet integrator allows energy to be well conserved over long trajectories, a thermostat is typically used to initially equilibrate the simulation to a given temperature and then to control heating due to integrator energy drift. Molecular dynamics simulations are chaotic in the sense that the specific trajectory is highly dependent on the initial coordinates and velocities. Therefore, the goal of the simulations is to characterize typical conformations and fluctuations, and to extract thermodynamic properties such as free energy differences.

## NAMD Features

NAMD 2.x has been in production use for over a decade and has been developed in a highly active user environment. As a result of user requirements, NAMD has acquired a wide variety of simulation features. NAMD relies on the capabilities of its sister program VMD (Visual Molecular Dynamics, <http://www.ks.uiuc.edu/Research/vmd>) for the setup, visualization, and analysis of simulations. VMD reads and writes a wide variety of file formats and is used extensively for visualizing and analyzing the output of MD codes besides NAMD, as well as other experimental and computational data. VMD can connect to a running NAMD simulation, allowing the user to apply forces to steer the simulation using the mouse or a haptic interface.

NAMD supports non-periodic and periodic boundary conditions, including non-orthogonal cells and periodicity in only one or two dimensions. Minimization is performed using the conjugate gradient algorithm. Atoms may be fixed or restrained to an initial position to stabilize a structure during minimization and equilibration. Both three-point and four-point water models (TIP3P and TIP4P) are supported. Water molecules and bonds to hydrogen atoms may be held rigid to increase the maximum stable time step. Velocity

rescaling and Langevin dynamics constant temperature methods and Berendsen and Langevin piston constant pressure methods may be applied to provide an NVT or NPT ensemble.

Free energy of conformational change surfaces or potentials of mean force may be studied along an arbitrary number of user-defined collective variables via a variety of methods, including meta-dynamics, the adaptive biasing force method, umbrella sampling, and steered molecular dynamics. Alchemical transformations, in which the chemical structure of a molecule is modified, may be studied via free energy perturbation or thermodynamic integration methods.

Special support is provided for the analysis of cryo-electron-microscopy data via molecular dynamics flexible fitting. In this method, an experimentally derived electron density map is transformed into a potential on a grid. The molecular structure is restrained via additional bonded terms and simulated to allow the atoms to conform to the grid potential. This allows the combination of atomic-resolution x-ray crystallographic structures with lower-resolution cryo-electron-microscopy data and has been particularly useful in the study of large structures with complex motions such as the ribosome.

An important goal of NAMD is to provide a uniform user interface (as far as possible) across all platforms, from laptops and desktops to clusters and massively parallel Cray XT and IBM Blue Gene supercomputers. Within the limits of memory and scalability, any simulation can be run on any number of processors. The program's spatial decomposition method is adjusted from full-sized to half-sized patches in one, two, or three dimensions as the number of processors increases. Similarly, the particle-mesh Ewald FFT decomposition is switched from slabs to pencils and spanning trees are used for coordinate and force messages. While manual tuning options are available, they are only beneficial at the limits of parallel scalability – most users happily ignore them.

## Parallel Design and Implementation

The sequential algorithm in most molecular dynamics applications is similar. The simulation time is broken down into a large number of very small time steps (typically 1 or 2 fs each). At each time step, forces on every

atom are calculated due to every other atom. Acceleration and velocities for the particles are obtained from the forces, based on which the new positions of the particles are calculated. This is repeated every time step. Two kinds of forces are to be calculated: (1) forces due to bonds and (2) non-bonded forces. If the nonbonded forces are calculated pairwise for all pairs of atoms, it results in a naïve  $\mathcal{O}(n^2)$  implementation. Most MD applications use an optimization where forces are not calculated explicitly beyond a certain cutoff distance. For atoms that lie beyond the cutoff distance, forces are calculated by using the particle-mesh Ewald (PME) or Gaussian spread Ewald (k-GSE) method. In these methods, the electric charge of each atom is transferred to electric potential on a grid that is used to calculate the influence of all atoms on each atom.

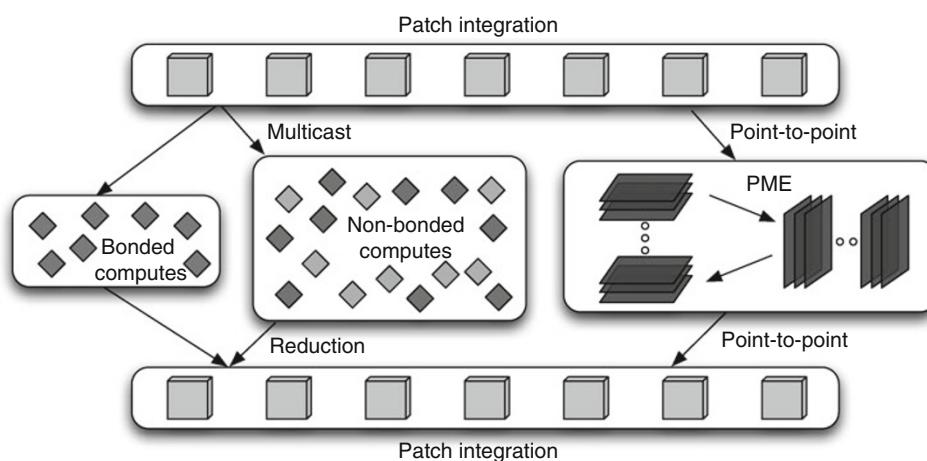
Parallelization of the algorithm mentioned above has been traditionally done in various ways (a good survey can be seen in Plimpton et al. [5]). Some of them are outlined here:

- Atom Decomposition: All atoms in the simulation box are distributed among the processors and each processor is responsible for the force calculations of the atoms it holds.
- Force Decomposition: The force matrix is distributed across the processors for parallelization.
- Spatial Decomposition: The three-dimensional simulation box is spatially divided among the processors and each processor is responsible for the atoms within that area.

Atom and force decomposition have high communication to computation ratios and spatial decomposition has problems of limited parallelism and load imbalance. For NAMD, a hybrid between spatial and force decomposition was developed, which combines the advantages of both [6]. Similar methods have been used in MD packages such as Blue Matter and Desmond.

The simulation box consisting of a spatial distribution of atoms is divided into smaller cells called “patches.” The dimensions of each patch are roughly equal to  $r_c + \text{margin}$  where  $r_c$  is the cutoff distance and  $\text{margin}$  is a small constant. This ensures that atoms within a cutoff radius of each other stay within neighboring patches over a few time steps. Patches are assigned to some processors (typically the number of patches is much smaller than the number of processors). Force calculations for each pair of patches is assigned to a different object called a “compute.” The number of computes in a simulation is much larger and they are load balanced across all the processors. Computes responsible for nonbonded force calculations within the cutoff radius account for most of the computation time in NAMD. Computes can be assigned to any processor irrespective of where the associated patches are placed.

**Figure 1** shows the communication and computation involved in a time step. At the beginning of a time step, patches send their atoms to the respective computes (nonbonded, bonded, and PME). The computes do the force calculations and send the forces back to the patches. The patches calculate velocities and new



NAMD (NA noscale Molecular Dynamics). **Fig. 1** Flow diagram showing the computation of forces in NAMD every step

positions for their particles and then this process is repeated. The load balancing framework in Charm++ is used to adaptively balance work units or computes across processors. The runtime instruments the load of each compute for a few time iterations and then uses this information to move work units around. The first load balancing phase migrates many computes based on a greedy strategy but the subsequent phases do refinement-based balancing. On machines such as IBM Blue Gene/L and Blue Gene/P, the load balancer also uses topology information to optimize communication and minimize contention.

## Performance

NAMD was designed more than 10 years ago but it has withstood the changes in technology because of its parallel-from-scratch, migratable-objects-based design. The strategy of hybrid decomposition coupled with adaptive overlap of communication and computation, supported by dynamic load balancing has helped NAMD scale to very large machines. NAMD can be run on most parallel machines due to the portability of the CHARM++ runtime. It can be run on any number of processors and has no powers-of-two restrictions

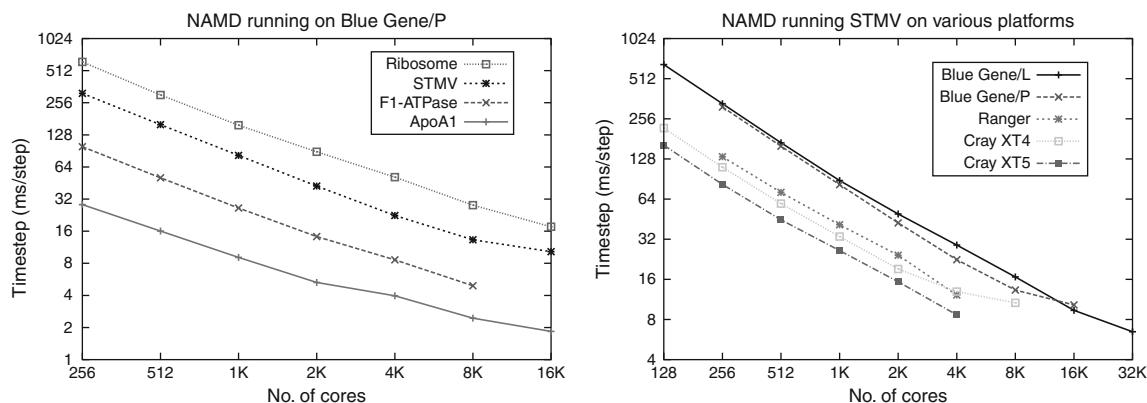
on the number. Another important feature of NAMD is that it performs well for a wide range of molecular system sizes.

**Table 1** shows the four molecular systems for which benchmarking results are being presented. Important configuration parameters such as the cutoff distance used and duration of each time step (in femtoseconds) are listed. These systems represent a wide range of sizes from 92 thousand atoms to 2.8 million atoms. The left plot in [Fig. 2](#) shows the performance of NAMD on the Blue Gene/P machine at Argonne National Laboratory (ANL). Four different lines show the strong scaling performance of NAMD for the chosen molecular systems from 256 to 16,384 cores. For ApoA1, one can simulate 47 ns per day when running on 16,384 cores of Blue Gene/P. Simulation of ribosome, a 2.8 million atom system, has an efficiency of 55% at 16,384 cores compared to the performance at 256 cores.

NAMD is used by biophysicists and chemists at most national laboratories for biomolecular research. **Table 2** presents architectural specifications of some of the machines NAMD has been run on and demonstrated to have good performance. Of these, the Cray XT5 system (Jaguar) at ORNL is currently the fastest

**NAMD (NA noscale Molecular Dynamics).** **Table 1** Molecular systems used for benchmarking NAMD

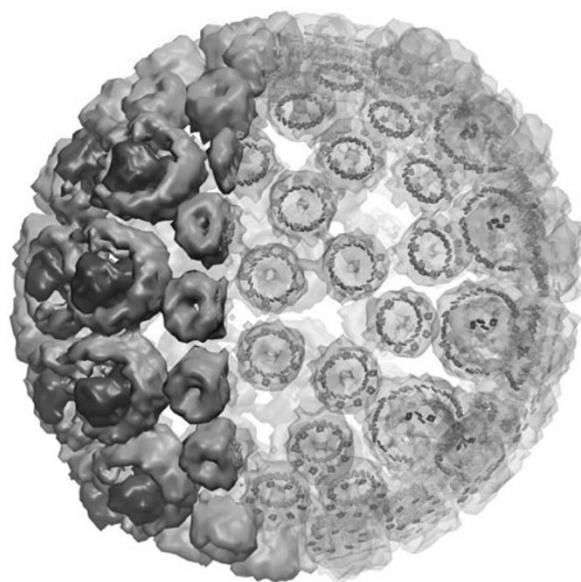
| Molecular system | Atom count | Cutoff (Å) | Simulation cell (Å <sup>3</sup> ) | Time step (fs) |
|------------------|------------|------------|-----------------------------------|----------------|
| ApoA1            | 92,224     | 12         | 108.86 × 108.86 × 77.76           | 1              |
| F1-ATPase        | 327,506    | 12         | 178.30 × 131.54 × 132.36          | 1              |
| STMV             | 1,066,628  | 12         | 216.83 × 216.83 × 216.83          | 1              |
| Ribosome         | 2,820,530  | 12         | 264.02 × 332.36 × 309.04          | 1              |



**NAMD (NA noscale Molecular Dynamics).** **Fig. 2** Performance (strong scaling) of NAMD

**NAMD (NAnoscale Molecular Dynamics). Table 2** Specifications of the parallel systems used for the runs

| System name | Location | No. of nodes | Cores per node | CPU type | Clock speed | Memory per node | Type of network |
|-------------|----------|--------------|----------------|----------|-------------|-----------------|-----------------|
| Blue Gene/L | IBM      | 20,480       | 2              | PPC440   | 700 MHz     | 512 MB          | 3D Torus        |
| Blue Gene/P | ANL      | 40,960       | 4              | PPC450   | 850 MHz     | 2 GB            | 3D Torus        |
| Cray XT4    | ORNL     | 7,832        | 4              | Opteron  | 2.1 GHz     | 8 GB            | SeaStar 2       |
| Cray XT5    | ORNL     | 18,688       | 12             | Opteron  | 2.6 GHz     | 16 GB           | SeaStar 2+      |
| Ranger      | TACC     | 3,936        | 16             | Xeon     | 2.3 GHz     | 32 GB           | Infiniband      |



**NAMD (NAnoscale Molecular Dynamics). Fig. 3** Structural model of a photosynthetic chromatophore vesicle, a biological light-harvesting machine (shown are the constituent proteins: 101 LH2 complexes surrounding 18 dimers of LH1-reaction center complexes). The system contains approximately 4,000 bacteriochlorophylls. Proteins are shown as semitransparent on the right-hand side to display the bacteriochlorophyll network. Image generated with VMD by Melih Sener based on work by himself, Jen Hsin, Chris Harrison, and Klaus Schulten

supercomputer in the world (as per the November 2009 Top500 list). NAMD has a small memory footprint and hence it can simulate a range of molecular systems on machines with limited memory (such as the IBM Blue Gene machines). However, scientists are now planning to simulate 25–100 million atom systems using NAMD. This will be discussed further in the Future Directions section.

Figure 2 (right plot) shows the benchmarking results for NAMD on five supercomputers at four different sites. They are representative of IBM machines, Cray machines, and Infiniband clusters. The molecular system being simulated is the one million atom STMV (Satellite Tobacco Mosaic Virus). NAMD clearly scales well on all these machines. Using 4,096 cores of the Cray XT5 machines, one can simulate around 10 ns per day for the STMV system.

## Conclusion

Molecular dynamics simulations are hard to parallelize because of the relatively small size of molecular systems and because one needs to run for millions of time steps to simulate a small period in the life of a biomolecule. NAMD has been used as an MD package for many years now and has withstood the changes in technology. Its fundamentally parallel design based on migratable CHARMM++ objects, supported by dynamic load balancing, has made it possible to scale NAMD to the machines of the petascale era. NAMD is continually being used to simulate larger and larger molecular systems, which pose new challenges for parallelization on large supercomputers. Recent modifications to NAMD such as parallel input/output and hierarchical load balancing have made this possible.

## Future Directions

Supercomputers are becoming larger each year with very little increase in the computational power of each individual processing core. Biomolecules are of fixed size, which places NAMD's required performance characteristics in the “strong-scaling” category. This constrains future research for this style of molecular dynamics to either the study of systems sufficiently

large to perform well on future machines, maintaining consistent performance improvements for ever-decreasing computation granularity, or integration with other methods in multi-physics models. Efforts are being put in all of those approaches.

NSF's Blue Waters project, a sustained Petaflop/s machine due to enter production in 2011, includes an MD benchmark acceptance test of 100 million atom system of DOPS, DOPC, and curvature-inducing protein BAR domains in water solvent. Support for running this system, already two orders of magnitude larger than is typical, has been added to NAMD and performance tuning for this system on the Blue Waters architecture is ongoing. Systems of similar size, such as the photosynthetic chromatophore shown in Fig. 3, will leverage these technologies to meet cutting edge science goals and permit the study of large structures for at least tens of nanoseconds of simulation time.

The NAMD and CHARMM++ development efforts continue to improve strong scaling performance through a variety of overhead reduction schemes and to adapt the software to make use of the network and computation resources idiosyncratic to new supercomputer architectures.

## Related Entries

- [Anton, a Special-Purpose Molecular Simulation Machine](#)
- [Charm++](#)
- [N-Body Computational Methods](#)

## Bibliographic Notes and Further Reading

A good survey of parallelization techniques for MD programs can be found in Plimpton et al. [5]. Snir discusses the communication requirements of force and spatial decomposition and proposes a hybrid algorithm similar to that of NAMD [7]. Other MD packages similar to NAMD are CHARMM, AMBER, GROMACS, Blue Matter, and Desmond.

The paper by Kale et al. [3] was awarded the Gordon Bell award at Supercomputing 2002. Detailed performance benchmarking of NAMD and recent algorithmic changes to enable scaling to large machines can be found in [1] and [8].

## Bibliography

1. Bhatele A, Kumar S, Mei C, Phillips JC, Zheng G, Kalé LV (2008) Overcoming scaling challenges in biomolecular simulations across multiple platforms. In: Proceedings of IEEE international parallel and distributed processing symposium, Miami, 2008
2. Phillips JC, Braun R, Wang W, Gumbart J, Tajkhorshid E, Villa E, Chipot C, Skeel RD, Kalé LV, Schulten K (2005) Scalable molecular dynamics with NAMD. *J Comput Chem* 26(16):1781–1802
3. Phillips JC, Zheng G, Kumar S, Kale LV (2002) NAMD: biomolecular simulation on thousands of processors. In: Proceedings of the 2002 ACM/IEEE conference on supercomputing, pp 1–18, Baltimore, 2002
4. Kalé LV, Skeel R, Bhandarkar M, Brunner R, Gursoy A, Krawetz N, Phillips JC, Shinozaki A, Varadarajan K, Schulten K (1999) NAMD2: greater scalability for parallel molecular dynamics. *J Comput Phys* 151:283–312
5. Plimpton SJ, Hendrickson BA (1996) A new parallel method for molecular-dynamics simulation of macromolecular systems. *J Comp Chem* 17:326–337
6. Kalé LV, Skeel R, Bhandarkar M, Brunner R, Gursoy A, Krawetz N, Phillips JC, Shinozaki A, Varadarajan K, Schulten K (1998) NAMD2: greater scalability for parallel molecular dynamics. *J Comput Phys* 151:283–312, 1999
7. Snir M (2004) A note on n-body computations with cutoffs. *Theory Comput Syst* 37:295–318
8. Bhatelè A, Kalé LV, Kumar S (2009) Dynamic topology aware load balancing algorithms for molecular dynamics applications. In: 23rd ACM international conference on supercomputing, Yorktown Heights, 2009

## NAnoscale Molecular Dynamics (NAMD)

- [NAMD \(NAnoscale Molecular Dynamics\)](#)

## NAS Parallel Benchmarks

DAVID H. BAILEY

Lawrence Berkeley National Laboratory, Berkeley, CA, USA

## Acronyms

NAS, NPB

## Definition

The NAS Parallel Benchmarks (NPB) are a suite of parallel computer performance benchmarks. They were

originally developed at the NASA Ames Research Center in 1991 to assess high-end parallel supercomputers [2]. Although they are no longer used as widely as they once were for comparing high-end system performance, they continue to be studied and analyzed a great deal in the high-performance computing community. The acronym “NAS” originally stood for the Numerical Aeronautical Simulation Program at NASA Ames. The name of this organization was subsequently changed to the Numerical Aerospace Simulation Program, and more recently to the NASA Advanced Supercomputing Center, although the acronym remains “NAS.” The developers of the original NPB suite were David H. Bailey, Eric Barszcz, John Barton, David Browning, Russell Carter, LeoDagum, Rod Fatoohi, Samuel Fineberg, Paul Frederickson, Thomas Lasinski, Rob Schreiber, Horst Simon, V. Venkatakrishnan, and Sisira Weeratunga.

## Discussion

The original NAS Parallel Benchmarks consisted of eight individual benchmark problems, each of which focused on some aspect of scientific computing. The principal focus was in computational aerophysics, although most of these benchmarks have much broader relevance, since in a much larger sense they are typical of many real-world scientific computing applications.

The NPB suite grew out of the need for a more rational procedure to select new supercomputers for acquisition by NASA. The emergence of commercially available highly parallel computer systems in the late 1980s offered an attractive alternative to parallel vector supercomputers that had been the mainstay of high-end scientific computing. However, the introduction of highly parallel systems was accompanied by a regrettable level of hype, not only on the part of the commercial vendors but even, in some cases, by scientists using the systems. As a result, it was difficult to discern whether the new systems offered any fundamental performance advantage over vector supercomputers, and, if so, which of the parallel offerings would be most useful in real-world scientific computation.

In part to draw attention to some of the performance reporting abuses prevalent at the time, the present author wrote a humorous essay “Twelve Ways to Fool the Masses,” which described in a light-hearted way a number of the questionable ways in which both

vendor marketing people and scientists were inflating and distorting their performance results [1]. All of this underscored the need for an objective and scientifically defensible measure to compare performance on these systems.

At the time (1991), the only widely available high-end benchmark was the scalable Linpack benchmark, which while it was useful and remains useful to this day, was not considered typical of most applications on NASA’s supercomputers or at most other sites. One possible solution was to employ an actual large-scale application code, such as one of those being used by scientists using supercomputers at the NAS center. However, due in part to the very large programming and tuning requirements, these were considered too unwieldy to be used to compare a broad range of emerging parallel systems. Compounding these challenges at this time was the lack of a generally accepted parallel programming model – note this was several years before the advent of the Message Passing Interface (MPI) and OpenMP models that are in common usage today.

For these reasons, the NAS Benchmarks were initially designed not as a set of computer codes, but instead were specified as “paper and pencil” benchmarks defined in a technical document [3]. The idea was to specify a set of problems only algorithmically, but in sufficient detail that the document could be used for a full-fledged implementation complying with the requirements. Even the input data or a scheme to generate it was specified in the document. Some “reference implementations” were provided, but these were intended only as aids for a serious implementation, not to be the benchmarks themselves. The original rules accompanying the benchmarks required that the benchmarks be implemented in some extension of Fortran or C (see details below), but otherwise implementers were free to utilize language constructs that give the best performance possible on the particular system being studied. The choice of data structures, processor allocation, and memory usage was (and are) generally left open to the discretion of the implementer.

The eight problems consist of five “kernels” and three “simulated computational fluid dynamics (CFD) applications.” The five kernels are relatively compact problems, each of which emphasizes a particular type of numerical computation. Compared with the simulated

CFD applications, they can be implemented fairly readily and provide insight as to the general levels of performance that can be expected on these specific types of numerical computations.

The three simulated CFD applications, on the other hand, usually require more effort to implement, but they are more indicative of the types of actual data movement and computation required in state-of-the-art CFD application codes, and in many other three-dimensional physical simulations as well. For example, in an isolated kernel, a certain data structure may be very efficient on a certain system, and yet this data structure would be inappropriate if incorporated into a larger application. By comparison, the simulated CFD applications require data structures and implementation techniques in three physical dimensions, and thus are more typical of real scientific applications.

## Benchmark Rules

Even though the benchmarks are specified in a technical document, and implementers are generally free to code them in any reasonable manner, certain rules were specified for these implementations. The intent here was to limit implementations to what would be regarded as “reasonable” code similar to that used in real scientific applications. In particular, the following rules were presented, and, with a couple of minor changes, are still in effect today (these rules are the current version):

- All floating point operations must be performed using 64-bit floating point arithmetic (at least).
- All benchmarks must be coded in either Fortran (Fortran-77 or Fortran-90), C, or Java, with certain approved extensions. Java was added in a recent version of the NPB.
- One of the three languages must be selected for the entire implementation – mixed code is not allowed.
- Any language extension or library routine that is employed in any of the benchmarks must be supported by the system vendor and available to all users.
- Subprograms and library routines not written in Fortran, C, or Java (such as assembly-coded routines) may only perform certain basic functions [a complete list is provided in the full NPB specification].

- All rules apply equally to subroutine calls, language extensions, and compiler directives (i.e., special comments).

## The Original Eight Benchmarks

A brief (and necessarily incomplete) description of the eight problems is given here. For full details, see the full NPB specification document [3].

*EP.* As the acronym suggests, this is an “embarrassingly parallel” kernel – in contrast to others in the list, it requires virtually no interprocessor communication, only coordination of pseudorandom number generation at the beginning and collection of results at the end. There is some challenge in computing required intrinsic functions (which, according to specified rules, must be done using vendor-supplied library functions) at a rapid rate.

The problem is to generate pairs of Gaussian random deviates and tabulate the number of pairs in successive square annuli. The Gaussian deviates are to be calculated by the following well-known scheme. First, generate a sequence of  $n$  pairs of uniform  $(0,1)$  pseudorandom deviates  $(x_j, y_j)$  (using a specific linear congruential scheme specified in the benchmark document). Then, for each  $j$ , check whether  $t_j = x_j^2 + y_j^2 \leq 1$ . If so, set  $X_k = x_j\sqrt{(-2\log t_j)/t_j}$  and  $Y_k = y_j\sqrt{(-2\log t_j)/t_j}$ , where the index  $k$  is incremented with every successful test; if  $t_j > 1$ , then reject the pair  $(x_j, y_j)$ . In this way, the resulting pairs  $(X_k, Y_k)$  are random deviates with a Gaussian distribution. The sums  $S_1 = \sum_k X_k$  and  $S_2 = \sum_k Y_k$  are each accumulated, as are the counts of the number of hits in successive square annuli.

The verification test for this problem requires that the sums  $S_1$  and  $S_2$  each agree with reference values to within a specified tolerance, and also that the ten counts of deviates in square annuli exactly agree with reference values.

*MG.* This is a simplified multigrid calculation. This requires highly structured long-distance communication and tests both short- and long-distance data communication.

The problem definition is as follows. Set  $v = 0$  except at the twenty specified points, where  $v = \pm 1$ . Commence an iterative solution with  $u = 0$ . Each of the four

iterations consists of the following two steps, in which  $k = 8$ :

$$\begin{aligned} r &:= vAu \quad (\text{Evaluate residual}) \\ u &:= u + M^k r. \quad (\text{Apply correction}) \end{aligned}$$

Here  $M^k$  denotes the following V-cycle multigrid operator:  $z_k := M^k r_k$ , where if  $k > 1$  then

$$\begin{aligned} r_{k-1} &:= Pr_k \quad (\text{Restrict residual}) \\ z_{k-1} &:= M^{k-1} r_{k-1} \quad (\text{Recursive solve}) \\ z_k &:= Qz_{k-1} \quad (\text{Prolongate}) \\ r_k &:= r_k Az_k \quad (\text{Evaluate residual}) \\ z_k &:= z_k + Sr_k, \quad (\text{Apply smoother}) \end{aligned}$$

else

$$z_1 := Sr_1. \quad (\text{Apply smoother})$$

The coefficients of the  $P$ ,  $M$ ,  $Q$ , and  $S$  arrays, together with other details, are given in the benchmark document.

The benchmark problem definition requires that a specified number of iterations of the above V-cycle be performed, after which the  $L_2$  norm of the  $r$  array must agree with a reference value to within a specified tolerance.

**CG.** In this problem, a conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel is typical of unstructured grid computations in that it tests irregular long-distance communication, employing unstructured matrix vector multiplication.

The problem statement in this case is fairly straightforward: Perform a specified number of conjugate gradient iterations in approximating the solution  $z$  to a certain specified large sparse  $n \times n$  linear system of equations  $Az = x$ . In this problem, the matrix  $A$  must be used explicitly. This is because after the original NPB suite was published, it was noted that by saving the random sparse vectors  $x$  used in the specified construction of the problem, it was possible to reformulate the sparse matrix-vector multiply operation in such a way that communication is substantially reduced. Therefore this scheme is specifically disallowed.

The verification test is that the value  $\gamma = \lambda + 1/(x^T z)$  (where  $\lambda$  is a parameter that depends on problem size) must agree with a reference value to a specified tolerance.

**FT.** Here, a 3-D partial differential equation is solved using FFTs. This performs the essence of many “spectral” codes. It is a rigorous test of heavy long-distance communication performance.

The problem is to numerically solve the Poisson partial differential equation (PDE)

$$\frac{\partial u(x,t)}{\partial t} = \alpha \nabla^2 u(x,t),$$

where  $x$  is a position in three-dimensional space. When a Fourier transform is applied to each side, this equation becomes

$$\frac{\partial v(z,t)}{\partial t} = -4\alpha\pi^2|z|^2v(z,t),$$

where  $v(z,t)$  is the Fourier transform of  $u(x,t)$ . This has the solution

$$v(z,t) = e^{-4\alpha\pi^2|z|^2t}v(z,0).$$

The benchmark problem is to solve a discrete version of the original PDE by computing the forward 3-D discrete Fourier transform (DFT) of the original state array  $u(x,0)$ , multiplying the results by certain exponentials, and then performing an inverse 3-D DFT. Of course, the DFTs can be rapidly evaluated by using a 3-D fast Fourier transform (FFT) algorithm.

The verification test for this problem is to match the checksum of a certain subset of the final array with reference values.

**IS.** This kernel performs a large integer sort operation that is important in certain “particle method” codes. It tests both integer computation speed and communication performance.

The specific problem is to generate a large array by a certain scheme and then to sort it. Any efficient parallel sort scheme may be used. The verification test is to certify that the array is in sorted order (full details are given in the benchmark document).

**LU.** This performs a synthetic computational fluid dynamics (CFD) calculation by solving regular-sparse, block ( $5 \times 5$ ) lower and upper triangular systems.

*SP.* This performs a synthetic CFD problem by solving multiple, independent systems of nondiagonally dominant, scalar, pentadiagonal equations.

*BT.* This performs a synthetic CFD problem by solving multiple, independent systems of nondiagonally dominant, block tridiagonal equations with a  $(5 \times 5)$  block size.

These last three “simulated CFD benchmarks” together represent the heart of the computationally intensive building blocks of CFD programs in most common use today for the numerical solution of three-dimensional Euler/Navier-Stokes equations using finite-volume, finite-difference discretization on structured grids. LU and SP involve global data dependencies. Although the three benchmarks are similar in many respects, there is a fundamental difference with regard to the communication-to-computation ratio. BT represents the computations associated with the implicit operator of a newer class of implicit CFD algorithms. This kernel exhibits somewhat more limited parallelism compared with the other two.

In each of these three benchmarks, the same high-level synthetic problem is solved. This synthetic problem differs from a real CFD problem in the following important aspects:

1. Absence of realistic boundary algorithms
2. Higher than normal dissipative effects
3. Lack of upwind differencing effects
4. Absence of geometric stiffness introduced through boundary conforming coordinate transformations and highly stretched meshes
5. Lack of evolution of weak solutions found in real CFD applications during the iterative process
6. Absence of turbulence modeling effects

Full details of these three benchmark problems are given in the benchmark document, as are the verification tests.

## Evolution of the NAS Parallel Benchmarks

The original NPB suite was accompanied by a set of “reference” implementations, including implementations for a single-processor workstation, an Intel Paragon system (using Intel’s message passing library) and a CM-2 from Thinking Machines Corp. Also, the original release defined three problem sizes: Class W (for the sample workstation implementation), Class A

(for what was at the time a moderate-sized parallel computer), and Class B (for what was at the time a large parallel computer). The Class B problems were roughly four times larger than the Class A problems, both in total operation count and in aggregate memory requirement.

After the initial release of the NPB, the NASA team published several sets of performance results, for example, [4, 5], and in the next few years, numerous teams of scientists and computer vendors submitted results.

By 1996, several of the original NPB team had left NASA Ames Research Center, and, for a while, active support and collection of results lagged. Fortunately, some other NASA Ames scientists, notably William Saphir, Rob Van der Wngaart, Alex Woo, and Maurice Yarrow, stepped in to continue support and development. These researchers produced a reference implementation of the NPB with MPI constructs, which, together with a few minor changes, was designated NPB 2.0. Then a Class C problem size was defined as part of NPB 2.2.

Another enhancement was the addition of the “BT I/O” benchmark. In this benchmark, implementers were required to output some key arrays to an external file system as the BT program is running. Thus, by comparing the performance of the BT and BT I/O benchmarks, one could get some measure of I/O system performance. This change, together with the definition of Class D problem sizes, was released in 1997 and designated NPB 2.4.

In 1999, NASA Ames researchers Haoqiang Jin, Michael Frumkin, and Jerry Yan released NPB 3.0, which included an OpenMP reference implementation of the NPB.

In 2002, Rob van der Wngaart and Michael Frumkin released a grid version of the NPB. In the next year or two, several additional minor improvements and bug fixes were made to all reference implementations, and Class E problem sizes were defined. Reference implementations were released in Java 3.0 and High-Performance Fortran, and a “multi-zone” benchmark, reflective of a number of more modern CFD computations, was added. These changes were designated NPB 3.3, which is the latest version on the NASA Web site.

Full details of the current version of NPB, as well as the actual documents and reference implementations, are available at: <http://www.nas.nasa.gov/Resources/Software/npb.html>

## Related Entries

- [Benchmarks](#)
- [HPC Challenge Benchmark](#)
- [LINPACK Benchmark](#)

## Acknowledgment

Supported in part by the Director, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy, under contract number DE-AC02-05CH11231.

## Bibliography

1. Bailey DH (1991) Twelve ways to fool the masses when giving performance results on parallel computers. *Supercomput Rev* 54–55 (Also published in *Supercomputer*, Sep. 1991, 4–7. Available at <http://crd.lbl.gov/~dhbailey/dhbpapers/twelve-ways.pdf>.)
2. Bailey DH, Barszcz E, Barton JT, Browning DS, Carter RL, Dagum L, Fatoohi RA, Frederickson PO, Lasinski TA, Schreiber RS, Simon HD, Venkatakrishnan V, Weeratunga SK (1991) The NAS parallel benchmarks. *Int J Supercomput Appl* 5(3):63–73
3. Bailey DH, Barszcz E, Barton JT, Browning DS, Carter RL, Dagum L, Fatoohi RA, Frederickson PO, Lasinski TA, Schreiber RS, Simon HD, Venkatakrishnan V, Weeratunga SK (1991) The NAS parallel benchmarks. Technical report RNR-94-007, NASA Ames Research Center, Moffett Field, CA 94035, Jan. 1991, revised 1994 (Available at <http://www.nas.nasa.gov/News/Techreports/1994/PDF/RNR-94-007.pdf>)
4. Bailey DH, Barszcz E, Dagum L, Simon HD (1992) NAS parallel benchmark results. In: *Proceedings of Supercomputing 1992*, Minneapolis, MN, pp 386–393
5. Bailey DH, Barszcz E, Dagum L, Simon HD (1993) NAS parallel benchmark results. *IEEE Parallel and Distr Tech* 43–51

## N-Body Computational Methods

JOSEPH FOGARTY<sup>2</sup>, HASAN AKTULGA<sup>1</sup>, SAGAR PANDIT<sup>2</sup>, ANANTH Y. GRAMA<sup>1</sup>

<sup>1</sup>Purdue University, West Lafayette, IN, USA

<sup>2</sup>University of South Florida, Tampa, FL, USA

## Synonyms

[Particle dynamics](#); [Particle methods](#)

## Definition

N-body computation refers to a class of simulation methods that model the behavior of a physical system using a set of discrete entities (e.g., atoms, astrophysical

bodies, etc.) and a set of interactions among them (coupling potentials). These simulations are typically time dependent. In each timestep, attributes of the discrete entities are updated (typically force, acceleration, velocity, and position), and the process is repeated, to study spatiotemporal evolution of the system.

## Discussion

### Introduction

Many interesting physical problems can be modeled as the time-evolution of a set of interacting, classical objects (assumed to be point masses). The behavior of these particles is governed by Newton's second law. A series of seminal efforts by, among other notables, Newton, Euler, Lagrange, Hamilton, Delaunay, and Sundman demonstrated the difficulty of analytically solving the problem for systems comprised of more than two bodies. Simulation techniques solve the equation for many bodies by discretizing time and integrating particle behavior over discrete timesteps. These particles are treated as point masses, interacting through specified potentials. Systems of interest, though small in terms of macroscopic or astronomical scales, normally comprise of a large number of particles (up to  $10^9$  or more). The tools of statistical mechanics have been developed to analyze the macroscopic (average) properties and behavior of such large systems. While powerful in the context of many problems, statistical mechanics is ill-suited to the study of smaller structures, such as nanoparticles and interactions of individual molecules. N-body simulations are able to overcome these limitations, providing powerful simulation methodologies.

N

### Computational Techniques and Algorithmic Considerations

Astrophysical simulations are classical applications of N-body methods. In these simulations, interacting particles are large-scale stellar structures (stars, stellar clusters, galaxies, galaxy clusters, dark matter, etc. [33, 34]) interacting through long-range gravitational forces. Range here refers to the distance dependence of the potential function. At the other end of the spatiotemporal spectrum, one may model atomic-scale systems to understand physical properties of materials or behavior of molecular systems. Individual particles in such

systems correspond to atoms, and coupling potentials span long-range electrostatics and short-range Lennard-Jones potentials. In molecular scale systems, properties are modeled using customized coupling potentials. While the overall structure of computation is similar for these diverse  $N$ -body simulations, there are significant differences in the nature and computational cost of interactions and required time resolution. The computational challenges in astrophysical simulations stem largely from the long-range nature of the gravitational force, whereas atomistic simulations involve a variety of complex interaction potentials. This article discusses  $N$ -body methods primarily in the context of atomistic simulations. Where relevant, it highlights differences with respect to astrophysical simulations and the implications thereof.

In molecular dynamics (MD), the system is modeled as a set of point charges (particles) whose evolution in time is governed by classical mechanics. Potentials are used to simulate electrostatics, harmonic bonds, hard sphere repulsion, and other types of interactions. Simulations may also be performed using a Monte Carlo (MC) approach. As its name suggests, the MC technique relies on a randomized process, appropriately sampling an ensemble to infer statistical quantities.

The physical environment modeled in a simulation further constrains the computational approach. For example, simulations may be performed under different collections of microstates (ensembles). Under the microcanonical ensemble (NVE) particle number ( $N$ ), volume ( $V$ ) and total energy ( $E$ ) are held constant. The isochoric-isothermal canonical ensemble (NVT) maintains constant temperature ( $T$ ) by placing the system in contact with a thermal reservoir. This requires the implementation of a thermostat such as Anderson [36], Nose-Hoover [35], or Berendsen [37]. The isobaric-isothermal canonical ensemble (NPT) adds a piston to the canonical ensemble. The volume of the system is allowed to vary in order to maintain a constant pressure ( $P$ ). Different barostats such as Berendsen [37] and Parrinello-Rahman [38] may be used for this purpose.

In a general  $N$ -body problem, forces and potentials governing the motion of particles can be classified as short- or long-range. Short-range potentials decay faster than the growth of the volume element

in three dimensions (i.e., they approach zero faster than  $1/r^3$ ) [31]. Consequently, they can be appropriately truncated for a desired level of accuracy. Long-range forces, on the other hand, do not decay as rapidly – requiring either expensive computation or algorithmic techniques for reducing the complexity. Note that a simple all-to-all computation of long-range forces would result in an  $O(N^2)$  complexity per timestep. Algorithmic techniques typically rely on hierarchical decompositions to aggregate/approximate the impact of distant particle clusters, thereby reducing overall complexity. Methods such as Barnes–Hut (BH) [3] and Fast Multipole Methods (FMM) [10] rely on these principles to reduce the  $O(N^2)$  complexity to  $O(N \log N)$  and  $O(N)$ , respectively.

## Parallelization Challenges

In parallelizing  $N$ -body simulation, two primary considerations motivate the choice of how the computation should be decomposed into different processes. First the load on each processor should be approximately equal. Second, the system should be decomposed in order to minimize interprocess communication, which is a primary consideration for computational efficiency. A number of decomposition techniques satisfy these constraints to varying degrees.

*Domain Decomposition:* In domain decomposition, the physical system is divided into regions of space. A constituent particle falling within a specified volume is assigned to the corresponding processor. Inter-processor communication is needed for computing forces on boundary particles, as well as to “push” particles at the end of the timestep. Since timesteps are small, the system does not change drastically from one timestep to the next. For this reason, the communication associated with pushing particles at the end of a timestep is typically low. Great care must be exercised to reduce the communication overhead. Load balancing an  $N$ -body simulation with domain decomposition is most easily achieved when the system consists of identical particles and uniform density. In this case, an equipartitioning of the space into processor subdomains ensures load balance. Since systems of interest are rarely so

simple, more sophisticated domain decomposition techniques are required.

One such technique is oct-tree decomposition. In this method, the simulation box is recursively divided until the desired number of particles are contained within each subdomain. The oct-tree is rebuilt when necessary to achieve desired load balance. Implementation and maintenance of the oct-tree decomposition method is much more complicated than equipartitioning of the space.

Another domain decomposition technique that ensures dynamic load balance is the staggered decomposition method used by GROMACS [14]. Here the processor subdomains are aligned in one dimension, but staggered in the other two dimensions. Based on the cpu time taken by each processor at any timestep, processor boundaries slide to obtain an even work-load distribution. Due to sliding boundaries and staggered arrangement of processors, interprocessor communication is complicated.

**Particle Decomposition:** In a particle decomposition, particles are assigned to processors for the duration of the simulation. This method yields good load balancing despite nonuniform density and is best suited to systems with minimal translational motion, for example, simulations of solids. Communication overhead, however, may be high when the neighborhood of atoms is rapidly changing.

**Interaction Decomposition:** Interaction decomposition assigns specific interactions to each processor, resulting in load balance. Communication is minimized since any interacting atoms are handled by the same processor. Despite its desirable characteristics, the implementation of an interaction-based decomposition is often difficult, since it requires prior knowledge of the number and nature of interactions.

**Zonal Decomposition:** A recently developed algorithm [15] greatly reduces communication costs. In this method, spatial decomposition is the primary approach. In addition to local atoms, buffer atoms are communicated so that all interactions are computed locally. Inclusion of buffer atoms is based on interactions rather than spatial location. Note that at present time zonal methods are limited to pair-wise interactions.

## Boundary Conditions

In typical large-scale simulations, one does not simulate the entire domain, but rather a small “cutout” of the domain. A small subdomain assumed to be surrounded by infinitely replicated domains in each direction is simulated. This yields the boundary conditions for the system. The above-mentioned scenario (infinite replicated domains) is referred to as a periodic boundary condition. Periodic boundary conditions lead to problems with long-range potentials, as all (infinitely many) of the periodic images interact with the system. Specifically, electrostatic interactions which decay as  $1/r$ , need to be handled with a method that accounts for infinitely many periodic images. Most commonly, the Ewald method [32] is used for this purpose. Implementation of this method requires the use of a reciprocal lattice. The complexity of this computation stems from underlying Fourier transforms.

Periodic boundaries also lead to difficulty in defining and computing external pressure. For pair-additive potentials, the virial expansion can be used to calculate pressure despite the implementation of periodic boundary conditions. Polarizable force fields and ab initio molecular dynamics must use a more computationally expensive approach [39].

N

## Parallel Implementation of N-body Methods

A number of important domain considerations influence the choice of algorithms, resulting computational complexity, structure, and parallelization strategy of  $N$ -body methods. The main factors that influence parallelization include (i) handling of long-range potentials, (ii) computation of short-range potentials, (iii) implementation of suitable boundary conditions, (iv) time-integration of the interaction, and (v) additional processing associated with particles (analysis routines, checkpointing trajectories, etc.). This section describes in more detail the nature of these computations and their parallelization.

## Interactions in N-Body Simulations

A molecular dynamics simulation begins with an initial geometry for the system. Normally, assuming equipartition of energy, each atom is assigned a random velocity based on the Maxwell–Boltzmann distribution.

The interaction potentials between atoms are then calculated. Forces are computed from the potentials using the equation

$$\vec{F} = -\nabla \phi,$$

where  $\vec{F}$  is the force and  $\phi$  the interaction potential. The resulting coupled differential equations are then integrated in order to determine the velocities and positions of each atom in the next timestep. Nearly 70–90% of processing time in an  $N$ -body simulation is spent in the computation of interaction potentials and forces. Parallelization of these computations is the major challenge in large-scale  $N$ -body simulations.

This subsection will focus on the methods for parallelizing the computation of interatomic interactions. These interactions are broadly classified in two types, viz bonded and nonbonded. Standard classical MD approaches treat bonds as invariants. Hence the parallelization of the computation of bonded terms is relatively straightforward, depending on the decomposition method. Nonbonded interactions, on the other hand, require more careful treatment. These terms are further classified as short range or long range. By their very nature, long-range interactions are more difficult to compute.

### Computation of Bonded Terms

Bonded interactions in molecular dynamics are typically modeled using a harmonic potential. Much as the compression or elongation of a spring changes potential energy, bond stretching and shrinking contributes to the overall energy of the system. These two-body terms, determined by the positions of bonded atoms and their types, must be computed at each timestep. In addition, higher order bonded terms such as three-body (angle) and four-body (torsion and dihedral) terms must be calculated. To compute bond and higher order terms, lists of all two-, three-, and four-body interactions are maintained. Using suitable indices, the position of each atom's bonded neighbors, and thus the corresponding forces can be computed. Since conventional molecular dynamics approaches do not model the process of bond-breaking or formation, these lists are static. For computing these terms, an interaction-based decomposition of the computation is preferred, since the interacting constituents are known and constant.

### Computation of Short-Range Terms

Typical coupling terms between bodies decay with distance. For example, electrostatic coupling potentials decay as  $1/r$ , gravitational interaction decays as  $1/r^2$ , Lennard-Jones potential decays as  $1/r^6$ , etc. Depending on the rate of decay and the required accuracy, it is possible to truncate these coupling terms. For example, Lennard-Jones potentials are often truncated at some multiple of  $\sigma$ , the distance at which the interparticle potential becomes zero. For typical molecular dynamics simulations, this is on the order of 10 Å.

Short-range interactions, due to their dynamic nature, typically require spatial decomposition to reduce computational cost. Computing short-range interactions for a particle requires complete knowledge of all particles within the neighborhood defined by the cutoff distance. If one assumes a constant (or bounded) particle density and a constant cutoff distance, it is easy to see that the number of short-range interactions is constant (or bounded) as well. The challenge is to identify all particles within the cutoff distance from a given particle.

Most interactions modeled are governed by central forces, depending only on the distance separating two bodies. Determining the distances between particles which is the most expensive part of a short-range computation, is achieved by use of neighbor lists. Since a brute force algorithm for neighbor list generation involves an  $O(N^2)$  computation, more efficient methods have been developed. For instance, a grid-based search is often utilized to reduce the complexity to  $O(N)$ . The granularity of the grid has a large effect on the net efficiency of the method. The grid method can further be improved by excluding a large number of computations within the grid based on the geometry. Careful consideration must be given to the design of neighbor lists in order to maximize cache performance. In molecular dynamics, particles typically do not have large displacements. For this reason, neighbor list generation can be delayed with the use of Verlet lists. Communication overhead for neighbor list generation can be further reduced by the use of a buffer zone, known as a skin, around processor boundaries.

While typical molecular dynamics simulations are consistent with a bounded particle density assumption, astrophysical  $N$ -body problems may not satisfy this

assumption. For such simulations, we must generalize the cell structure into an oct-tree decomposition of the domain. With suitably optimized oct-tree structures, we can realize  $O(N)$  complexity for short-range interactions for arbitrarily unstructured particle distributions as well. For a detailed discussion of these results, please see the work of Callahan and Kosaraju [4].

In some atomistic simulations such as ReaxFF [40], bond activity, and hence chemical reactivity, are explicitly modeled through the computation of a distance-dependent *bond order*. In effect, a bond order takes inter-atomic distances and computes the likelihood of bonds between pairs of atoms. This bond order is subsequently corrected to account for valency. In such cases, bond lists must be updated at each timestep and bonded interactions must be treated as short-range interactions.

### Computation of Long-Range Terms

Coupling potentials that are proportional to  $1/r$ , (e.g., electrostatic or gravitational potentials) introduce unacceptable errors when truncated. For this reason, these terms are handled separately. Naively computing long-range potentials on an all-to-all basis results in an  $O(n^2)$  complexity (since each of the  $n$  particles is impacted by every other particle). For large systems, this per-timestep complexity is not tractable. Consequently, a number of approximation techniques have been proposed to reduce complexity while incurring minimal error.

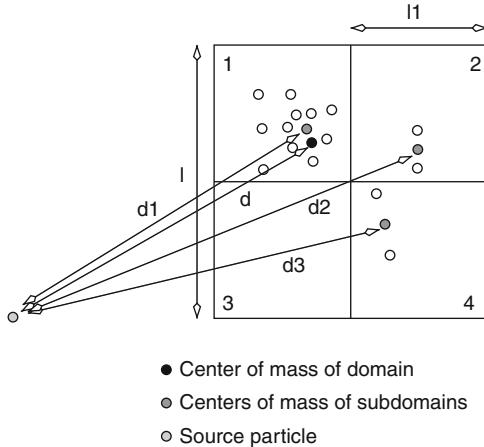
The basis for a family of approximation techniques is as follows: when a group of particles is sufficiently distant from an observation point, their impact can be approximated through a single interaction with the center of mass of the distant particle cluster. Indeed, when we compute the gravitational potential at a point on earth from the moon, we approximate the volume integral (over moons volume) by a point mass at the center of the moon. Since the radius of the moon is much less than its distance from earth, this approximation introduces controllable errors.

Many fast algorithms use this principle to reduce the computational complexity of  $N$ -body methods. These fast algorithms use a hierarchical representation of the domain using a spatial tree data structure. The leaf nodes consist of aggregates of particles. Each node in

the tree contains a series representation of the effect of the particles contained in the subtree rooted at the node. These representations are typically based on Taylor or Legendre polynomials. Interactions between nodes and particles are dictated by a multipole acceptance criteria (MAC). This criteria determines whether the error in a particular interaction is within acceptable bounds or not (can the moon really be treated as a point object with respect to the earth or not?). Different algorithms use different MAC. Selection of an appropriate MAC is critical to controlling the error in simulation. Methods in this class include those due to Appel [2, 5], Barnes and Hut [3], and Greengard and Rokhlin [8, 9, 10].

The Barnes–Hut method is one of the most popular methods due to its simplicity. It works in two phases: the-tree construction phase and the force-computation phase. In the tree-construction phase, a spatial tree representation of the domain is derived. At each step in this phase, if the domain contains more than one particle, it is recursively divided into eight equal parts. This process continues until each oct has a single element in it. The resulting tree is an unstructured oct-tree for arbitrary particle distributions. This tree is now traversed in post-order. Each internal node in the tree computes and stores an approximate representation of the particles contained in that tree. Once the tree has been computed, the force on each particle can be computed as follows: the multipole acceptance criteria is applied to the root of the tree to determine if an interaction can be computed; if not, the node is expanded and the process is repeated for each of the four (or eight) children. The multipole acceptance criteria for the Barnes–Hut method computes the ratio of the dimension of the box to the distance of the point from the center of mass of the box. If this ratio is less than some constant  $\alpha$  (a in Fig. 1), an interaction can be computed. The algorithm is illustrated in Fig. 1.

For a balanced tree, each of the  $n$  particles needs  $O(\log n)$  interactions. This results in a total computational complexity of  $O(n \log n)$ . However, the tree size can be made arbitrarily large by bringing a pair of particles closer. The corresponding tree needs a large number of boxes to resolve the pair into separate boxes. Due to this, the worst-case complexity of this technique is unbounded [1, 4]. However, using box-collapsing techniques (the box is first collapsed to the smallest box



**N-Body Computational Methods. Fig. 1** Illustration of the serial Barnes–Hut method

that contains all the particles in the subdomain), this complexity can be reduced to  $O(n \log n)$  [4]. In addition to box-collapsing, it has been shown that using binary decomposition of the domain (instead of quad or oct-tree decompositions) yields better aspect ratios and reduced operation-count for treecodes.

The fast multipole method (FMM) of Greengard and Rokhlin [10] is another hierarchical technique for computing  $n$ -body interactions. FMM computes the potential due to a cluster of particles at the center of well-separated clusters. This can then be disseminated to individual particle positions to determine required potentials. FMM therefore uses cluster-cluster interactions in addition to particle–cluster interactions. The computational complexity of FMM can be shown to be  $O(n)$  for well-structured particle distributions.

Several parallel formulations of hierarchical methods for long-range electrostatics have been proposed [6, 11, 12]. These methods generally rely on a spatially oriented domain decomposition for constructing hierarchical representations, as well as computing potential at each particle. Hierarchical representations of the domain are constructed in parallel as follows: each processor starts with an initial subdomain assigned (exclusively) to it. Processors construct a hierarchical representation (oct-tree) independently using only the particles assigned to it. In each processors’ oct-tree, it is possible to identify a minimal set of nodes such that all particles in the corresponding domains are exclusively assigned to a single processor. These nodes are referred

```

if ($l/d < a$)
 compute direct force interaction
 with the center of mass of domain,
else
 if ($l_1/d_1 < a$)
 compute direct force computation
 with center of mass of subdomain 1
 else
 expand subdomain 1 further.

```

Apply similar criteria to domains 2, 3, and 4.

to as branch nodes. It can be seen that hierarchical representations at and below the branch nodes are accurate. To construct a consistent image of the top part of the tree, branch nodes are broadcast to all processors. These nodes are inserted into each processor’s tree and the top part of the tree is recomputed by all the processors. This results in a data structure at each processor that is accurate at and above the branch nodes, as well as below local branch nodes.

Each processor can traverse this hierarchical representation to compute electrostatic potential at its assigned particles. The tree traversal phase can proceed in two different ways. As a tree is traversed, a processor may encounter nodes that are not present locally. In this case, the node may either be fetched from the remote processor (*data shipping*), or the particle itself may be shipped to the remote processor that holds the data (*function shipping*). The schemes of Singh et al. [11] and Warren and Salmon [12] are based on the data-shipping paradigm, whereas the scheme presented in [6] is based on a function-shipping paradigm. Function shipping simplifies the problem of addressing arbitrary nodes in a tree, since only branch nodes are remotely addressed. This is implemented using a hash table to access branch nodes. In data-shipping formulations, a processor may request an arbitrary node from a remote processor. Therefore, data-shipping formulations provide fast access to the entire data-set. This is typically accomplished by using the Morton key as a hash function.

In some applications particle distributions may be highly unstructured. In such cases, a naive domain partitioning potentially results in significant load imbalance or high communication overhead. To address this problem, space-filling curves such as the Peano–Hilbert ordering (costzones) and Morton ordering (also known as the Z-curve) are used. Particles are ordered in a list according to the space-filling curve and assigned to processors by partitioning the list so that each processor gets equal load. An estimate of the particles load can be derived by keeping track of the computation associated with the particle in the previous timestep. The motivation for using space-filling curves comes from the spatial proximity and the fact that both of these orderings number all particles in an oct before proceeding to subsequent octs. Furthermore, in data-shipping message passing formulations, space-filling curves allow easy location of arbitrary nodes in a tree without expensive pointer chasing.

The computation of long-range interactions is complicated by the application of periodic boundary conditions. Since these interactions cannot be easily truncated, naive computation of long-range interactions under periodic boundaries lead to an infinite number of terms. The Ewald sum is a method originally developed to calculate the electrostatic interaction in crystals. Systems with periodic boundary conditions mimic the infinite lattice of a crystal structure. Hence, the Ewald method is easily applied to molecular dynamics simulation. The Particle-Mesh Ewald summation decomposes the long-range potential into a short-range term that sums quickly in real space and a long-range term that sums quickly in the Fourier space. The latter term is computed using a Fast Fourier Transform, which contributes primarily to the computational cost of the method.

Parallel techniques for computing Fast Fourier transforms have been well studied in literature [7]. The scalability of these methods is critically impacted by the bisection bandwidth of the parallel platform on which they are implemented. For this reason, this part of the algorithm does not easily scale to large numbers of processors. One way of circumventing this scalability bottleneck is to execute the FFT separately on a subset of processors, while other processors execute other parts of the algorithm. Indeed, as we describe, a number of existing codes use this approach.

Since different interactions are ideally parallelized by different decomposition methods, the nature of interactions within an  $N$ -body computation determines the preferred method. Since spatial decomposition is often preferred in situations containing nonbonded interactions, bonded interactions must often be computed under non-ideal circumstances. The communication of neighboring atoms under spatial decomposition is further complicated by the existence of bonded interactions. Since bonded interactions require atoms several bonds away, these atoms must be communicated regardless of their distance from the processor boundary. Therefore, the generation of a list of skin atoms must also incorporate atoms in a 2-, 3-, and 4-body interaction across the subdomain, in addition to atoms within the cutoff distance.

## Some Major Public-Domain Efforts

### **GROMACS**

GROMACS is a freely available, open source molecular dynamics code in C distributed under the GPL. It was originally developed at the University of Groningen and is currently maintained by research groups in Europe. Parallelization of GROMACS has been significantly improved with the release of GROMACS 4 [14]. It relies on the eighth shell domain decomposition technique introduced by the D. E. Shaw group [15] to reduce communication bandwidth requirements. Processor domains are aligned along the  $x$  dimension, but staggered along  $y$  and  $z$  dimensions to facilitate dynamic load balancing.

N

### **NAMD**

NAMD [13] is a parallel molecular dynamics code designed primarily for the simulation of large biomolecular systems. It is written in C++, based on Charm++ parallel objects. NAMD works with popular biomolecular simulation force fields AMBER and CHARMM. The dynamic load-balancing procedure implemented in NAMD has a slow start, but performs well in the long run by scaling to a large number of processors. Recent benchmarks, however, show that NAMD scales worse than Desmond and GROMACS [14]. NAMD also offers biomolecular utilities in addition to common molecular simulation functionalities.

## Outstanding Challenges

### Long Timestep Approaches

One of the key challenges in scalable  $N$ -body computations is the serialization bottleneck of synchronized timesteps. Velocity Verlet is the most widely used scheme for integrating the Newtonian equations of motion in MD simulations due to its simplicity and symplectic nature. Studies of nonlinear stability indicate that the timestep lengths using Verlet integrators are upper bounded by  $\frac{P}{\sqrt{2\pi}}$  [19, 20], where  $P$  is the periodicity of the fastest oscillation in a given system. Considering the presence of Hydrogen atoms in most simulations, we would have  $P \approx 10$  fs, and so the timestep length  $\delta t$  should be less than 2.2 fs. In fact, timesteps of 2–5 fs are used in most classical MD simulations.

Lagrange multiplier-based constraining algorithms such as SHAKE [21], RATTLE [17], and LINCS [22] constrain the bond lengths and freeze the fastest oscillations in the target system, allowing for longer timesteps. However, these methods are not extensible to the next highest frequency oscillations due to strong vibrational coupling. Even though SHAKE and RATTLE are efficient serial algorithms, their parallel implementations do not currently scale well due to the iterative schemes needed for solving the linear systems involved. LINCS can, however, be made efficient and scalable [14].

Another way to achieve longer timesteps is hierarchical timestepping, also known as multiple timestep methods (MTS). These methods compute slower changing interactions less frequently, using nested loops, reducing average computation per timestep. MTS methods can be parallelized like regular MD methods, since they only change the frequency of computation of certain forces; the way these forces are computed stays the same.

### Accelerated Dynamics

Parallel MD does not scale to long timescales because of the time-serialization. Many interesting phenomena, however, take place at timescales beyond microseconds. Accelerated dynamics methods such as parallel replica dynamics, hyperdynamics, temperature accelerated dynamics, and on the fly kinetic Monte Carlo methods aim to extend the timescale limitations of

molecular simulations by exploiting ideas from Transition State Theory (TST) [23].

In a system with  $N$  atoms, the potential energy surface is a function in  $3N$  dimensions. This potential energy surface can be thought of as a composition of local minima separated by dividing surfaces. In a typical molecular simulation, the trajectory spends most of its time wandering around a basin making only infrequent escapes to neighboring basins (which intuitively corresponds to a significant conformational change or a reaction). Accelerated dynamics methods focus on capturing the jumps from one basin to another accurately while accelerating the trajectory through its path inside a single basin by alternate means.

Parallel replica dynamics [24] is the simplest and most accurate of the accelerated dynamics techniques. As the name suggests, it replicates a system across processors, goes through a short dephasing stage where correlations between these initially identical copies are annihilated, and lets each copy run independently. When an escape is detected at any of the processors, all processors except the one that detects the escape are stopped. The trajectory that made the escape is traced further to detect any correlated escapes. The method itself is inherently parallel. Communication is minimal but parallel efficiency is limited by both the dephasing stage (which does not advance the system clock) and the correlated event stage, during which only one processor computes. The system size to be simulated is bounded by the capabilities of a single node.

Hyperdynamics [25] is inspired by the basic concept of importance sampling. The potential energy surface in a simulation is modified by adding a nonnegative bias potential, which must be zero at/around the dividing surfaces. This way the energy barrier in front of a state transition is lowered giving rise to an enhanced rate of trajectory escapes. While the dynamics inside a basin become meaningless with the addition of a bias potential, the evolution from state to state is correct, because the relative TST rates for escape paths are not altered. The acceleration comes from the boosting of timestep lengths around the minima. In hyperdynamics, time is advanced at each step by the regular timestep length scaled by the inverse Boltzmann factor for the bias potential at that point. The ideal bias factor should be easy to compute and gives large boost

factors. Different bias potentials have been reported in literature: one that is based on the lowest eigenvalue of the Hessian [26], flat bias potential [27], and local bias potential [28]. Boost factors upto 10,000 have been reported using hyperdynamics [23].

A more approximate method is the Temperature Accelerated Dynamics (TAD) [29], which relies on harmonic TST approximation. The idea here is to run the simulation at a high temperature triggering more frequent trajectory escapes from a start basin. The times of escapes from the start basin are projected to their corresponding times at the original simulation temperature by using concepts from TST. Since the order of escapes at high temperature are different from those at the original temperature, once the earliest escape is detected with high probability, the TAD procedure is started again in the new basin. Boost factors achieved by TAD can be on the order of  $10^7$  thus reaching seconds of simulation time for certain systems [29].

The most important problem with accelerated dynamics methods is that they are practical only for small systems (tens to hundreds of atoms). Consequently, the underlying challenges associated with parallelization are not interesting. The most straightforward way of parallelization would be to couple the parallel replica dynamics with other accelerated methods such as hyperdynamics and TAD, that is, to use parallel replica scheme but instead of running regular MD on each processor, use hyperdynamics or TAD. Accelerated dynamics has been used to understand diffusion dynamics in solid state systems and folding of proteins. As these methods are further developed and used, scaling with system size and parallelization issues may emerge as important problems.

## Bibliography

1. Aluru S (1996) Greengard's n-body algorithm is not  $O(n)$ . *SIAM J Sci Comput* 17(3):773–776
2. Appel AW (1985) An efficient program for many-body simulation. *SIAM J Comput* 6
3. Barnes J, Hut P (1986) A hierarchical  $O(n \log n)$  force calculation algorithm. *Nature* 324
4. Callahan PB, Kosaraju SR (May 1992) A decomposition of multi-dimensional point-sets with applications to k-nearest-neighbors and n-body potential fields. In: Proceedings of 24th annual ACM symposium on theory of computing, Victoria, BC, Canada, pp 546–556
5. Esselink K (1992) The order of Appel's algorithm. *Inform Process Lett* 41:141–147
6. Grama A, Kumar V, Sameh A (1994) Scalable parallel formulations of the Barnes–Hut method for n-body simulations. In: Supercomputing '94 proceedings
7. Grama A, Gupta A, Karypis G, Kumar V (2004) Introduction to parallel computing. Addison Wesley, Reading, MA
8. Greengard L (1987) The rapid evaluation of potential fields in particle systems. MIT Press, Cambridge
9. Greengard L, Gragg W (1987) A parallel version of the fast multipole method. *Parallel Process Sci Comput*, pp 213–222
10. Greengard L, Rokhlin V (1987) A fast algorithm for particle simulations. *J Comp Phys* 73:325–348
11. Singh J, Holt C, Hennessy J, Gupta A (1993) A parallel adaptive fast multipole method. In: Proceedings of the supercomputing '93 conference
12. Warren M, Salmon J (1993) A parallel hashed oct-tree N-body algorithm. In: Supercomputing '93 proceedings, IEEE Comp. Soc. Press, Washington, DC, pp 12–21
13. Phillips JC, Braun R, Wang W, Gumbart J, Tajkhorshid E, Villa E, Chipot C, Skeel RD, Kal L, Schulten K (2005) Scalable molecular dynamics with NAMD. *J Comput Chem* 26(16):1781–1802
14. Hess B, Kutzner C, van der Spoel D, Lindahl E (2008) GROMACS 4: algorithms for highly efficient, load-balanced, and scalable molecular simulation. *J Chem Theory Comput* 4: 435–447
15. Bowers KJ, Dror RO, Shaw DE (2007) Zonal methods for the parallel execution of range-limited N-body simulations. *J Comp Phys* 221:303–329
16. Schlick T, Skeel RD, Brunger AT, Kale LV, Board JA Jr, Hermans J, Schulten K (1999) Algorithmic challenges in computational molecular biophysics. *J Comput Phys* 151:9–48
17. Andersen HC (1983) Rattle: a velocity version of the SHAKE algorithm for molecular dynamics calculations. *J Comput Phys* 52:24
18. Garcia-Archilla B, Sanz-Serna JM, Skeel RD (1998) Long-time-step methods for oscillatory differential equations. *SIAM J Sci Comput* 20:930
19. Mandziuk M, Schlick T (1995) Resonance in the dynamics of chemical systems simulated by the implicit midpoint scheme. *Chem Phys Lett* 237:525
20. Schlick T, Mandziuk M, Skeel RD, Srinivas K (1998) Nonlinear resonance artifacts in molecular dynamics simulations. *J Comput Phys* 139:1
21. Ryckaert JP, Ciccotti G, Berendsen HJC (1977) Numerical integration of the Cartesian equations of motion of a system with constraints: Molecular dynamics of n-alkanes. *J Comput Phys* 23:327
22. Hess B, Bekker H, Berendsen HJC, Fraaije JGEM (1997) LINCS: a linear constraint solver for molecular simulations. *J Comput Chem* 18:1463–1472
23. Voter AF, Montalenti F, Germann TC (2002) Extending the time scale in atomistic simulation of materials. *Annu Rev Mater Res* 32:32146

24. Voter AF (1998) Parallel replica method for dynamics of infrequent events. *Phys Rev B* 57:22
25. Voter AF (1997) Hyperdynamics: accelerated molecular dynamics of infrequent events. *Phys Rev Lett* 78:3908–3911
26. Voter AF (1997) A method for accelerating the molecular dynamics simulation of infrequent events. *J Chem Phys* 106:4665
27. Steiner MM, Genilloud PA, Wilkins JW (1998) Simple bias potential for boosting molecular dynamics with the hyperdynamics scheme. *Phys Rev B* 57:17
28. Fang QF, Wang R (2000) Atomistic simulation of the atomic structure and diffusion within the core region of an edge dislocation in aluminum. *Phys Rev B* 62:9317–9324
29. Montalenti F, Voter AF (2001) Applying accelerated molecular dynamics to crystal growth. *Phys Stat Solidi B* 226:21–27
30. Henkelman G, Jonsson H (2001) Long time scale kinetic Monte Carlo simulations without lattice approximation and predefined event table. *J Chem Phys* 115:9657
31. Frenkel D (2002) Understanding molecular simulation. Academic, New York
32. Ewald PP (1921) The calculation of optical and electrostatic grid potential. *Ann Phys* 64:253
33. Mardline RA, Aarseth SJ (2001) Tidal interactions in star cluster simulations. *Mon Not R Astron Soc* 321:398
34. Fryxell B, Olson K, Ricker P, Timmes FX, Zingale M, Lamb DQ, MacNeice P, Rosner R, Truran JW, Tufo H (2000) FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *Astrophysical J* 131(Suppl.):27334
35. Nosé S (1984) A unified formulation of the constant temperature molecular dynamics methods. *J Chem Phys* 81:511–519
36. Andersen HC (1980) Molecular dynamics simulations at constant pressure and/or temperature. *J Chem Phys* 72:2384
37. Berendsen HC, Postma JPM, van Gunsteren WF, DiNola A, Haak JR (1984) Molecular dynamics with coupling to an external bath. *J Chem Phys* 81:3684
38. Parrinello M, Rahaman A (1981) Polymorphic transitions in single crystals: a new molecular dynamics method. *J Appl Phys* 52:7182
39. Louwense MJ, Baerends EJ (2006) Calculation of pressure in case of periodic boundary conditions. *Chem Phys Lett* 421:138
40. van Duin ACT, Dasgupta S, Lorant F, Goddard WA (2001) ReaxFF: a reactive force field for hydrocarbons. *J Phys Chem A* 105:9396

## nCUBE

The nCUBE machines [1] were hypercube-connected multicomputers designed and built by nCube Corporation in the 1980s. Several models were built starting with the nCUBE/ten.

## Related Entries

- [Hypocubes and Meshes](#)

## Bibliography

1. CORPORATE Ncube. (1988) The NCUBE family of high-performance parallel computer systems. In: Geoffrey Fox (ed) Proceedings of the third conference on Hypercube concurrent computers and applications: architecture, software, computer systems, and general issues - vol 1 (C3P), vol 1. ACM, New York, pp 847–851

## NEC SX Series Vector Computers

HIROSHI TAKAHARA

NEC Corporation, Tokyo, Japan

## Synonyms

- [SIMD \(Single Instruction, Multiple Data\) Machines](#)

## Definition

The SX Series is the parallel vector supercomputer that has been provided by NEC as its flagship high-performance computing system. After the advent of its pioneering models of SX-1/2 back in 1983, NEC has continuously been enhancing this series toward the SX-9, which has the world's fastest single CPU core performance of 102.4 GFLOPS. It makes up 1 node system of 1.6 TFLOPS peak performance, which can be configured up to 512 nodes enabling almost petascale computing with the maximum vector performance of 839 TFLOPS. The technology basis of the SX Series has led to the realization of the Earth Simulator at JAMSTEC (Japan Agency for Marine-Earth Science and Technology) that has earned the number one position on the Linpack benchmark during 2002–2004 for five times in the row, as well as its successor model (renewed Earth Simulator) put into operational use in 2009. The SX Series has been utilized for a spectrum of applications ranging from scientific research and mission-critical weather forecasting to engineering and material design.

## Discussion

### History of the Supercomputer SX Series

In April 1983, NEC made an entry in the supercomputer market with its simultaneous announcement for the

first two models in the series: the SX-1 and the SX-2 – the world's fastest supercomputers at the time. The SX-2 achieved a peak performance of 1.3 GFLOPS (1.3 billion floating-point operations per second), marking the beginning of the era of gigaflop computing.

The successor model, SX-3, announced in April 1990 attained new world speed with its peak performance of 22 GFLOPS using four multiprocessors and maximum 5.5 GFLOPS per CPU. Offering a lineup of commercial application software, the SX-3 not only employed the shared-memory-type multiprocessor combined with the parallel processing technology, but also pioneered the era of open systems featuring a 64-bit SUPER-UX operating system, which was developed based on UNIX.

With an increase in expandability to a maximum of 512 processors, the SX-4 was announced in November 1994, achieving a maximum performance of 1 TFLOPS.

The SX-4 adopted CMOS for the CPU instead of conventional silicon bipolar LSI, which resulted in both reduced power consumption and higher density packaging, consequently leading to a cost reduction by allowing the user to utilize air cooling instead of previously used liquid-cooling systems. The system has enabled expansion in memory capacity by the use of DRAM instead of SRAM as the memory device and the consequent improvement in a price performance ratio combined with a comprehensive list of application software.

In June 1998, the SX-5, the fourth generation in this series, was unveiled. Doubling both the clock frequency and the number of vector pipelines, the evolved model achieved a CPU vector performance of 8 GFLOPS, resulting in the system peak vector performance of 4 TFLOPS with a configuration of parallel processing with 512 CPUs. The release of the fifth generation SX-6 in October 2001 enabled the integration in a single chip of what required 30 LSI chips in the previous CPU model (SX-5).

With 1,024 CPUs each having a maximum CPU performance of 8 GFLOPS, the SX-6 delivered a peak performance of 8 TFLOPS. Equipped with a maximum of 32 CPUs per node and a maximum of 256 GB of shared memory combined with automatic parallelization, the SX-7 introduced in October 2002 realized further ease of operation.

Also during this period, the delivery of the Earth Simulator, for which NEC was in charge of the hardware and basic software development, was completed. The Earth Simulator began operation in March 2002, and its performance not only stunned the world, but also has continued to make significant contributions to understanding challenging environmental issues such as global climate change and advancing a wide spectrum of scientific findings and industrial applications.

Shipping of the sixth generation SX-8 started at the end of 2004, achieving a CPU performance of 16 GFLOPS and a system peak performance of 65 TFLOPS with a maximum of 4,096 CPUs. Subsequently in October 2007, the seventh generation SX-9 was announced. Featuring the world's fastest 102.4 GFLOPS performance per single core, the large-scale shared memory up to 1 TB, and the interconnects with a data transfer rate of 128 GB/s, it can deliver near-PFLOP peak performance of 839 TFLOPS. In addition, the improved LSI technology and high-density packaging technology have reduced both power consumption and required installation space to approximately a quarter of that required for conventional supercomputers.

[Figure 1](#) shows the pictures of the major models of the SX Series. The evolution of the SX Series is summarized with its specifications in [Table 1](#).

## SX-9 Architecture

In subsequent sections, the description of the SX Series is made with the particular focus on the latest model



**NEC SX Series Vector Computers. Fig. 1** Pictures of the SX Series hardware

**NEC SX Series Vector Computers. Table 1** SX Series system specification

| Model  | Year of announcement | Single CPU core performance | Memory bandwidth per CPU (GB/s) | Number of CPUs per node | Clock cycle (ns) | Single node performance (GFLOPS) | Main memory capacity (GB) | Memory bandwidth per node (GB/s) | Max. system performance (GFLOPS) |
|--------|----------------------|-----------------------------|---------------------------------|-------------------------|------------------|----------------------------------|---------------------------|----------------------------------|----------------------------------|
| SX-1/2 | 1983                 | 1.3                         | 10.7                            | 1                       | 6                | 1.3                              | 0.256                     | 10.7                             | 1.3                              |
| SX-3   | 1989                 | 5.5                         | 44                              | 4                       | 2.9              | 22                               | 8                         | 176                              | 22                               |
| SX-4   | 1994                 | 2.0                         | 16                              | 32                      | 8                | 64                               | 8                         | 512                              | 1,024                            |
| SX-5   | 1998                 | 8.0                         | 64                              | 16                      | 4                | 128                              | 128                       | 1,024                            | 4,096                            |
| SX-6   | 2001                 | 8.0                         | 32                              | 8                       | 1                | 64                               | 64                        | 256                              | 8,192                            |
| SX-7   | 2002                 | 8.83                        | 35.3                            | 32                      | 0.9              | 283                              | 256                       | 1,130                            | 18,083                           |
| SX-8   | 2004                 | 16                          | 64                              | 8                       | 0.5              | 128                              | 128                       | 512                              | 65,536                           |
| SX-8R  | 2006                 | 35.2                        | 70.4                            | 8                       | 0.455            | 282                              | 256                       | 563                              | 144,179                          |
| SX-9   | 2007                 | 102.4                       | 256                             | 16                      | 0.3125           | 1,638                            | 1,024                     | 4,096                            | 838,861                          |

SX-9. The development of the SX-9 aims at facilitating both high performance for real application programs and economical operation.

The SX-9 has been developed with the following features:

### 1. Fast single-chip vector processor

Using the 65 nm CMOS technology, the SX-9 realizes 102.4 GFLOPS per processor – the world's fastest single core performance.

### 2. Multiprocessor system with excellent scalability

A single SX-9 node has a flat shared memory configuration for a maximum of 16 CPUs with a peak single-node performance of 1.638 TFLOPS and a maximum memory capacity of 1 TB. This shared memory node is interconnected with a special high-speed switch at a maximum of 128 GB/s per node, enabling configurations of up to a maximum of 512 nodes (839 TFLOPS) to provide both shared and distributed memory systems with excellent scalability.

### 3. Superior energy-saving and installation advantages

In addition to the reduced processor energy consumption and heat generation, the system adopts the high-efficiency cooling technology, resulting in improved system power consumption and easy installation.

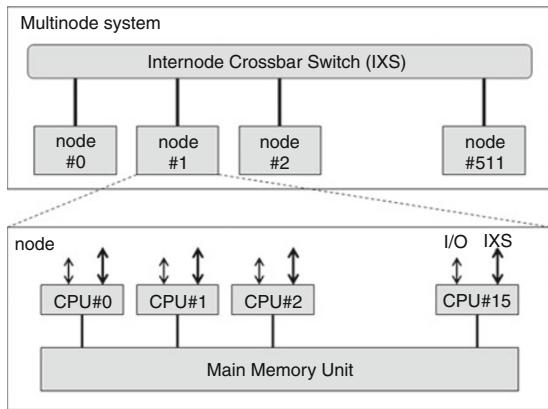
### Hardware Overview

Figure 2 shows the SX-9 system configuration. The product lineup ranges from a shared-memory type single-node model that tightly integrates a maximum of 16 CPUs and the main memory unit (MMU) into a multinode system model connecting each node in a cluster configuration via a high-speed internode cross-bar switch (IXS). The single-node system is available in two models: the model A with a maximum number of CPUs of 16 (1.638 TFLOPS), a maximum main memory capacity of 1 TB, and a maximum 32 input/output slots, and the model B with eight CPUs at a maximum performance of 819.2 GFLOPS, a maximum main memory capacity of 512 GB, and a maximum of 16 input/output slots. The multinode system interconnects 2–512 nodes in a cluster configuration via IXS with up to 8,192 CPUs.

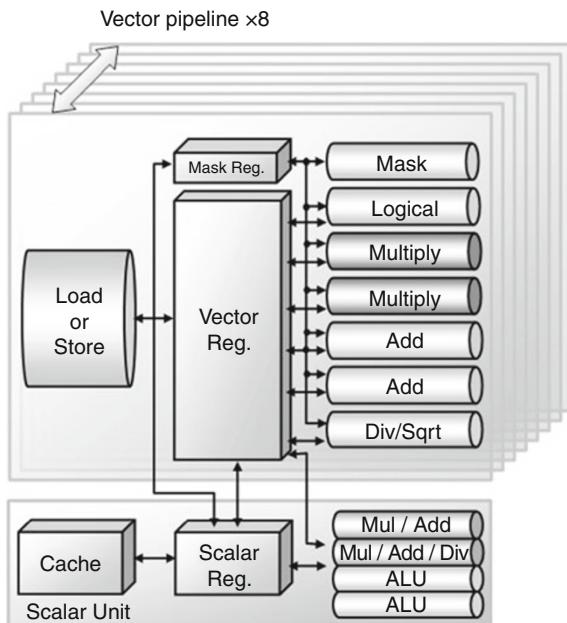
The CPU of the SX-9 uses eight sets of vector pipelines to achieve a peak performance of over 100 GFLOPS with a single unit. The memory with a maximum capacity of 1 TB can be shared by up to 16 CPUs, the maximum data transfer rate between the CPU and the MMU is 4 TB/s, and that between the I/O unit and the MMU is 64 GB/s × 2.

### Processor

The CPU is composed of a vector unit and a scalar unit, both of which are connected to the MMU through the processor-memory network. The SX-9 features the addition of the ADB (Assignable Data Buffer) within a



**NEC SX Series Vector Computers. Fig. 2** SX Series system configuration



**NEC SX Series Vector Computers. Fig. 3** SX-9 CPU configuration

CPU for the selective buffering of data. [Figure 3](#) shows the configuration of the CPU of the SX-9.

### Vector Unit

The vector unit is composed of the vector operation block and the vector control block.

### Vector Operation Block

The vector operation block has basic operations including the Logical, Multiply, Add, and Divide/Sqrt, as well

as the Mask and Load/Store functions. This block also includes 16 mask registers and 72 vector registers. The vector operation block supports the IEEE double and single-precision floating point data formats.

### Vector Operation Pipeline

The vector operation block consists of eight pipelines. To reduce the time to transfer the result of one vector operation to another operation, the vector unit has the enhanced data forwarding capability (chaining), improving the processing speed of programs with a short vector loop length. The vector pipelines have vector data compression and decompression capabilities for the efficient use of memory bandwidth.

### Vector register

The vector operation block has 72 vector registers. Each vector register consists of 256 64 bit-wide registers. The total capacity of the registers reaches 144 KB.

### Vector mask register

Some vector operations can be processed with a 256 bit mask, which can be switched to enable/disable the operation for each element corresponding to the mask bit. Mask bits are generated in the pipeline according to the result of logical operations. There are 16 mask registers prepared, each of which is 256 bit-wide.

### Vector control block

The SX-9 CPU has a vector control block which enables the out-of-order execution of vector instructions, contributing to hiding memory latency. Furthermore, this block controls chaining to perform vector processing with high efficiency.

### Scalar Unit

The scalar unit of the SX-9 employs the 64-bit RISC architecture that is compatible with the SX Series products and incorporates 128 general-purpose 64-bit registers and two 32 KB L1 caches for instructions and data. Its features are as described below.

### 6-way super-scalar configuration

The SX-9 uses a super-scalar configuration with a total of six execution pipelines (two for the floating-point arithmetic operations, two for integer arithmetic

operations, and two for memory operations), thus improving the performance of application codes with a high degree of instructionlevel parallelism.

### Instruction execution control

The out-of-order execution mechanism contributes to the high performance by executing instructions that become executable regardless of the instruction order. For the fast operation over a large domain, the SX-9 enables search and retrieval across up to eight branch instructions. It also adopts a speculative instruction execution mechanism that performs tentative execution of subsequent instructions before the actual execution of a branch instruction and restarts an instruction with the appropriate condition in a case when the branch prediction fails. In order to expand the instruction window in the out-of-order execution mechanism, the SX-9 has the reorder buffer entries enhanced to 64 entries.

### ADB (Assignable Data Buffer)

The ADB is located within a CPU for the selective buffering of data. The ADB enables a more efficient data transfer between the ADB and the vector unit with shorter latency than that between the processor and the memory, resulting in higher performance by storing frequently used data in the ADB and the reduced memory bank contention arising from the concurrent processing. The data stored in the ADB can be set to vector data only, scalar data only, or both vector and scalar data depending on the running application. The ADB for each vector load/store instruction is assigned for retaining the necessary data in the ADB as much as possible. A software-controlled prefetch instruction is supported.

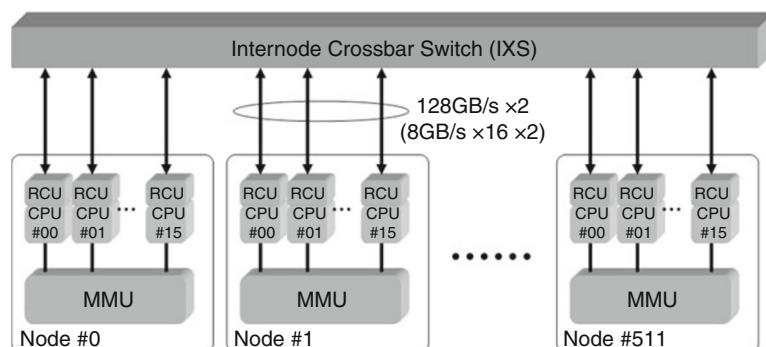
### Multinode Configuration

The multinode system of the SX-9 can connect up to 512 nodes by grouping shared-memorytype single nodes into clusters and connecting them to the IXS ultra-high-speed crossbar switch. The multinode system features not only a wide bandwidth of internode data transfer but also reduced communication latency by capitalizing on the RCU (remote access control unit), which is the IXS connection unit for each node. [Figure 4](#) shows the configuration of the SX-9 multinode system. Each node incorporates up to 16 RCUs, which are connected to the IXS via cables. Each RCU forms a single lane, which has two connection ports of  $4\text{ GB/s} \times 2$ , offering a transfer rate of  $8\text{ GB/s} \times 2$ . As a result, each node can incorporate a maximum of 16 lanes with 32 connection ports, and the total transfer performance reaches a maximum of  $128\text{ GB/s} \times 2$ .

### Main Memory Unit (MMU)

The MMU adopts the shared memory system and is composed of 512 MMU cards for its maximum configuration. The memory of the SX-9 has a large capacity thanks to the use of DDR-SDRAM (Double Data Rate-SDRAM) devices for all the MMUs.

The memory capacity per MMU card is 2GB, and the system supports the capacity from 256 GB up to 1TB. The MMU cards are capable of handling concurrent operations of memory access requests from the CPU with a data transfer rate of 8GB/s per MMU card, or a maximum of 4TB/s for the entire system. The maximum 32,768-way parallel memory interleaving is adopted for the efficient data transfer with DDR3-SDRAM.



NEC SX Series Vector Computers. [Fig. 4](#) SX-9 multinode system configuration

## I/O Processing Unit

All of the processors in the system are allowed to access all of the I/O devices. The SX adopts the direct I/O method, with which the memory in the host bus adapter (HBA) is accessed directly. The I/O interface can be selected according to the types of packaged channel cards, facilitating the system expansion. The I/O processing unit of the SX-9 is composed of the IO Features (IOFs: host bridge units) and the PCI Express control units. Up to 16 IOFs can be mounted per system and two PCI Express control units can be connected to each IOF. As each IOF is virtually represented as a singlechannel device, all of the functions incorporated in the IOFs can be set and modified at the desired timing from the software using the same access method as a general purpose channel card.

## Core Technology Behind the SX-9

Figure 5 summarizes some of the core technologies built into the SX Series for enhanced performance and ease of use. The SX Series has the advanced architecture of large-scale shared memory, high-speed data transfer between the CPU and the memory, and an ultra-high-speed network interconnecting nodes. One of the key

technologies indispensable to achieving higher system efficiency is the interconnection between processors. NEC has developed an optical interconnection technology that achieves a 20 Gbps signal transmission rate between two LSIs of a high-performance computer, thus surpassing the communication speed made available with the existing electrical transmission.

## Software Technology of the SX-9

### Outline of the SUPER-UX Operating System for the SX-9

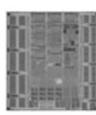
The SUPER-UX is an operating system based on the UNIX System V operating system that features functions inherited from the BSD and SVR4 2MP as well as enhancements of the functions required to support supercomputers. The SUPER-UX features the flexible resource management and the high parallel processing capabilities of kernels and I/Os in order to guarantee high scalability. Four page sizes are supported, which are 32 KB for general commands, 4 MB for compiler and system commands, and 64 and 256 MB for large-scale user programs. This strategy improves the execution performance of programs that use large arrays

### Technologies for the SX-9

World fastest single-core processor (102GFLOPS)  
→ 1.6TFLOPS/node

#### LSI technology

CMOS LSI with the 65nm design rule



High performance technology

- Parallel processing technology
- Cluster control technology

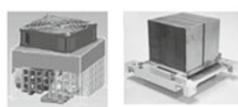
#### High-speed interface technology

➤ High-speed interface for high performance optical interconnection



#### Cooling technology

Cooling for the heat generation of 200W+

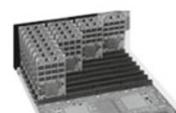


#### Supercomputer

#### SX-9



#### High-density packaging technology



NEC SX Series Vector Computers. Fig. 5 Technologies for the SX-9

and reduces the overheads in the memory management. The SUPER-UX of the SX-9 has expanded the user's virtual space to 8 TB, enabling an efficient layout of parallel programs and MPI programs. The SX Memory File Facility (SX-MFF) is also provided for high-speed I/O by building a conventional file system on the large-capacity memory as a disk cache.

### **Basic OS Functions**

The SUPER-UX supports not only large-scale multiprocessors with efficient resource management but also high-speed input/output and the sophisticated gang scheduling for maximizing parallel processing performance.

### **File Management Functions**

The SUPER-UX makes it possible to create large-scale files and file systems. While retaining the advantages of a standard UNIX system, it realizes a highspeed file system. The high-speed shared file system gStorageFS enables the efficient data transfer comparable to local file systems without involving the CPUs on remote servers. The NFS V3 compatibility is maintained as a user interface.

### **Batch Processing Functions**

The SUPER-UX supports the concepts of jobs (sets of processes) at the kernel level. NQSII is the batch processing system that enables appropriate processing by large-scale clusters, and JobManipulator is provided as an NQSII scheduler extension to maximize system operational efficiency with backfill scheduling. NQSII incorporates functional enhancements of NQS job management, resource management, and load balancing to adapt to cluster systems with the improved single system image (SSI) and system operability.

### **Operation Management Functions**

SUPER-UX supports a checkpoint/restart function for the interruption of a program in progress at a user-specified point and its smooth resumption. The unified operation management software provides the integrated management of multiple host machines on the network from a single machine, as well as automatic operation control, thus resulting in reduced operation cost.

### **Software Development Environment**

The compilers for Fortran, C, and C++ provide highlevel optimization and automatic vectorization/

parallelization features for maximizing the performance. The vector data buffering function, which utilizes the ADB, is open to the user, thus enabling highly effective memory access optimization. Both the MPI library and the HPF compiler are provided for comprehensive distributed memory programming. The PSUITE tool is also available for the GUI-based program development, debugging, and tuning.

### **Compiler and Library**

The SX-9 provides FORTRAN90/SX and C++/SX, respectively, a Fortran compiler and a C/C++ compiler; both feature vectorization and parallelization functions. HPF/SX V2 (a compiler for High Performance Fortran, which is a standard language for distributed parallel processing) is also provided, as well as MPI/SX and MPI2/SX (fully compliant with the distributed parallel processing interfaces MPI-1.3 and MPI-2.1). The system also supports a number of the Fortran 2003 features. Both FORTRAN90/SX and C++/SX support the OpenMP standard API for sharedmemory type parallel processing.

Two proprietary mathematical libraries are provided; one is the ASL scientific library aimed at a wide usage in scientific and engineering applications with enhanced performance on key functions, such as linear algebra and FFT. The other is MathKeisan, which collects a set of public domain mathematical library components, such as BLAS.

### **Application Program and Performance**

The application of the SX Series spans a wide spectrum of areas ranging from fundamental research such as nanomaterial sciences and environmental sciences to industrial designing, drug designing, and emerging life sciences. The SX Series realizes high performance on real application programs, as well as its well-balanced performance from various architectural perspectives. Meantime, the Earth Simulator at JAMSTEC has inherited the parallel vector architecture, realizing the peak performance of 131 TFLOPS with its updated model (160 nodes of SX-9/E). This system is ahead of the pack with respect to the HPC Challenge benchmark (HPCC), which is gaining popularity separately from the Linpack score. The updated Earth Simulator topped the Global FFT (Fast Fourier Transform), one of the measures of the HPC Challenge Awards,

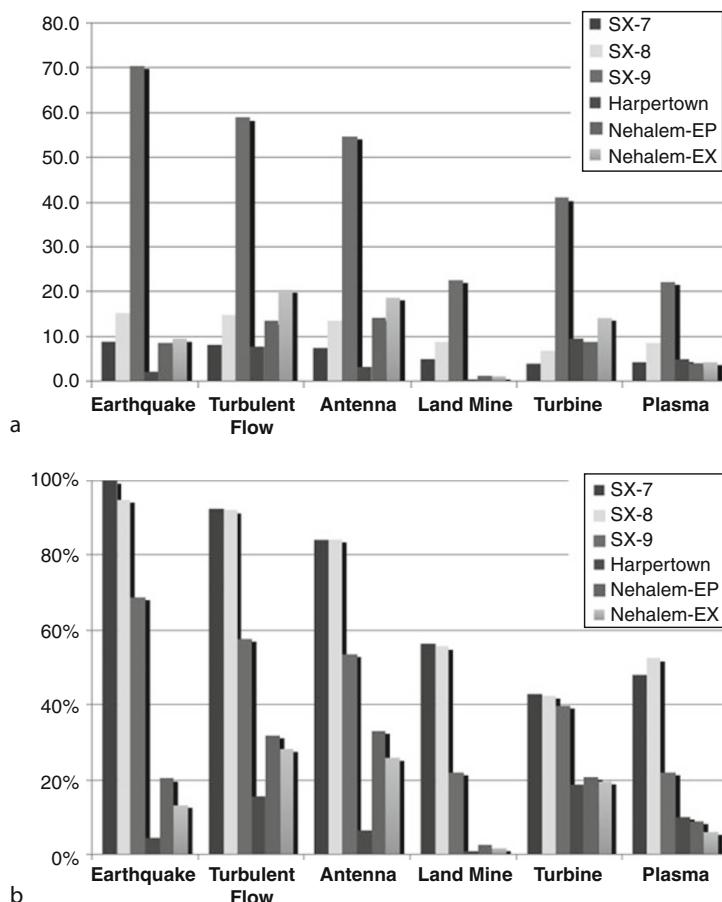
with the performance number of 11.876 TFLOPS. The HPCC is designed to give multiple measures on performance under the US governmental project, including such factors as memory bandwidth and interconnect, as well as processor capability, all of which are suitable to the evaluation of the capabilities of a computer more comprehensively.

The Earth Simulator showcases excellent performance and peak performance ratio on scientific application programs in such an area as earth environmental modeling.

**Figure 6** shows the performance numbers on real scientific applications evaluated for six benchmark programs on various computer platforms, including the SX Series. The target applications are: (1) the computations of inter-plate earthquakes in a subduction zone, (2) the turbulent flow based on the direct

numerical simulation, (3) the electromagnetic analysis about various antennas, (4) the land mine detection with the Synthetic Aperture Radar (Maxwell's equations), (5) the unsteady flows around a turbine based on the direct numerical simulation, and (6) the wave-particle interactions between electrons and plasma waves.

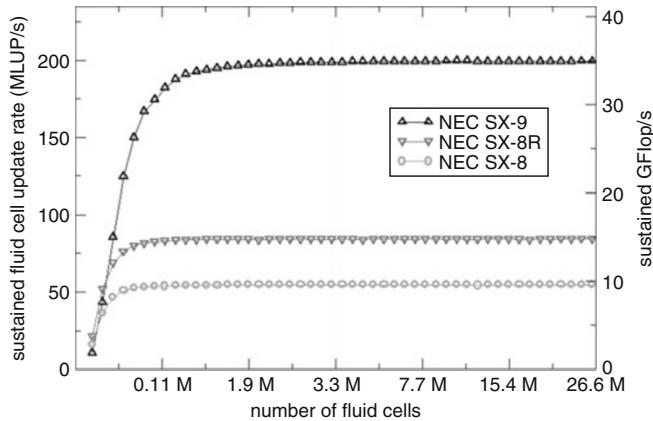
**Figure 6** includes the comparison of single CPU performance of these benchmark programs for various NEC vector supercomputers and Intel multicore processors (Harpertown, Nehalem-EP, and Nehalem-EX) as indicated in **Table 2**. The Nehalem processors are directly connected to the memory system, enabling efficient data transfer. Each application program is set up to utilize all the cores for the Intel processors. As clearly seen in **Fig. 6**, the SX-9 processor shows much higher performance over other processors due to its high



**NEC SX Series Vector Computers. Fig. 6** Comparison of single-core performance for scientific application programs. (a) Measured performance (GFLOPS), (b) Peak performance ratio (%). Each bar represents SX-7, SX-8, SX-9, Harpertown, Nehalem-EP, and Nehalem-EX respectively from left to right

**NEC SX Series Vector Computers. Table 2** Specifications of referenced hardware platforms for performance evaluation

| Computer model | Clock (GHz) | Peak performance (GFLOPS) | Memory bandwidth (GB/s) | Number of cores | BYTES/FLOP | Cache size                   |
|----------------|-------------|---------------------------|-------------------------|-----------------|------------|------------------------------|
| SX-9           | 3.2         | 102.4                     | 256                     | 1               | 2.5        | ADB: 256 KB                  |
| SX-8           | 2           | 16                        | 64                      | 1               | 4          | –                            |
| SX-7           | 1.1         | 8.83                      | 35.3                    | 1               | 4          | –                            |
| Harpertown     | 3.16        | 50.56                     | 10.67                   | 4               | 0.2        | L2: 12 MB                    |
| Nehalem-EP     | 2.66        | 42.56                     | 25.6                    | 4               | 0.6        | L2: 256 KB/core<br>L3: 8 MB  |
| Nehalem-EX     | 2.26        | 72.48                     | 34.1                    | 8               | 0.46       | L2: 256 KB/core<br>L3: 24 MB |

**NEC SX Series Vector Computers. Fig. 7** Single-core performance of Lattice-Boltzmann CFD program. (Left) Cell update ratio (in Million updates per second) (Right) Measured performance (in GFLOPS)

theoretical peak performance for all the six programs. In addition, the execution efficiency (peak performance ratio) is more than 40% for many application programs on the SX Series, which is significantly higher than the commodity scalar processors. The actual performance varies depending on such factors as the memory access frequency, which depends on the ratio of the CPU-memory bandwidth and the floating-point operation capability.

Figure 7 shows the performance of a Lattice Boltzmann (LB) CFD program measured with a single CPU on different models of the SX Series for various numbers of fluid cells. The LB method emerges from a highly simplified gas-kinetic description. It represents the behavior of fluid by the translational movement

and the collision of virtual particles. The left axis indicates the fluid cell update rate (per second), while the right one represents the measured performance (in GFLOPS). Taking into account the maximum theoretical performance of each CPU (SX-9: 102.4 GFLOPS, SX-8R: 32 GFLOPS, SX-8: 16 GFLOPS), the computational efficiency (peak performance ratio) is proven to be high (35–60%) for a wide range of the number of fluid cells, thus surpassing the efficiency on conventional scalar computers [1].

### Concluding Remarks

While the commodity-chip-based parallelism is getting pervasive, complex control mechanisms and massive

hardware resources are necessary for realizing high performance with the current scalar architecture, spurring the widening gap between the theoretical peak performance of a processor and the performance for real scientific application programs [2, 3]. The vector mechanism is a possible solution to such an issue; it employs a highly developed single instruction multiple data stream (SIMD) approach because of a larger number of arithmetic operations handled by one instruction, which is further enhanced with multiple pipelines. It provides an easy way to express the concurrency needed to take advantages of the memory bandwidth with more tolerance about the memory systems' latency than scalar designs, facilitating the realization of high performance needed for scientific computing.

## Related Entries

- [Compilers](#)
- [Cray Vector Computers](#)
- [Earth Simulator](#)
- [Fortran 90 and Its Successors](#)
- [Fujitsu Vector Computers](#)
- [HPC Challenge Benchmark](#)
- [HPF \(High Performance Fortran\)](#)
- [Hybrid Programming with SIMPLE](#)
- [Instruction-Level Parallelism](#)
- [MPI \(Message Passing Interface\)](#)
- [OpenMP](#)
- [Load Balancing](#)
- [Shared-Memory Multiprocessors](#)
- [SIMD \(Single Instruction, Multiple Data\) Machines](#)

## Bibliographic Notes and Further Reading

The architectural aspects of the SX Series are described in [4–6]. The fundamental descriptions about vector processors (or SIMD approach) and the general trend of the performance of high-performance computers are available, for example, in [2, 3]. There are several papers that focus on the computing performance of the SX Series concerning real scientific application programs (References [1, 7]). As for more comprehensive performance evaluations of the SX-9, for example, reference [8] is suggestive. In relation to such benchmarks,

the descriptions of the referenced scalar computers can be found in [9, 10]. The elaborate descriptions of the HPC Challenge benchmark and its awards are available in [11, 12]. It is also useful to refer to the overview of the Earth Simulator featuring the vector-processing capabilities that inherit from the SX Series for deeper understanding of the NEC vector supercomputers (references [13, 14]).

## Bibliography

1. Zeiser T, Hager G, Wellein G (2009) Benchmark analysis and application results for Lattice Boltzmann simulations on NEC SX vector and Intel Nehalem systems. *Parallel Process Lett (PPL)* 19(4):491–511. DOI: 10.1142/S0129626409000389
2. Gebis J, Patterson D (2007) Embracing and extending 20th-century instruction set architectures. *Computer* 40(4) (published by the IEEE Computer Society):68–75
3. The high-end computing revitalization task force (HECRTF) (2004) Federal plan for high-end computing. Technical report
4. NEC SX-9 Supercomputer (2008) <http://www.nec.com/de/prod/solutions/hpc-solutions/sx-9/index.html>
5. Supercomputer SX-9/Special Issue (2008) *NEC Tech J* 3(4). <http://www.nec.co.jp/techrep/en/journal/g08/n04/g0804mo.html#name3-1>
6. Takahara H (2009) HPC architecture from application perspectives. In: Resch M et al (eds) *High performance computing on vector systems 2009*. Springer, Berlin, pp 59–67
7. Soga T, Musa A, Shimomura Y, Okabe K, Egawa R, Takizawa H, Kobayashi H, Itakura H (2009) Performance evaluation of NEC SX-9 using real science and engineering applications. SC09 technical paper (ACM SIGARCH/IEEE Computer Society). <http://scyourway.supercomputing.org/conference/view/pap229>
8. Kobayashi H, Egawa R, Takizawa H, Okabe K, Musa A, Soga T, Isobe Y (2009) Lessons learnt from 1-year experience with SX-9 and toward the next generation vector computing. In: Resch M et al (eds) *High performance computing on vector systems 2009*. Springer, Berlin, pp 3–22
9. Intel corporation (2007) Intel Xeon processor 5400 series product brief. <http://www.intel.com/Assets/PDF/prodbrief/318664.pdf>
10. Intel corporation (2009) Intel Xeon processor 5500 series product brief. <http://www.intel.com/Assets/PDF/prodbrief/322355.pdf>
11. Luszczek P, Bailey D, Dongarra J, Kepner J, Lucas R, Rabenseifner R, Takahashi D (2006) The HPC challenge (HPCC) benchmark suite. [http://icl.cs.utk.edu/projectsfiles/hpcc/pubs/sc06\\_hpcc.pdf](http://icl.cs.utk.edu/projectsfiles/hpcc/pubs/sc06_hpcc.pdf)
12. 2010 HPC Challenge Awards (2010) <http://www.hpcchallenge.org/custom/index.html?lid=103&slid=238>
13. Oliker L et al (2005) Leading computational methods on scalar and vector HEC platforms. In: IEEE/ACM SC2005, Seattle
14. System overview of ES2 (JAMSTEC) (2009) <http://www.jamstec.go.jp/es/en/system/system.html>

## NESL

GUY BLELLOCH

Carnegie Mellon University, Pittsburgh, PA, USA

## Definition

NESL is a strongly typed functional programming language designed in the early 1990s to make it easy to program and analyze parallel algorithms. It is loosely based on the ML class of languages and supports parallelism through operations on sequences. Its main features are nested data-parallelism, deterministic parallelism, and a built-in cost model for analyzing the asymptotic performance of algorithms.

## Discussion

### Introduction

NESL is a high-level parallel programming language with an emphasis on concisely and clearly expressing parallel algorithms and the ability to run them on a variety of sequential, parallel, and vector machines [1–4]. It is a strongly typed call-by-value (strict) functional language loosely based on the ML language [5]. The language uses sequences as a primitive parallel data type, and parallelism is achieved exclusively through operations on these sequences. As such it is a data-parallel, or collection-oriented language [6]. The main features of NESL are:

1. The support of *nested parallelism*. NESL fully supports nested sequences, and the ability to apply any user-defined function over the elements of a sequence, even if the function is itself parallel and the elements of the sequence are themselves sequences. Such functionality is important for expressing parallel divide-and-conquer algorithms as well as irregular nested parallel loops. The support for nested parallelism differentiates it from most other data-parallel languages such as C\* [7], High Performance Fortran [8], and ZPL [9]. NESL was the first data-parallel language for which the implementation supported nested parallelism. More recently Data Parallel Haskell [10] also supports nested data parallelism.

2. The support of *deterministic parallelism*. Since NESL has no side effects and hence no race conditions, its semantics is sequential and deterministic [3]. In particular a program with the same input will always return the same output independently of how it is run. Input and output and other nonfunctional features are only allowed at top-level and cannot be called in parallel (there are some well-specified exceptions, such as the random number generator). The philosophy behind NESL is that it is much easier to design, analyze, and prove the correctness of algorithms if the semantics is deterministic. Deterministic parallelism is also supported by other purely functional parallel languages such as ID [11], Sisal [12], and Data Parallel Haskell [10], and has been topic of recent interest [13].
3. The support for a *cost semantics* and corresponding *provable implementation bounds* that maps costs derived by the semantics onto asymptotic runtime on specific machines [3]. The semantics define two costs, *work* and *depth*, for the evaluation of every expression, and simple composition rules for composing these costs across parallel and sequential constructs. The work corresponds to the sequential running time (the total number of instructions executed during the evaluation) and is composed by adding costs across constructs. The depth (sometimes referred to as the critical path or span) corresponds to the longest chain of dependencies and is derived by taking the maximum of costs when composing in parallel and adding costs when composing sequentially. The notions of work and depth have since been used in other languages such as Cilk [14].

NESL has been implemented on a variety of architectures, including both SIMD and MIMD machines, with both shared and distributed memory. The released compiler generates a portable intermediate code called VCODE [15], which runs on vector machines, distributed memory machines, and shared memory machines [2, 16]. The implementation supplies an interactive read-eval-print loop.

### Parallel Operations on Sequences

NESL supports parallelism through operations on sequences, which are specified using square brackets. For example,

```
[2, 1, 9, -3]
```

is a sequence of four integers. In NESL, all elements of a sequence must be of the same type, and all sequences must be of finite length. Parallelism on sequences can be achieved in two ways: the ability to apply any function concurrently over each element of a sequence, and a set of built-in parallel functions that operate on sequences. The application of a function over a sequence is achieved using set-like notation similar to *set-formers* in SETL [17] and *list-comprehensions* in Miranda [18] and Haskell [19]. For example, the expression

```
{negate(a) : a in [3, -4, -9, 5]};
⇒ [-3, 4, 9, -5] : [int]
```

negates each element of the sequence  $[3, -4, -9, 5]$ . This construct can be read as “in parallel for each  $a$  in the sequence  $\{3, -4, -9, 5\}$ , negate  $a$ .” The symbol  $\Rightarrow$  points to the result of the expression, and the expression  $[int]$  specifies the type of the result: a sequence of integers. The semantics of the notation differs from that of SETL, Miranda, or Haskell in that the operation is defined to be applied in parallel. This notation is referred to as the *apply-to-each* construct and each subcall on a sequence element as an *instance*. As with set-comprehensions, the apply-to-each construct also provides the ability to subselect elements of a sequence: the expression

```
{negate(a) : a in [3, -4, -9, 5] | a < 4};
⇒ [-3, 4, 9] : [int]
```

can be read as, “in parallel for each  $a$  in the sequence  $\{3, 4, 9, 1\}$  such that  $a$  is less than 4, negate  $a$ .” The elements that remain maintain their order relative to each other. It is also possible to iterate over multiple sequences. The expression

```
{a + b : a in [3, -4, -9, 5]
 ; b in [1, 2, 3, 4]};
⇒ [4, -2, -6, 9] : [int]
```

adds the two sequences elementwise. In NESL, any function, whether primitive or user defined, can be applied to each element of a sequence. So, for example, one can define a factorial function

```
function factorial(i) =
 if (i == 1) then 1
 else i*factorial(i-1);
```

```
⇒ factorial : int -> int
```

and then apply it over the elements of a sequence

```
{factorial(x) : x in [3, 1, 7]};
⇒ [6, 1, 5040] : [int]
```

In this example, the `function name(arguments) = body;` construct is used to define `factorial`. The function is of type `int -> int`, indicating a function that maps integers to integers. The type is inferred by the compiler.

In addition to the apply-to-each construct, a second way to take advantage of parallelism in NESL is through a set of sequence functions. The sequence functions operate on whole sequences and all have relatively simple parallel implementations. For example, the function `sum` sums the elements of a sequence.

```
sum([2, 1, -3, 11, 5]);
⇒ 16 : int
```

Since addition is associative, this can be implemented on a parallel machine in logarithmic time using a tree. NESL supports many other built-in functions on sequences, including scans, permuting a sequence, appending sequences, extracting subsequences, and inserting into a sequence.

## Nested Parallelism

In NESL, the elements of a sequence can be any valid data item, including sequences. This rule permits the nesting of sequences to an arbitrary depth. A nested sequence can be written as

```
[[2, 1], [7, 3, 0], [4]]
```

This sequence has type: `[[int]]` (a sequence of sequences of integers). Given nested sequences and the rule that any function can be applied in parallel over the elements of a sequence, NESL necessarily supplies the ability to apply a parallel function multiple times in parallel; this is referred to as *nested parallelism*. For example, one can apply the parallel sequence function `sum` over a nested sequence:

```
{sum(v) : v in [[2, 1], [7, 3, 0], [4]]};
⇒ [3, 10, 4] : [int]
```

In this expression there is parallelism both within each sum, since the sequence function has a parallel implementation, and across the three instances of sum, since the apply-to-each construct is defined such that all instances can run in parallel.

NESL supplies a handful of functions for moving between levels of nesting. These include flatten, which takes a nested sequence and flattens it by one level, for example,

```
flatten([[2, 1], [7, 3, 0], [4]]);
⇒ [2, 1, 7, 3, 0, 4] : [int]
```

and collect that takes a flat sequence of pairs and collects elements with an equal first value together into a nested sequence, for example,

```
collect([(3, 7), (0, 1), (3, 9), (1, 11),
 (0, 6), (3, 5)])
⇒ [(0, [1, 6]), (1, [11]), (3, [7, 9, 5])]
 : [[int, [int]]]
```

As an example of nested parallelism, consider a parallel variation of quicksort (see Fig. 1). When applied to a sequence  $s$ , this version splits the values into three subsets (the elements lesser, equal, and greater than the pivot) and calls itself recursively on the lesser and greater subsets. To execute the two recursive calls, the lesser and greater sequences are concatenated into a nested sequence and qsort is applied over the two elements of the nested sequences in parallel. The final line extracts the two results of the recursive calls and appends them together with the equal elements in the correct order.

The recursive invocation of qsort generates a tree of calls that looks something like the tree shown in Fig. 2. In this diagram, taking advantage of parallelism within each block as well as across the blocks is critical

```
function qsort(a) =
 if (#a < 2) then a
 else
 let pivot = a[#a/2];
 lesser = {e in a| e < pivot};
 equal = {e in a| e == pivot};
 greater = {e in a| e > pivot};
 result = {qsort(v): v in [lesser,greater]}
 in result[0] ++ equal ++ result[1];
```

NESL. Fig. 1 An implementation of quicksort

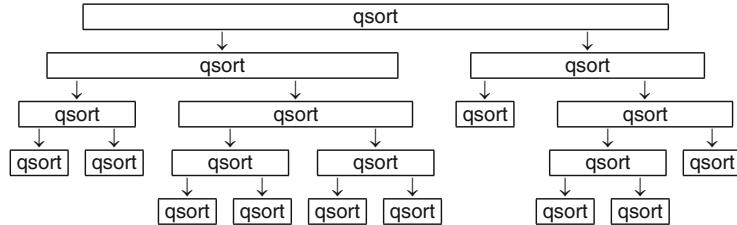
to getting a fast parallel algorithm. If one were to only take advantage of the parallelism within each quicksort to subselect the two sets (the parallelism within each block), the algorithm would do well near the root and badly near the leaves (there are  $n$  leaves which would be processed serially). Conversely, if one were only to take advantage of the parallelism available by running the invocations of quicksort in parallel (the parallelism between blocks but not within a block), the algorithm would do well at the leaves and badly at the root (it would take  $O(n)$  time to process the root). In both cases the parallel “depth” is  $O(n)$  rather than the ideal  $O(\lg n)$  that can be achieved using both forms of parallelism, as described below.

## The Cost Semantics

There are two costs associated with all computations in NESL.

1. **Work:** the total work done by the computation, that is to say, the amount of time that the computation would take if executed on a serial random access machine.
2. **Depth:** this represents the parallel depth of the computation, that is to say, the amount of time the computation would take on a machine with an unbounded number of processors. The depth cost of all the sequence functions supplied by NESL is constant.

The idea of using work and depth for analyzing the asymptotic behavior of parallel algorithms dates back at least to the work on Boolean circuit models, where work (typically referred to as size) is the number of circuit elements, and depth is the longest path in an acyclic circuit [20]. The power of using work and depth was demonstrated by Brent [21] who showed that any circuit of size (work)  $W$  and depth  $D$  can be simulated on a  $P$  processor parallel random access machine [22] (PRAM) in at most  $W/P + D$  steps. One should note that this bound is near optimal since in general the simulation must require at least  $\max\{W/P, D\}$  steps – that is, at least  $D$  steps to follow the chain of dependencies in the circuit, and at least  $W/P$  steps to simulate each of the  $W$  operations. Work and depth costs were also used in the (VRAM) model [23], a strictly flat data-parallel model.



**NESL Fig. 2** The quicksort algorithm. Just using parallelism within each block yields a parallel running time at least as great as the number of blocks ( $O(n)$ ). Just using parallelism from running the blocks in parallel yields a parallel running time at least as great as the largest block ( $O(n)$ ). By using both forms of parallelism, the parallel running time can be reduced to the depth of the tree (expected  $O(\lg n)$  in the Nesl cost model)

NESL introduced the idea of including the costs into the semantics of the language to give a well-defined abstract cost model. The precise definition is given in terms of a profiling operational semantics [3]. The costs are combined using simple combining rules. Expressions are combined in the standard way – for both work and depth, the cost of an expression is the sum of the costs of the arguments plus the cost of the call itself. For example, the cost of the computation:

```
sum(dist(7, n) * #a
```

( $\text{dist}(a, n)$  creates a sequence of  $n$  as) can be calculated as:

|            | Work   | Depth  |
|------------|--------|--------|
| dist       | $n$    | 1      |
| sum        | $n$    | 1      |
| # (length) | 1      | 1      |
| *          | 1      | 1      |
| Total      | $O(n)$ | $O(1)$ |

The apply-to-each construct is combined in the following way. The work is the sum of the work of the instantiations, and the depth is the maximum over the depths of the instantiations. Specifically, let the work required by an expression  $\text{exp}$  applied to some data  $a$  be  $W(\text{exp}(a))$ , and the depth required be

$D(\text{exp}(a))$ , then these combining rules can be written as

$$\begin{aligned} W(\{e1(a) : a \in e2(b)\}) &= W(e2(b)) \\ &\quad + \text{sum}(\{W(e1(a)) : a \in e2(b)\}) \end{aligned} \quad (1)$$

$$\begin{aligned} D(\{e1(a) : a \in e2(b)\}) &= D(e2(b)) \\ &\quad + \text{max\_val}(\{D(e1(a)) : a \in e2(b)\}) \end{aligned} \quad (2)$$

where  $\text{sum}$  and  $\text{max\_val}$  just take the sum and maximum of a sequence, respectively.

As an example, the cost of the computation:

```
{[0:i] : i in [0:n]}
```

( $[0:n]$  creates the sequence  $[0, 1, \dots, n-1]$ ) can be calculated as:

|                       | Work                   | Depth                  |
|-----------------------|------------------------|------------------------|
| $[0:n]$               | $n$                    | 1                      |
| <b>Parallel Calls</b> |                        |                        |
| $[0:i]$               | $\sum_{i=0}^{j < n} i$ | $\max_{i=0}^{j < n} 1$ |
| Total                 | $O(n^2)$               | $O(1)$                 |

Once the work ( $W$ ) and depth ( $D$ ) costs have been calculated in this way, the formula

$$T = O(W/P + D \lg P) \quad (3)$$

places an upper bound on the asymptotic running time of an algorithm on the CRCW PRAM model ( $P$  is the number of processors) [3]. The  $\lg P$  term shows up because of the cost of allocating tasks to processors, and the cost of implementing the  $\text{sum}$  and  $\text{scan}$  operations. On the scan-PRAM [24], where it is assumed that the scan operations are no more expensive than

references to the shared memory (they both require  $O(\lg P)$  on a machine with bounded degree circuits), the equation is:

$$T = O(W/P + D) \quad (4)$$

As an example of how the work and depth costs can be used, consider the `qsort` algorithm described in Fig. 1. To calculate the work one can just use the runtime from the standard sequential analysis (e.g., [25]). Assuming a random selected pivot, this gives  $O(n \log n)$  work with high probability. To calculate the depth one needs to analyze the depth of the recursion tree, but it is not hard to show that with high probability the tree has depth  $O(\log n)$ . All the sequence operations that are used (subselection and appending) are defined to have constant depth. The overall depth is therefore  $O(\log n)$ . This gives a running time on a PRAM of:

$$\begin{aligned} T(n) &= O\left(\frac{n \lg n}{p} + \lg^2 n\right) \quad \text{PRAM} \\ &= O\left(\frac{n \lg n}{p} + \lg n\right) \quad \text{scan-PRAM} \end{aligned}$$

To analyze quicksort by trying to map it onto a  $P$  processor PRAM or other  $P$  processor parallel model is quite complicated. The overall goal of NESL is therefore not only to simplify expressing an algorithm such as quicksort, but also to be able to reasonably easily analyze the asymptotic performance of the algorithm based directly on the code.

## Implementation

The released compiler for NESL translates the language to VCODE [15], a portable intermediate language. The compiler uses a technique called *flattening nested parallelism* [26] to translate NESL into the simpler flat data-parallel model supplied by VCODE. VCODE is a small stack-based language with about 100 functions all of which operate on sequences of atomic values (scalars are implemented as sequences of length 1). A VCODE interpreter was implemented for running VCODE on vector machines such as the Cray C90 and J90, on distributed memory machines based on an MPI back end, and on any serial machine with a C compiler [2]. The interactive NESL environment runs within Common Lisp and can be used to run VCODE on remote machines. This allows the user to run the environment, including the compiler, on a local workstation while executing interactive calls to NESL programs on the remote parallel machines.

Another approach to implementing NESL is to use various scheduling techniques such as work stealing [27] or parallel depth first scheduling [28].

## Bibliographic Notes and Further Reading

The original version of NESL was released in 2002 [29]. It was influenced by SETL [17], CM-Lisp [30], ID [11], and Miranda [18] among other languages. An extensive set of documentation and examples is available online (Web search, NESL). A reasonable set of examples of using NESL for algorithm design and analysis is given in a paper on programming parallel algorithms [4]. The technique of using flattening nested parallelism used in the released implementation is given in [26], and further formalized in [31]. A description of the NESL implementation and experimental results are summarized in [2]. The cost model is described informally in the manual [1] and more formally in a paper on the semantics and provable implementation [3]. Other languages that are similar to NESL in some ways include Cilk [14] (supports nested parallelism and costs based on work and depth – although they refer to them as  $T_0$  and  $T_\infty$ ) and Data-Parallel Haskell [10] (supports nested parallelism in a functional language).

## Bibliography

1. Blelloch GE (1995) NESL: a nested data-parallel language (version 3.1). Technical report CMUCS-95-170, School of Computer Science, Carnegie Mellon University, July 1995
2. Blelloch GE, Chatterjee S, Hardwick JC, Sipelstein J, Zagha M (1994) Implementation of a portable nested data-parallel language. *J Parallel Distrib Comput* 21(1):4–14
3. Blelloch GE, Greiner J (1996) A provable time and space efficient implementation of NESL. In: Proceedings of the ACM SIGPLAN international conference on functional programming, New York, pp 213–225, May 1996
4. Blelloch GE (1996) Programming parallel algorithms. *Commun ACM* 39(3):85–97
5. Milner R, Tofte M, Harper R (1990) The definition of standard ML. MIT Press, Cambridge, Mass
6. Sipelstein J, Blelloch GE (1991) Collection-oriented languages. *Proc IEEE* 79(4):504–523
7. Rose J, Steele GL Jr (1987) C\*: an extended C language for data parallel programming. Technical report PL87-5, Thinking Machines Corporation, April 1987
8. Müller A, Rühl R (1995) Extending high performance fortran for the support of unstructured computations. In: Proceedings of the 9th international conference on supercomputing, ICS '95, pp 127–136, 1995

9. Chamberlain BL, Choi SE, Christopher Lewis E, Lin C, Snyder L, Derrick Weathersby W (2000) Zpl: a machine independent programming language for parallel computers. *IEEE Trans Softw Eng* 26:197–211
10. Jones SP, Leshchinskiy R, Keller G, Chakravarty MMT (2008) Harnessing the multicores: nested data parallelism in Haskell. In: IARCS annual conference on foundations of software technology and theoretical computer science, 2008
11. Arvind, Nikhil RS, Pingali KK (1989) I-structures: data structures for parallel computing. *ACM Trans Program Lang Syst* 11(4):598–632
12. McGraw J, Skedzielewski S, Allan S, Oldehoeft R, Glauert J, Kirkham C, Noyce B, Thomas R (1985) SISAL: streams and iteration in a single assignment language, language reference manual version 1.2. Lawrence Livermore National Laboratory, March 1985
13. Bocchino RL Jr, Adve VS, Adve SV, Snir M (2009) Parallel programming must be deterministic by default. In: Proceedings of the first USENIX conference on hot topics in parallelism, HotPar'09, Berkeley, pp 4–4, March 2009
14. Blumofe RD, Joerg CF, Kuszmaul BC, Leiserson CE, Randall KH, Cilk YZ (1996) An efficient multithreaded runtime system. *J Parallel Distrib Comput* 37(1):55–69
15. Blelloch GE, Chatterjee S, Knabe F, Sipelstein J, Zagha M (1990) VCODE reference manual (version 1.1). Technical report CMU-CS-90-146, School of Computer Science, Carnegie Mellon University, July 1990
16. Chatterjee S (1991) Compiling data-parallel programs for efficient execution on shared-memory multiprocessors. PhD thesis, School of Computer Science, Carnegie Mellon University, October 1991
17. Schwartz JT, Dewar RBK, Dubinsky E, Schonberg E (1986) Programming with sets: an introduction to SETL. Springer-Verlag, New York
18. Turner D (1986) An overview of MIRANDA. *SIGPLAN Notices* 21(12):158–166
19. Hudak P, Wadler P (1990) Report on the functional programming language HASKELL. Technical report, Yale University, April 1990
20. Pippenger N (1979) On simultaneous resource bounds. In: Proceedings of the 20th annual symposium on foundations of computer science, pp 307–311, 1979
21. Brent RP (1974) The parallel evaluation of general arithmetic expressions. *J ACM* 21(2):201–206
22. Fortune S, Wyllie J (1978) Parallelism in random access machines. In: Conference record of the 10th annual ACM symposium on theory of computing, San Diego, CA, pp 114–118, May 1978
23. Blelloch GE (1990) Vector models for data-parallel computing. MIT Press, Cambridge, MA
24. Blelloch GE (1989) Scans as primitive parallel operations. *IEEE Trans Comput* C-38(11):1526–1538
25. Cormen TH, Leiserson CE, Rivest RL, Stein C (2001) Introduction to algorithms, 2nd edn. MIT Press and McGraw-Hill, Cambridge/New York
26. Blelloch GE, Sabot GW (1990) Compiling collection-oriented languages onto massively parallel computers. *J Parallel Distrib Comput* 8(2):119–134
27. Blumofe RD, Leiserson CE (1999) Scheduling multithreaded computations by work stealing. *J ACM* 46(5):720–748
28. Blelloch G, Gibbons P, Matias Y (1999) Provably efficient scheduling for languages with fine-grained parallelism. *J ACM* 46(2):281–321
29. Blelloch GE (1992) NESL: a nested data-parallel language. Technical report CMU-CS-92-103, School of Computer Science, Carnegie Mellon University, January 1992
30. Steele GL Jr, Daniel Hillis W (1986) Connection machine LISP: fine-grained parallel symbolic processing. In: LISP and functional programming, Cambridge, pp 279–297, 1986
31. Keller G, Simons M (1996) A calculational approach to attaining nested data parallelism in functional languages. In: Proceedings of the 2nd asian computing science conference on concurrency and parallelism, programming, networking, and security, Springer-Verlag, pp 234–243, 1996

## Nested Loops Scheduling

► [Parallelism Detection in Nested Loops, Optimal](#)

## Nested Spheres of Control

► [Transactions, Nested](#)

## NetCDF I/O Library, Parallel

ROBERT LATHAM

Argonne National Laboratory, Argonne, IL, USA

### Synonyms

[High-level I/O library](#); [Pnetcdf](#)

### Definition

The serial netCDF library has long provided scientists with a portable, self-describing file format and a straightforward programming interface based on multidimensional arrays of typed variables (more detail in the History section). Parallel-NetCDF provides an API for parallel access to traditionally formatted netCDF files. Parallel-NetCDF both produces and consumes

files compatible with serial netCDF, while providing a programming interface similar, though not identical, to netCDF. Parallel-netCDF API modifications provide a more appropriate interface when expressing parallel I/O.

## Discussion

Parallel-NetCDF is one of a family of libraries built to meet the needs of computational scientists, as opposed to lower-level I/O libraries that deal more with the details of underlying storage. Before going into more detail about Parallel-NetCDF, let us see how it fits in the bigger picture of scientific computing and application I/O.

### Parallel-NetCDF and the I/O Software Stack

Today's leading computing platforms contain hundreds of thousands of processors. In order to efficiently utilize these computers, applications use simplifying abstractions provided by tools and libraries. MPI libraries provide a standard programming interface for parallel computation on a wide array of hardware. Math libraries such as BLAS hide the details of CPU-specific optimizations. These lower-level math and communication libraries contribute to an overall software stack built to allow developers to focus on the science behind their applications.

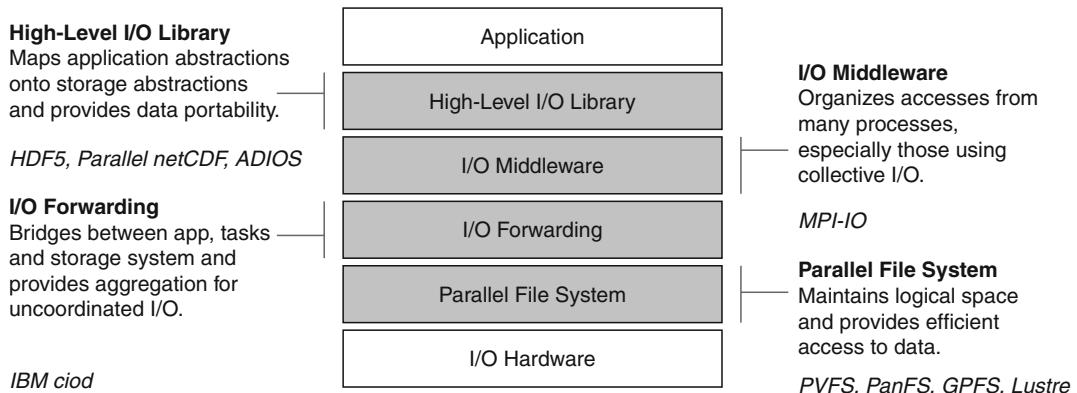
I/O performance on high-end computing platforms follows a similar story. Performance comes from aggregating many storage devices. In fact, CPU performance has consistently improved at a rate much faster than that of storage devices. This discrepancy makes the need for parallel I/O across multiple devices even more acute. Yet as the number of devices involved in I/O grows, coordinating I/O across those devices becomes more of a challenge. In the same way that communication and math libraries assist the parallel computation side of scientific simulations, an "I/O software stack" assists data management.

The large number of storage devices mentioned above forms the foundation of the software stack. Several additional layers provide abstractions to help make the task of extracting maximum performance accessible to applications programmers. The programming API and data model that Parallel-NetCDF presents to the scientific programmer are conceptually closer to the tasks common to scientific applications

and hide many lower-level details. A discussion of the lower levels of the software stack, though hidden by Parallel-NetCDF, will provide a better appreciation of Parallel-NetCDF's role at the upper level.

- Storage systems contain thousands of individual devices to increase the potential bandwidth. The *parallel file system* manages those separate devices and collects them into a single logical unit. The file system does lack some important features: typically file systems have no mechanism for coordinated I/O, and the data model is still fairly low level for computational scientists to use directly.
- The *I/O forwarding* layer typically exists only on the largest computer systems. Applications do not interact with this layer directly. Rather, this layer simplifies the scalability challenge: a large number of compute nodes communicate with a smaller number of I/O forwarding nodes, and these nodes in turn talk to the file system.
- The *middleware*, or *MPI-IO* [6], layer introduces more sophisticated algorithms tailored to parallel computing. This layer coordinates I/O among a group of processes, introducing collective I/O to the stack. MPI datatypes allow applications to describe arbitrary I/O access patterns. Applications can use the MPI-IO layer directly, but the programming model is still rather low level and not a perfect fit to the needs of common applications. MPI-IO libraries, however, provide an excellent foundation for higher-level libraries such as Parallel-NetCDF.

Parallel-NetCDF sits atop these lower-level abstractions; it belongs to the *high-level I/O library* layer. Libraries in this layer introduce concepts such as multidimensional arrays of typed data, which are better suited to scientific applications. For example, a climate code can represent temperature in the atmosphere with a three-dimensional array – latitude, longitude, and altitude – containing the number of degrees Celsius. In addition to array storage, applications can also provide descriptive information in the form of attributes on variables, dimensions, or the dataset itself. These libraries also define the layout of data on disk. These annotated, self-describing file formats make exchanging data with colleagues now and in the future much easier.



**NetCDF I/O Library, Parallel.** Fig. 1 The I/O software stack. Parallel-NetCDF sits at the top, allowing applications to express data manipulations more abstractly while lower levels deal with coordination among multiple processes and maximize performance of storage devices

## History

The netCDF library has long provided a straightforward programming API and file format for serial applications. In the summer of 2002, Argonne National Laboratory and Northwestern University started a joint effort to build a parallel I/O library while keeping the best parts of netCDF.

Parallel programs had been using netCDF for some time, though the serial library forced programmers to use serial approaches when writing out netCDF datasets. In one approach, each process writes out its own data set. This “N-to-N” model can put significant strains on both the file system and any postprocessing tools, especially since large systems today can have hundreds of thousands of processes. Another approach sends all data to a master I/O process, and that process in turn writes out the dataset. This “send-to-master” technique imposes an obvious bottleneck because all data must be funneled through a single process.

Without changing the existing netCDF file format, Parallel-NetCDF introduced a new programming API. This parallel-oriented API is not identical to serial netCDF, but is comfortable to anyone familiar with the existing serial netCDF API. The Parallel-NetCDF API allows computational scientists to write out datasets in parallel: all processes can participate in an “N-to-1” operation, producing a single dataset while leveraging a host of parallel I/O optimizations. The Parallel-NetCDF API and File Format section contains examples and more discussion of the Parallel-NetCDF API.

Parallel-NetCDF has become an important tool for high-performance I/O in the climate and weather domains. These domains have long-established netCDF-based workflows, using netCDF datasets for archiving, analysis, and data exchange. Since Parallel-NetCDF retains the same file format as netCDF, researchers in climate and weather can make changes to their simulation codes while leaving other components of their workflow the same. Parallel-NetCDF is useful in other domains as well: the programming API and file format make it possible to deliver good parallel I/O performance without having to master all of MPI-IO.

N

## The Parallel-NetCDF API and File Format

Discussions about Parallel-NetCDF need to cover two main points: the programming model and the on-disk data format. The API simplifies many MPI-IO concepts while retaining several key optimizations. The on-disk format affects I/O performance, but it also matters when exchanging data with collaborations and colleagues.

If applications were to use MPI-IO directly, the “basic type” they would build on would be a linear stream of bytes. In contrast, Parallel-NetCDF offers a more application-friendly model based on multidimensional arrays of typed data. Operations on these arrays could be as simple as reading the entire array, or more complicated such as having each process write a subcube representing a chunk of the Earth’s atmosphere. In addition to operations for manipulating data, the API contains routines to annotate the data in the file or the

file itself. Such annotations may include timestamps, machine information, experiment or workflow information, or other provenance information to better understand how an application produced the data in this file. The Standard Interface section goes into more detail about the programming interface as well as how Parallel-NetCDF interacts with the underlying MPI-IO library.

In addition to the programming interface, the file format plays a major role in how useful Parallel-NetCDF can be to application groups. Parallel-NetCDF applications create a self-describing, portable file with support for embedded metadata. API routines allow programs to query the contents of data files without any prior knowledge of the contents. The structure of the datasets extends to the representation of bytes on disk: even if a dataset is moved to a different machine, the library will still be able to understand that file. The IBM Blue Gene/P system, for example, has a 32-bit, big-endian architecture, while the Cray XT5 has a 64-bit little-endian architecture. Despite these differences, files created on one can be read on the other.

[Figure 2](#) depicts a simple Parallel-NetCDF application. A climate code wants to create a checkpoint file to store some key pieces of information. Atmospheric temperature is stored as a three-dimensional array of double-precision floating point values. Barometric pressure at the Earth's surface requires only two

dimensions and only single-precision floating points. Further, the application wants to ensure that future consumers of this dataset know which units these variables use. It therefore stores this information as an attribute on each variable. On disk, the dataset contains a small header describing the size and type of both variables and any attributes, followed by the actual data for the variables. This fairly simple file format comes with some restrictions, but the trade-off results in efficient file accesses for parallel I/O.

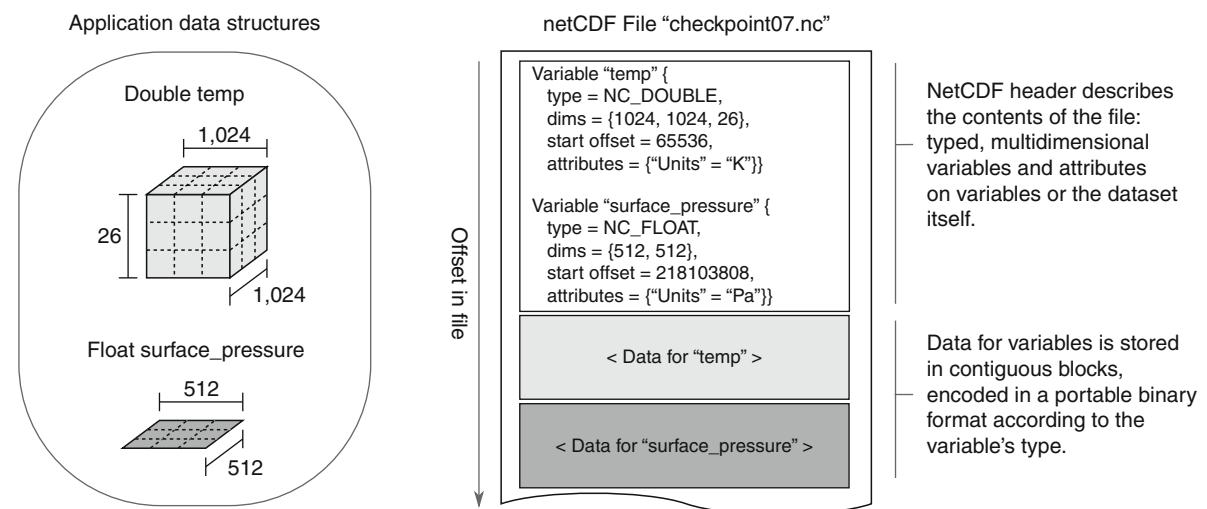
Parallel-NetCDF uses the same file format as netCDF, making it easier for application groups to adopt Parallel-NetCDF. A programmer can change the computationally intensive simulation code to use Parallel-NetCDF while the other components of the workflow (e.g., visualization, analysis, archiving) can continue to use serial netCDF.

### Annotated Examples

Parallel-NetCDF shares many netCDF concepts but uses a slightly different API. The following simple example programs should make the differences clear.

### Standard Interface

[Figure 3](#) contains a full, correct Parallel-NetCDF program to create a simple dataset and write data into that dataset in parallel. This program, though brief, demonstrates many key features of the library.



**NetCDF I/O Library, Parallel. Fig. 2** Left: how an application views netCDF objects. Right: how netCDF and Parallel-NetCDF store those objects on disk

```

1 #include <mpi.h>
2 #include <pnetcdf.h>
3
4 int main(int argc, char **argv) {
5 int ncid, nprocs, rank, dimid, varid1, varid2, ndims=1;
6 MPI_Offset start, count=1;
7 char buf[13] = "Hello World/n";
8
9 MPI_Init(&argc, &argv);
10 ncmpi_create(MPI_COMM_WORLD, "demo.nc",
11 NC_WRITE|NC_64BIT_OFFSET, MPI_INFO_NULL, &ncid);
12
13 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14 MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
15
16 ncmpi_def_dim(ncid, "d1", nprocs, &dimid);
17 ncmpi_def_var(ncid, "v1", NC_INT, ndims, &dimid, &varid1);
18 ncmpi_def_var(ncid, "v2", NC_INT, ndims, &dimid, &varid2);
19 ncmpi_put_att_text(ncid, NC_GLOBAL, "string", 13, buf);
20
21 ncmpi_enddef(ncid);
22
23 start = rank;
24 ncmpi_put_vara_int_all(ncid, varid1, &start, &count, &rank);
25 ncmpi_put_vara_int_all(ncid, varid2, &start, &count, &rank);
26
27 ncmpi_close(ncid);
28 MPI_Finalize();
29 return 0;
30 }

```

**NetCDF I/O Library, Parallel.** Fig. 3 Basic Parallel-NetCDF program using the standard interface. Each process collectively writes its MPI rank into two variables. For brevity, error checking/handling has been omitted

Because Parallel-NetCDF relies on MPI-IO, the program must include the mpi.h header file (line 1) and initialize (line 9) and finalize (line 28) the MPI library.

The ncmpi\_create routine (line 10) adds two arguments to netCDF's creation function: an MPI communicator and an MPI Info object. These two arguments do expose a bit of the underlying MPI library, but they give the programmer some flexibility and the ability to tune operations if needed. In this example, the communicator is just MPI\_COMM\_WORLD, or all processes in this MPI program. In some situations, though, the application may wish to have only a subset of processes participate in a dataset operation. The Info object in this example is empty, but the Tuning Parallel-NetCDF section will cover some situations where the object is used.

Parallel-NetCDF, like netCDF, has a *bimodal* interface: when creating a dataset, an application starts in *define mode*, where it must first describe what variables it will write, the size and type of those variables, and any attributes. This example defines a single dimension (line 16) and assigns that dimension to two variables (lines 17 and 18).

Datasets can contain a great deal of metadata. Both variables and dimensions have human-readable labels. In addition to those labels, attributes may be defined and placed on dimensions, variables, or the dataset itself. Line 19 places an attribute on the entire dataset.

In the example, line 21 calls ncmpi\_enddef to switch from define mode to *data mode*. By asking the programmer to predefine variables and attributes, the library can allocate space with little overhead. Reentering define mode can potentially trigger an expensive rewriting of the dataset. For a large class of problems, however, one knows ahead of time which information will be written out.

Lines 24-25 do the actual work of storing information into the dataset. Parallel-NetCDF provides one set of functions for uncoordinated *independent I/O* and another parallel set for coordinated *collective I/O*. This example uses the collective version (as shown by the \_all suffix). Every MPI process participates in these writes, each writing out its MPI rank. In a traditional serial netCDF program, each rank would send data to a master process, and this process would in turn write out the information. This “send-to-master” model

quickly becomes untenable, however, as the number of MPI processes increases and as the amount of memory available to each MPI process gets smaller.

The collective Parallel-NetCDF routines in turn call collective MPI-IO routines. MPI-IO collective routines can use powerful optimizations such as two-phase I/O [4] or data shipping [3]. It is possible to use independent I/O with Parallel-NetCDF, but the opportunities for optimization are greatly limited in that case.

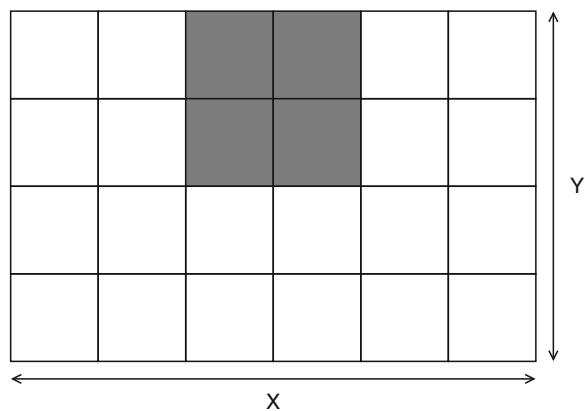
Once executed, this example will produce a file “demo.nc.” This dataset will contain two variables, as can be verified with either netCDF or Parallel-NetCDF utilities.

The Parallel-NetCDF standard interface shares much in common with the serial netCDF interface. The function name explicitly states whether the

operation is a read or write (`_put` or `_get`), the type of access (e.g., `_var`, `_vara`), and the type (e.g., `_int`, `_float`, `_double`). For example, from the name alone a programmer knows that the function `ncmpi_put_vara_int` writes integers into a subarray.

#### A more realistic example

This simple example, operating on a one-dimensional array, may not give the most insight into all the work going on behind the scenes. Figure 4 depicts a two-dimensional array of which this example program wants to only write a smaller subarray. Perhaps this array represents one frame of a movie and a parallel program renders portions of each frame.



**NetCDF I/O Library, Parallel. Fig. 4** Writing a subarray of a two-dimensional array

```

1 ...
2 #define NDIMS 2
3 int dims[NDIMS], varid1, ndims=NDIMS;
4 MPI_Offset start[NDIMS], count[NDIMS];
5
6 ncmpi_def_dim(ncfile, "y", 4, &(dims[0]));
7 ncmpi_def_dim(ncfile, "x", 6, &(dims[1]));
8
9 ncmpi_def_var(ncfile, "frame", NC_DOUBLE, ndims, dims, &varid1);
10
11 ncmpi_enddef(ncfile);
12
13 start[0] = 0; start[1] = 4;
14 count[0] = 2; count[1] = 2;
15 ncmpi_put_vara_double_all(ncfile, varid1, start, count, data);
16 ...

```

**NetCDF I/O Library, Parallel. Fig. 5** Writing the selected portion of the above array

Figure 5 contains the Parallel-NetCDF code needed to write the indicated region. First, the program must

describe the entire variable: lines 6, 7, and 9 describe the overall shape of the variable and the type of data it will contain. Lines 13 and 14 set up the shape of the desired selection or subregion of the variable and pass these arguments as parameters to the call at line 15.

These few lines of Parallel-NetCDF code trigger a lot of activity behind the scenes. Even though the desired region is logically contiguous, when the array is stored on disk, this selection will end up scattered across the file in a noncontiguous access pattern. Fortunately for Parallel-NetCDF, MPI-IO makes it easy to describe these accesses with a file view.

Storage systems yield the best performance with a few large, contiguous accesses. The MPI-IO library will

apply several optimizations to transform a noncontiguous access into one more likely to give good bandwidth. The Parallel-NetCDF call in this example is collective, which means that the lower MPI-IO layer can in turn use collective I/O optimizations: the MPI-IO library can communicate with the other processes participating in this I/O call and rearrange the requests. To the file system, this subarray access will end up looking like a more performance-friendly contiguous request.

While this discussion glosses over many of the details, the important point is that a few Parallel-NetCDF functions convey a great deal of information to the lower levels of the I/O stack. Applications using Parallel-NetCDF thus get the benefits of a self-describing portable file format and a relatively straightforward API while still achieving performance without requiring the application programmers to master MPI-IO.

### The Flexible Interface

Parallel-NetCDF provides an additional set of routines to allow for even more freedom in describing an application's data model. In Fig. 6, a write operation from Fig. 3 is converted to the Flexible Data Mode interface. This interface allows an application to use MPI datatypes to describe how information is laid out in memory.

The datatype in this example is simple (MPI\_INT), but a more complicated MPI datatype might, for example, write out all the data in a multidimensional array while skipping over the "ghost cells." Ghost cells, copies of data belonging to other MPI processes, are used to optimize communication in nearest-neighbor simulations and do not actually need to be written to disk. In many cases, the Flexible Data Mode interface permits

an application to avoid an additional buffer copy before making a Parallel-NetCDF call.

### Nonblocking Interface

Parallel-NetCDF further extends the traditional netCDF API through the introduction of nonblocking I/O routines. Programmers familiar with MPI nonblocking communication routines will note strong similarities between MPI nonblocking routines and those in Parallel-NetCDF. A program posts one or more operations and then waits for completion of those operations.

In Fig. 7, the example from Fig. 3 has been modified yet again, this time to use Parallel-NetCDF nonblocking operations. Throughout the I/O software stack, more information about I/O activity results in more opportunities for optimization. In this example, even though the calls operate on separate variables, those variables are still in the same dataset and appear to the MPI-IO layer as parts of a single file. Parallel-NetCDF can take these nonblocking requests and optimize them into a single, larger I/O operation [1].

Parallel-NetCDF can optimize these nonblocking requests in this way because the library makes no guarantees about when work will occur (sometimes called "make progress" in the MPI context). It may appear to the program that I/O work happens in the background when calling these nonblocking routines. Actually, when the operation is posted (lines 4 and 6), no work happens. Instead, the library defers all work until the application makes an additional "wait for completion" function call (line 9). Once Parallel-NetCDF has a list of all the outstanding operations, it can then construct a single I/O request encompassing all operations. Typically, storage systems perform better with larger

```

1 /* function prototype */
2 int ncmpi_put_vara_all(int ncid, int varid,
3 /* start and count describe access in file */
4 const MPI_Offset start[], const MPI_Offset count[],
5 /* 'buf' 'bufcount' and 'datatype'
6 /* describe data in memory */
7 const void *buf, MPI_Offset bufcount, MPI_Datatype datatype);
8
9 ...
10 start = rank;
11 ncmpi_put_vara_all(ncfile, varid1, &start, &count,
12 &rank, count, MPI_INT);
13 ...

```

NetCDF I/O Library, Parallel. Fig. 6 Prototype for and usage of one of the Flexible Data Mode routines

```

1 int requests[2], statuses[2], data[2];
2 ...
3 data[0] = rank + 1000;
4 data[1] = rank + 10000;
5 ncmpi_iwrite_vara_int(ncfile, varid1, &start, &count,
6 &(data[0]), &(requests[0]));
7 ncmpi_iwrite_vara_int(ncfile, varid2, &start, &count,
8 &(data[1]), &(requests[1]));
9 ncmpi_wait_all(ncfile, 2, requests, statuses);

```

**NetCDF I/O Library, Parallel. Fig. 7** The “implicit” nonblocking interface: no progress occurs in the background, but operations can be batched together when completed

I/O requests, so coalescing operations in this way can yield good performance improvements.

## Tuning Parallel-NetCDF

Extracting good performance out of Parallel-NetCDF comes down largely on following a handful of best practices: perform collective I/O to a small number of files. Doing so provides the most information to the lower I/O software stack layers, and allows for those layers to make as many optimizations as possible. For even more fine-tuning, the library does provide some additional tuning mechanisms for applications. For the most part, these tuning knobs help tweak parameters for layers farther down the software stack. The MPI Info parameter passed to the create and open calls can direct the optimizations the underlying MPI-IO library chooses to make. For example, on Lustre file systems file locks are exceedingly expensive. By setting the MPI Info hint `romio_ds_write` to “`disable`”, the ROMIO implementation of the MPI-IO library, the most common implementation on systems with Lustre installed, will avoid using locks. Knowing more about the characteristics of the storage system and the application workload will make the selection of appropriate hints easier and more productive. The available MPI-IO tuning parameters vary based on the MPI-IO implementation and file system used. Users should consult their MPI-IO library’s documentation.

Often, the most effective way for a computational scientist to improve the I/O performance of a program is to enlist the aid of the Parallel-NetCDF community (see the Bibliographic Notes and Further Reading section). If the scientist can remove the simulation and science aspects of the program, leaving only the representative data structures, the resulting “I/O kernel” becomes a

valuable resource for exploring all tuning options available. I/O kernels leave no doubt as to what the scientist requires from the Parallel-NetCDF library and storage system. Because these small programs have few if any dependencies on additional libraries, they can become part of the library’s correctness and performance tests, ensuring that modifications and improvements made to Parallel-NetCDF also benefit applications.

A discussion about tuning Parallel-NetCDF would not be complete without a few words concerning record variables. In Parallel-NetCDF and netCDF, variables can have an unlimited dimension. Often, variables that change over time use this feature: each iteration of the simulation can append data along the unlimited “time” dimension. When more than one variable contains an unlimited dimension, those variables are stored on disk in an interleaved fashion. The writing and reading of data interleaved in this way will often yield poor performance. Programmers must weigh the flexibility of record variable storage against the cost of performance. With an I/O kernel, Parallel-NetCDF experts can likely find a set of tuning parameters to mitigate some of the performance loss, and the project will be looking at more sophisticated ways of dealing with record variables in the future.

## Conclusion

The Parallel-NetCDF library provides a more natural programming interface for high-performance storage systems. By implementing parallel I/O concepts in terms of multidimensional arrays, many computational science simulations are able to naturally express their data structures.

Parallel-NetCDF provides a further benefit in that it encapsulates a great deal of I/O expertise. Computer scientists working on high-performance storage may not know a great deal about weather, climate, combustion,

or other scientific domains but know much about how to get the best performance out of storage devices, the file system, and MPI-IO. Application scientists in turn specialize in modeling phenomena, developing numerical methods, and exploring other topics more relevant to computational science. Parallel-NetCDF and other high-level I/O libraries provide a middle ground where scientists and storage experts can come together to maximize productivity.

## Related Work

The HDF5 library [5] was the first high-level I/O library to be built on top of MPI-IO. HDF5 provides a large and flexible API. For example, HDF5 applications do not need to enter a define mode to describe the variables and dimensions. HDF5 writes all metadata for the file on an as-needed basis. The trade-off for this flexibility is some additional overhead if the application is creating many new variables or other metadata-intensive workloads. HDF5 has the further benefit of allowing variables to grow without bound in any dimension. The HDF5 library allocates space for variables on demand. Consider the storage of a sparse matrix: only those values actually present in the matrix would be stored on disk. When multiple processes are updating variables in this way, however, this on-demand space allocation can result in some performance and consistency challenges. HDF5 currently forces all parallel metadata updates through rank 0, though the group is working on more sophisticated techniques.

More recently, the serial netCDF developers released netCDF-4 [7], a library with the netCDF programming API on top of the HDF5 file format. This project brought several HDF5 features to netCDF, including parallel I/O support, support for variables with multiple unlimited dimensions, and support for much larger variables. While introducing a new file format, it still can operate on older netCDF datasets. The netCDF-4 API does not have some of the more sophisticated Parallel-NetCDF features like the Flexible Mode interface or the Non-blocking interface.

## Related Entries

- [Benchmarks](#)
- [Distributed-Memory Multiprocessor](#)
- [HDF5](#)
- [MPI \(Message Passing Interface\)](#)

- [I/O](#)
- [MPI-IO](#)

## Bibliographic Notes and Further Reading

For more technical insight into Parallel-NetCDF, Jianwei Li et al.'s SC 2003 paper [2] presents the design choices made in developing the library; experimental results are also presented. The multivariable I/O optimizations are covered further in [1].

The Parallel-NetCDF website is [www.mcs.anl.gov/parallel-netcdf](http://www.mcs.anl.gov/parallel-netcdf). The site hosts pointers to additional Parallel-NetCDF papers and projects, as well as links to production releases and the latest code.

The Parallel-NetCDF community of users and developers communicates on the [parallel-netcdf@mcs.anl.gov](mailto:parallel-netcdf@mcs.anl.gov) mailing list. The quality of discourse on the list has been high from the beginning. Any topic related to Parallel-NetCDF is open for discussion.

Parallel-NetCDF developers regularly give tutorials and workshops covering the library.

## Acknowledgment

This work was supported in part by the U.S. Dept. of Energy under Contract DE-AC02-06CH11357.

## Bibliography

1. Gao K, Liao WK, Choudhary A, Ross R, Latham R (2009) Combining i/o operations for multiple array variables in parallel netcdf. In: Proceedings of the workshop on interfaces and architectures for scientific data storage, held in conjunction with the IEEE Cluster Conference, New Orleans, Louisiana, September 2009
2. Li J, Liao WK, Choudhary A, Ross R, Thakur R, Gropp W, Latham R, Siegel A, Gallagher B, Zingale M (2003) Parallel netCDF: a high-performance scientific I/O interface. In: Proceedings of SC2003: high performance networking and computing, Phoenix, AZ, November 2003. IEEE Computer Society Press, Los Alamitos
3. Prost JP, Treumann R, Hedges R, Jia B, Koniges A (2001) MPI-IO/GPFS, an optimized implementation of MPI-IO on top of GPFS. In: Proceedings of SC2001, November 2001
4. Thakur R, Gropp W, Lusk E (1999) Data sieving and collective I/O in ROMIO. In: Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation, IEEE Computer Society Press, pp 182–189, February 1999
5. The HDF Group (2008) HDF5. <http://www.hdfgroup.org>
6. The MPI Forum. MPI-2: extensions to the message-passing interface, July 1997. <http://www mpi-forum.org/docs/docs.html>
7. Unidata. netCDF4. <http://www.unidata.ucar.edu/software/netcdf/index.html>

## Network Adapter

### ► Network Interfaces

## Network Architecture

- Collective Communication, Network Support For
- Cray XT4 and Seastar 3-D Torus Interconnect
- Ethernet
- InfiniBand
- Myrinet
- Networks, Direct
- Quadrics

## Network Interfaces

HOLGER FRÖNING

University of Heidelberg, Heidelberg, Germany

### Synonyms

NI (Network Interface); NIC (Network Interface Controller or Network Interface Card); Network adapter

### Definition

A network interface – or network interface controller, network interface card, network adapter – is a hardware component that is designed to allow computers to access an interconnection network for communication and synchronization purposes. Therefore, a network interface typically provides two different kinds of interfaces, one toward the computer (host) side and one toward the network side. The network interface translates the protocol of the host interface to the network protocol and vice versa, and translates between the different physical media. From the network's point of view, network interfaces are the end points of the network, as here the network packets are injected into, respectively retrieved from the network. As all end points of a network must be uniquely addressable, each network interface is assigned a unique number. The major task of a network interface is to insert packets into the network on the sending node, and to receive them from the network on the target node. The accomplishment

of this task, however, is highly dependant on the network interface architecture. In particular, the network interface architecture defines which subtasks are to be performed by software layers and which are performed by hardware modules.

## Discussion

### Introduction

The main task of a network interface is to provide a computer access to an interconnection network in a performant, transparent, and safe manner [1]. A well-designed network interface is performant by providing a high bandwidth and low latency. Similarly, it is transparent because it only adds minimal overhead to the performance properties of the network, which are again bandwidth and latency. A safe network interface should prevent erroneous processes from disabling the network or reducing the network's performance. Typically, a well-designed network interface is unobtrusive. Opposed to this, a badly designed network interface will turn into a bottleneck, preventing a client from leveraging the network's full performance.

Considering the Open System Interconnection Reference Model (OSI Reference Model) [2], which divides the task of communication into seven layers, the main task of a basic network interface is to handle the lowest two layers. These are the Physical Layer (PHY) and Data Link Layer (DDL). In the Data Link Layer in particular the Media Access Control (MAC) is handled by the network interface. However, more sophisticated network interfaces can also inherit task from upper layers. Then CPU off-loading takes place, and tasks are handled by the network interface instead of a software instance running on a CPU.

Implementing only the PHY and MAC layer on a network interface is typically true for indirect networks [3]. Here, central switches are used to connect an arbitrary number of network nodes, and routing only takes place in these switches. In a direct network the switching resources are distributed over the network nodes; thus, each network interface additionally includes the switching and routing modules. The network degree defines the number of links per node, and packets are routed among these links (forwarding of packets) and the host interface (injection and retrieval of packets). Therefore, a network interface for a direct network implements beside the PHY and MAC layer

also the Network Layer, which is responsible for switching and routing. Considering for instance a case in which the network (and the network interface) also provides Quality of Service (QoS), the network interface has also to implement parts of the Transport Layer.

## Basic Architecture

Network interfaces can be classified on how messages (or more generally: packets) are injected and retrieved from the network – which is the basic and unique task of all network interfaces. The most basic approach uses one register located on the network device, the so-called register-based interface. A process running on a Central Processing Unit (CPU) generates the message by consecutively writing into this registers: as messages are usually separated into header, payload, and tail part, these three parts are written word after word into the device register. The network interface collects the data written and assembles a message out of this information. The message is forwarded over the network to the destination, where a process can receive the message by reading it word after word from a device register. This approach requires a fixed message format in terms of header, payload, and tail size, which restricts the use of it. A solution to overcome this is to use a set of registers instead, where the different registers serve for different message parts.

In the register-based interface, the CPU (i.e., a process running on the CPU) is writing data into a register, respectively, reading data from a register, for each word of the message. This Programmed I/O (PIO) is a viable solution for small size messages [4]. For larger messages the CPU is occupied for a long time by copying data to and from the network. A solution to off-load this task from the CPU to the network interface is Direct Memory Access (DMA). Here, the CPU describes the message to be sent – including header and tail information and location of the data payload in the main memory – in a descriptor. Only this descriptor is transferred by the CPU to the network interface, typically using PIO. Using the information contained in this descriptor, the network interface is able to independently send or receive a message. In the case of sending it accesses the main memory to fetch the data payload and assembles a message. In the case of receiving it writes the data payload to the main memory location

specified in the receive descriptor. As such a descriptor-based network interface is working asynchronously, the CPU has to be explicitly informed if a task has been completed.

The descriptor-based approach implicates a higher overhead compared to the register-based approach: the descriptor must be constructed, it must be handed over to the network interface, and – as an indirection stage – the network interface must access the main memory to fetch or store the payload. Because for small payloads this overhead is a large fraction of the communication time, the descriptor-based approach is typically used for large payloads. In this case, the overhead is small compared to the complete communication time. Additionally, communication and computation can overlap because the CPU is not actively handling messages.

## Notifications

Notifications are an important part of the communication architecture, as they trigger certain operations in the CPU or network interface. Basically, for every action the CPU has to inform the network interface and vice versa. This is typically done using explicit notifications. An exception are device registers, because accessing these on-device structures can result in side effects which equal an explicit notification.

Basic notifications are possible using device registers, where the CPU can read out the current state of the network interface, for example, if a message in a register set has been consumed or if a new message has been received and is available in a register set. Similarly, the network interface uses side effects to get notified of new requests in the register sets.

However, side effects cannot be applied to the CPU because here no side effects exist. The usual solution to notify a CPU is either an interrupt that is asserted by the network interface, or to ensure that a CPU is periodically checking certain memory or register locations for changes (polling). While the polling method offers the lowest latency possible it consumes a lot of CPU time. On the other hand, interrupts do not consume CPU time when no event is present, but the latency of the notification is much higher due to the interrupt handling which has to be done by the Operating System (OS).

## Queues

While the basic architecture presented above is sufficient in principle, it is desirable to allow more outstanding send or receive requests for an improved decoupling between CPU and network interface. The register (set) – either being used for the message itself or for a descriptor – allows only one outstanding request: the CPU writes the information into the register set, and then waits until this data has been consumed by the network interface. Only then the register (set) can be used for the next request.

The solution for this problem are queues, which are used as interface between CPU and network interface. They provide a decoupling between CPU and network interface. By storing either a message or a descriptor as an entry in a queue, as many requests as queue entries exist can be outstanding. For sending a packet, the CPU inserts a new entry in the appropriate queue and the network interface consumes it. For receiving, the queue is used vice versa. Queues can be located in an arbitrary space – for instance main memory, on-chip memory of the network interface, or DRAM on the network interface card. For the send queue, it has to be ensured that the network interface is informed about new entries in the queues. This is in particular true for queues located in main memory, while for on-device queues side effects can eventually be leveraged. For the receive queue the CPU has to be notified about new entries. The only exception is a polling approach with guaranteed change of data in a queue entry.

Queues are applicable for all interactions between CPU and network interface, including notifications. Several approaches do exist where these queues are located. So-called memory-less network interfaces use queues in main memory, thus they do not require on-card memory. The reason is that main memory is one of the cheapest memories available, whereas on-card or on-chip memory is more expensive. In addition, main memory is also more scalable.

## Advanced Architecture

The network interface is typically a single instance located in between highly parallelized environments. On the host side the degree of parallelism is increasing due to the multi-core trend [5]. More and more CPU cores are integrated into CPU sockets and – in the worst case – all of them want to communicate using

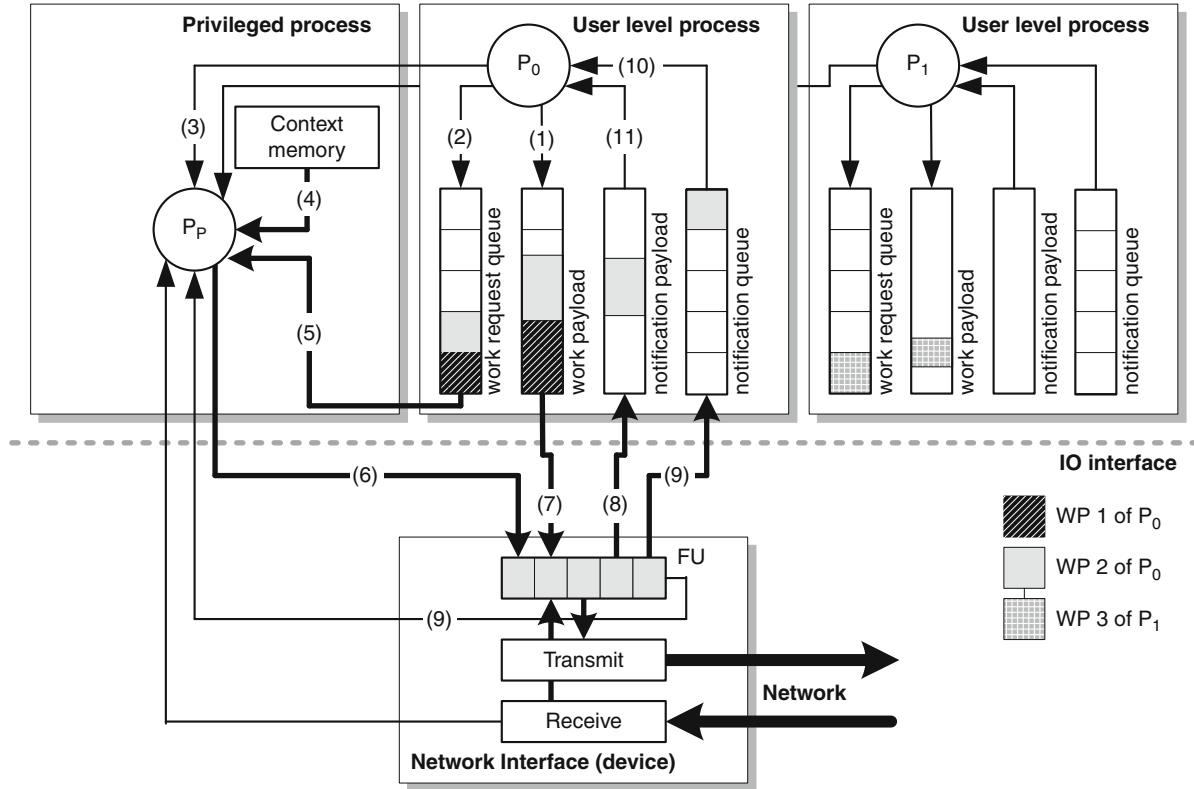
the network interface. The resurgence of interest in Virtual Machine (VM) environments [6, 7] is increasing the number of running processes potentially accessing the network. On the network side, a large number of remote network nodes are potentially accessing local resources over the local network interface. Only a network interface that adds only minimal overhead allows a performant network connection and does not turn into a bottleneck.

Many network interfaces rely on the OS to check requests and multiplex the network interface between competing processes. Only if requests are not erroneous they are forwarded to the network interface where they can be processed safely. Such network interfaces are typically of very low complexity. However, the involvement of the OS requires system calls (or inter process communication, IPC), which results in an increase of overhead, in particular CPU time and latency.

An example work flow for a multiplexed access is shown in Fig. 1. For each process, work packets and notifications are held in exclusive data structures located in main memory. A privileged process supervises the network access and multiplexes the network device among the competing user-level processes. For this purpose, each user process notifies the privileged process and the network device updates its status to the privileged process. If the network device is unoccupied, the privileged process can schedule a new work request from one of the user processes or a receive request to the network device. For a Virtual Machine environment, the privileged process is typically part of the Hypervisor, as a guest OS does not have access to hardware.

**User-level communication** [8] avoids the involvement of the OS (OS bypass) and allows user level processes to directly communicate with the device. In this case, however, it is up to the network interface to check requests for errors. Only then a safe operation is guaranteed and erroneous packets cannot disable the network or reduce its performance. Because the OS is bypassed, no multiplexing between processes can take place. This multiplexing must now be performed by the device, either in a time sliced manner, using module replication [9] or device virtualization.

**Device virtualization** [10] is an approach that virtualizes the network interface in way, that an arbitrary number of processes can use the network interface directly and without involvement of the OS. This



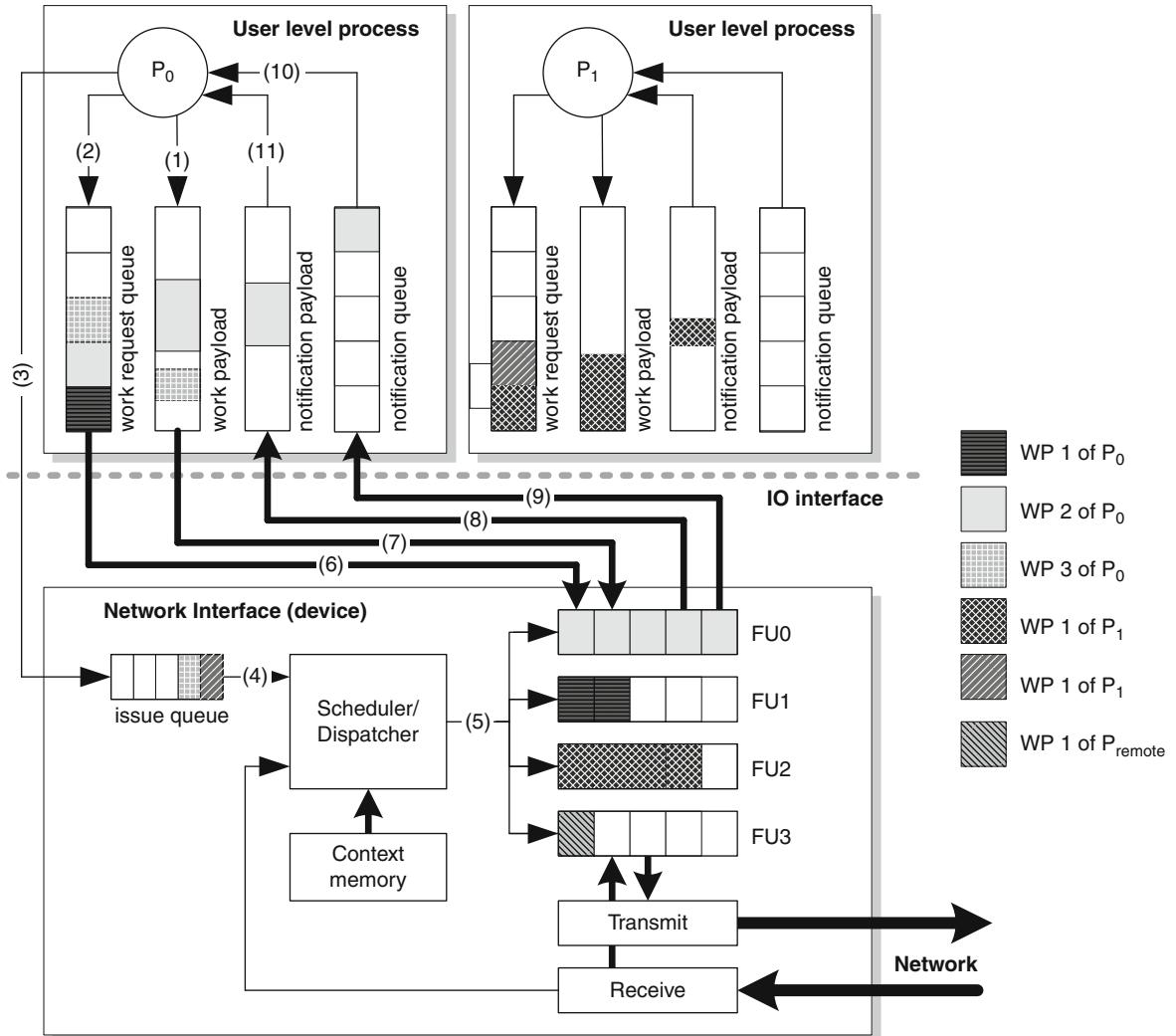
**Network Interfaces. Fig. 1 Work flow for multiplexed access.** Several processes are accessing the device by storing an optional payload (1) and a work request (2) in the appropriate exclusive data structures. Once a process has prepared such a work packet (WP), it notifies the supervising process or software instance using IPC or system calls (3). In the case of an unoccupied network device, the privileged process selects one user process and fetches the corresponding context information (4). It reads the work request entry from the queue (5) and checks it for correctness. It then forwards it to the network device's functional unit (FU) (6). The work request is enriched with information from the context, so the FU receives all information needed for processing. Also, if referenced it fetches the payload from the main memory data structure (7). It performs the processing (i.e., generation of a network packet) and then optionally writes back a notification (9), either with or without an associated payload (8). It also notifies the privileged process that it is free to receive a new work request (9). The user process consumes the notification (10), which optionally includes a reference to the payload (11)

typically relies on a context-switching approach, where a number of contexts are held on the device and upon an incoming request the corresponding context is used to configure the device for this process.

The work flow for a virtualized network interface is shown in Fig. 2: two processes are accessing a device for work processing, while an incoming packet from a remote process also requires processing. The interface between processes and device is here implemented by exclusive queues in main memory, only the issue queue on the device is shared. The combination of exclusive queue sets with only the issue queue being shared allows

bypassing any kind of privileged software instance. On the network interface, the scheduling/dispatching unit selects work packets either from local processes (i.e., from the issue queue) or from remote processes by incoming packets. It configures an unoccupied functional unit with the appropriate context, which can then process the work request.

Generally, a network interface must behave according to the properties of the whole network. For instance, if a network ensures reliability – that is, it guarantees that packets once injected are delivered without errors within finite time – the network interface must



**Network Interfaces. Fig. 2 Work flow for a virtualized device [12].** Several processes are accessing the device by storing optional payloads (1) and work requests (2) in appropriate exclusive data structures located in main memory. Once a process has prepared such a work packet (WP), it notifies the device by enqueueing an entry in the issue queue (3), which is consumed by the scheduler/dispatcher unit (4). This unit then configures a free functional unit (FU) for this process (5). The configured FU fetches the work request and optionally the payload. It performs the processing (i.e., generation of a network packet) and then optionally writes back a notification (9), either with or without an associated payload (8). The user process consumes the notification (10), which optionally includes a reference to the payload (11)

also ensure this. Similar applies for in-order delivery of packets: if packets are not allowed to bypass within the network, the same applies for the network interface. Other examples are Quality of Service or congestion management techniques. The latter is of particular interest, as many congestion management techniques use throttling of the injection rate to avoid or reduce congestions [11].

Considering the requirements above, the implementation of a high-performance network interface requires advanced architectural concepts. In particular, with regard to the highly parallelized environment a parallelization of the network interface's functional units seems appropriate. Concepts from modern processor architectures can be leveraged [12], for instance superscalar functional units, Simultaneous

Multi-Threading (SMT) [13] or Chip Multi-Processing (CMP) [5].

## Programming Paradigm

Architecture and functional behavior of a network interface are in particular dependent on the used programming paradigm.

Conceptually, there are several paradigms how communication and synchronization takes place in such a parallel computing system. The first paradigm is based on the exchange of messages, and this exchange is based on SEND and RECEIVE operations. For the purpose of communication or synchronization, or both, one process embeds the data as payload into a message and sends out this message to a remote process, which receives the message and unpacks the payload. The receiver decides where the payload is stored. Note that it is not possible to read remote locations directly, instead a message has to be sent in order to trigger a SEND operation with the data from the desired location. As both on the sending and receiving node processes are actively involved, this is also known as two-sided communication. The second paradigm is also based on the exchange of messages, but here PUT/GET operations are used. Here, the sender specifies where the data has to be stored, respectively read from. Thus, no receiving process is actively involved and one-sided communication takes place. Usually, notifications are used to inform target processes of completed operations. The last paradigm does not rely on the exchange of messages; instead, a global address space is set up. Each network node contributes to this global address space by allowing direct remote access to parts of its local memory. For communication and synchronization purposes, LOAD and STORE instructions on remote memory locations are used. Compared to the exchange of messages this approach is considered to be more intuitive for programmers, as it is similar to the multi-threaded programming paradigm used for multi-core architectures.

Depending on the used programming paradigm – either message passing or shared memory – a network interface must provide the appropriate functionality. A message passing network interface allows a process to send and receive messages, which includes for instance routing calculation (for source-path routed networks) and payload assembly (for descriptor-based interfaces).

Note that each operation – either send, receive, put, or get – is process dependent, because an appropriate set of queues or registers must be selected. Thus, support for multiple processes has to be implemented separately, for instance by hardware replication or by device virtualization. Opposed to this, a shared memory network interface (or “shared memory mapper”) has to perform an address translation from source local address to global address, and from global address to target local address. The destination is typically determined using parts of the global address. Then the operation is forwarded to the target node, where it is either completed or an appropriate response is sent back to the source node. As neither a LOAD nor a STORE instruction is process dependent, virtualization is provided inherently.

## Bibliographic Notes and Further Reading

An introduction to network interfaces covering the basics of different network interface types is provided by Dally and Towles in [1].

Practical examples for network interfaces can be separated into message passing and shared memory network interfaces: a well-documented example for a message-passing IO-attached network interface is Myrinet [14], which uses a network processor for protocol handling. Other examples for message passing network interfaces include Quadrics [15] and Infiniband [16]. Various other do exist, but due to commercial reasons they are mostly fairly documented. Further approaches to virtualize a network interface are described in [7, 17, 18].

Compared to the vast amount of message-passing-based networks, the number of shared memory networks is rather small. One of the first shared memory research projects is the Stanford DASH/FLASH [19, 21]. Examples for commercial projects implementing a global coherency include the SGI Origin [21] and SGI Altix [22]. A research project targeting a shared memory system with globally noncoherent but locally coherent memory is described by Yalamanchili et al. in [23]. Similar to this, the HyperTransport High Node Count specification [24] allows performing memory transactions on remote nodes, thereby forming a non-coherent shared memory system.

## Bibliography

1. Dally W, Towles B (2003) Principles and practices of interconnection networks. Morgan Kaufmann Publishers Inc
2. Tanenbaum AS (2002) Computer networks. Prentice Hall International
3. Duato J, Yalamanchili S, Ni L (2002) Interconnection networks: an engineering approach. Morgan Kaufman Publishers Inc
4. Litz H, Fröning H, Nüssle M, Brüning U (2008) VELO: A novel communication engine for ultra-low latency message transfers. In Proceedings of the 2008 37th international Conference on Parallel Processing (September 09 – 11, 2008). ICPP. IEEE Computer Society, Washington, DC, 238–245
5. Olukotun K, Nayfeh BA, Hammond L, Wilson K, Chang K (1996) The case for a single-chip multiprocessor. SIGPLAN Notices 31, 9 (Sep. 1996), 2–11
6. Goldberg RP (1974) Survey of virtual machine research. In Computer, pp 34–45, June 1974
7. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A (2003) Xen and the art of virtualization. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (Bolton Landing, NY, USA, October 19 – 22, 2003). SOSP '03. ACM, New York, NY, 164–177
8. Felten EW, Alpert RD, Bilas A, Blumrich MA, Clark DW, Damianakis SN, Dubnicki C, Iftode L, Li K (1996) Early experience with message-passing on the SHRIMP multicomputer. SIGARCH Computer Architecture News 24, 2 (May 1996), 296–307
9. Fröning H, Nüssle M, Slogsnat D, Haspel PR, Brüning U (2005) Performance evaluation of the ATOLL interconnect. In Proceedings of IASTED Conference: Parallel and Distributed Computing and Networks (PDCN), February 15 – 17, 2005, Innsbruck, Austria
10. Fröning H, Litz H, Brüning U (2009) Efficient virtualization of high-performance network interfaces. In Proceedings of the 2009 Eighth international Conference on Networks (March 01–06, 2009). ICN. IEEE Computer Society, Washington, DC, 434–439
11. Duato J, Johnson I, Flieh J, Naven F, Garcia P, Nachiondo T (2005) A new scalable and cost-effective congestion management strategy for lossless multistage interconnection networks. In Proceedings of the 11th international Symposium on High-Performance Computer Architecture (February 12 – 16, 2005). HPCA. IEEE Computer Society, Washington, DC, 108–119
12. Fröning H (2002) Architectural improvements of interconnection network interfaces. Doctoral Thesis, University of Mannheim, Germany
13. Tullsen DM, Eggers SJ, Levy HM (1998) Simultaneous multithreading: maximizing on-chip parallelism. In 25 Years of the international Symposia on Computer Architecture (Selected Papers) (Barcelona, Spain, June 27 – July 02, 1998). G. S. Sohi, Ed. ISCA '98. ACM, New York, NY, 533–544
14. Boden NJ, Cohen D, Felderman RE, Kulawik AE, Seitz CL, Seizovic JN, Su W (1995) Myrinet: A gigabit-per-second local area network. IEEE Micro 15, 1 (Feb. 1995), 29–36
15. Petrin F, Feng W, Hoisie A, Coll S, Frachtenberg E (2001) The quadrics Network (QsNet): High-Performance Clustering Technology. In Proceedings of the Ninth Symposium on High Performance interconnects (August 22 – 24, 2001). HOTI. IEEE Computer Society, Washington, DC, 125
16. Shanley T (2002) Infiniband network architecture. addison-Wesley Longman Publishing Co., Inc
17. Liu J, Huang W, Abali B, Panda DK (2006) High performance VMM-bypass I/O in virtual machines. In Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference (Boston, MA, May 30 – June 03, 2006). USENIX Annual Technical Conference. USENIX Association, Berkeley, CA, 3–3
18. Raj H, Schwan K (2007) High performance and scalable I/O virtualization via self-virtualized devices. In Proceedings of the 16th international Symposium on High Performance Distributed Computing (Monterey, California, USA, June 25 – 29, 2007). HPDC '07. ACM, New York, NY, 179–188
19. Lenoski D, Laudon J, Gharachorloo K, Weber W, Gupta A, Hennessy J, Horowitz M, Lam MS (1992) The stanford dash multiprocessor. Computer 25, 3 (Mar. 1992), 63–79
20. Kuskin J, Ofelt D, Heinrich M, Heinlein J, Simoni R, Gharachorloo, K, Chapin J, Nakahira D, Baxter J, Horowitz M, Gupta A, Rosenblum M, Hennessy J (1998) The Stanford FLASH multiprocessor. In 25 Years of the international Symposia on Computer Architecture (Selected Papers) (Barcelona, Spain, June 27 – July 02, 1998). Sohi GS, Ed. ISCA '98. ACM, New York, NY, 485–496
21. Laudon L, Lenoski D (1997) The SGI Origin: A ccNUMA Highly Scalable Server. In Proceedings of the 24th Annual International Symposium on Computer Architecture
22. Saini S, Jespersen DC, Talcott D, Djomehri J, and Sandstrom T (2008) Application-based early performance evaluation of SGI altix 4700 systems for SGI systems. In Proceedings of the 5th Conference on Computing Frontiers, Ischia, Italy
23. Yalamanchili S, Young J, Duato J, Silla F (2008) A Dynamic, Partitioned Global Address Space Model for High Performance Clusters. In CERCS Technical Reports, Georgia Institute of Technology, Georgia
24. Duato J, Silla F, Holden B, Miranda P, Underhill J, Cavalli M, Yalamanchili S, Brüning U, Fröning H (2009) Scalable Computing: Why and How. HyperTransport Consortium Whitepaper

## Network Obliviousness

KIERAN T. HERLEY

University College Cork, Cork, Ireland

### Synonyms

Architecture independence

### Definition

A parallel algorithm is network-oblivious if it is formulated in a manner that is completely independent of any machine-specific parameters (such as the number of

processors and communication characteristics) of any parallel machine that might ultimately be employed to execute it. The objective is to design algorithms which, although oblivious to machine parameters, nonetheless exhibit good performance across a range of parallel machines with differing computing and communication characteristics.

## Discussion

A central issue in the study of parallel algorithms is the search for a model of computation that strikes a balance between the three desirable qualities of *usability*, the suitability of the model to provide a simple, expressive, and convenient basis for the design and development of algorithms; *effectiveness*, its ability to capture enough of the essence of specific parallel machines to ensure that the algorithms produced are likely to perform well on such machines; and *portability*, the quality that allows an algorithm to execute with good performance across a range of machines with differing computing and communication characteristics.

The large and varied menagerie of existing models for parallel computation attests to the lack of a universal consensus on a model that blends all three qualities in satisfactory way. The fact, for example, that many texts on the subject of parallel algorithms, such as that of Leighton[8], are organized by machine architecture (mesh algorithms, hypercube algorithms, and so on), demonstrates that, in many cases at least, the most efficient algorithms are highly architecture-specific, essentially pointing to a conflict between effectiveness and portability.

The well-known PRAM (Parallel Random Access Memory) model is admirably usable, but is often criticized for its lack of effectiveness. Since the model does not explicitly model communication costs, it does not discourage profligacy with regard to communication, with the result that PRAM algorithms often perform poorly when related to real parallel machines.

Valiant's BSP (Bulk Synchronous Parallel) model [9] arguably strikes a better balance between usability and effectiveness. But, while it shields the algorithm designer from the finer details of the communication fabric of the machine being modeled, the model does force the algorithm designer to tune his algorithm specifically to a number of machine characteristics,

namely the number of processors and the bandwidth-latency parameters that BSP uses to model communication costs. In addition, BSP assumes a symmetric model of communication that takes no account of the fact that for machines involving a large number of processors, communication between "nearby" processors is likely to be less costly than that between more distant ones.

A natural question is whether it is possible to devise a framework that balances usability on the one hand with effectiveness and portability on the other. First, the model should allow algorithms to be formulated in a manner that frees the algorithm designer from any consideration of machine characteristics such as the number of processors, the interconnection structure, and the communication capabilities of the machine ultimately used to execute it. Second, the model should ensure that the algorithms produced exhibit good performance across a range of different machines with widely varying characteristics.

Somewhat surprisingly, at least for certain problems and certain classes of machine architectures, the answer to the question posed above is yes.

The results sketched in the remainder of this article consider only machine architectures that can be modeled by processor networks. In this model, a set of processor-memory nodes are interconnected by means of a set of channels that link pairs of nodes, and nodes may exchange messages directly only with their immediate neighbors.

Algorithms will be described in terms of a high-level model that takes no account of the number of processors or the specific interconnection structure. In that sense the algorithm is *network-oblivious* in the same spirit as the cache-oblivious algorithms studied by Frigo et al. [6]. It will be shown that network-oblivious algorithms can be devised for a variety of problems that while formulated to be completely independent of any machine characteristics, exhibit good (often optimal) performance for a wide range of machine architectures of widely varying characteristics.

## Cache-Oblivious Algorithms

The term "*network-oblivious*" echoes the concept of *cache-oblivious* algorithms in the sequential computing setting. Since the two concepts share certain ideas, the cache obliviousness concept is summarized briefly in this section.

It has long been recognized that the interplay between the execution of a sequential algorithm and the cache(s) on the underlying machine can have a significant effect on performance, yet the familiar RAM model traditionally used to formulate and evaluate sequential algorithms completely ignores all caching effects.

The cache-oblivious algorithm-design framework, formulated by Frigo et al. [6], allows the algorithm designer to formulate his algorithm within the familiar RAM framework, but the algorithm's performance is evaluated using to an idealized cache model that captures the amount of cache activity that the algorithm entails.

Informally, the model assumes a processor connected to a single-level cache of capacity  $Z$  words, partitioned into lines of  $L \leq \sqrt{Z}$  words each. The cache is in turn connected to the main memory, also partitioned into lines. A cache miss occurs whenever a request is issued for a word  $w$  that is not present in the cache. In this case, the line including  $w$  is copied from memory into the cache, replacing another line to make room. Algorithm performance is evaluated in terms of the work involved (i.e., the number of processor steps, as for the RAM model) and also of the number of cache misses.

It should be stressed that the algorithm designer must formulate his algorithm in a manner that is completely independent of the cache parameters  $Z$  and  $L$ . The objective is for the algorithm to have optimal or near-optimal performance regardless of the cache characteristics of the machine ultimately used to execute it.

To illustrate the idea, consider the problem of multiplying two square  $\sqrt{n} \times \sqrt{n}$  matrices  $A$  and  $B$  that are stored in row major order in memory. The following identity, based on a partition of the matrices into quadrants of size  $\sqrt{n}/2 \times \sqrt{n}/2$ , suggests a simple divide-and-conquer algorithm:

$$\begin{aligned} & \begin{pmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} \\ B_{1,0} & B_{1,1} \end{pmatrix} \\ &= \begin{pmatrix} C_{0,0} = A_{0,0}B_{0,0} + A_{0,1}B_{1,0} & C_{0,1} = A_{0,0}B_{0,1} + A_{0,1}B_{1,1} \\ C_{1,0} = A_{1,0}B_{0,0} + A_{1,1}B_{1,0} & C_{1,1} = A_{1,0}B_{0,1} + A_{1,1}B_{1,1} \end{pmatrix} \end{aligned} \quad (1)$$

The idea being that the eight  $\sqrt{n}/2 \times \sqrt{n}/2$ -sized subproblems  $A_{0,0}B_{0,0}, A_{0,1}B_{1,0}, \dots, A_{1,1}B_{1,1}$  that feature on the right-hand side of Eq. 1 are completed recursively, and then the results are combined as indicated to form the product  $A \cdot B$ .

Although this simple algorithm is not explicitly tuned to cache parameters  $Z$  and  $L$ , it turns out that it uses the cache optimally. No algorithm for this problem, including those specifically tuned to the cache parameters, can have an asymptotically better performance in terms of the work or number of cache misses. In other words, the algorithm, though completely oblivious to the cache characteristics, is competitive with those formulated in a cache-aware manner. Moreover, this algorithm is also optimal (under certain conditions) in relation to machines with multiple levels of caches.

The above algorithm exhibits a certain “self-tuning” quality. The recursive divide-and-conquer structure, entailing a series of subproblems of geometrically decreasing sizes  $\sqrt{n}, \sqrt{n}/2, \dots$ , implicitly exposes some spatial/temporal locality within the computation that the caching mechanism exploits to yield good performance. Once the subproblem size is sufficiently small that all the data related to it can fit in the cache, the subproblem can be completed without triggering any further cache misses.

Efficient cache-oblivious algorithms are also known for a variety of other problems such as matrix transposition, FFT (Fast Fourier Transform), sorting, among others. In most cases, the algorithm has the form a divide-and-conquer algorithm with highly regular structure. Some negative results are also known. Frigo et al. have also shown that algorithms that are efficient in the cache-oblivious sense often also exhibit good performance when executed on real machines.

## A Model for Network Obliviousness

To explore the concept of network obliviousness, Bilardi et al. [3] introduced the following framework for the formulation and evaluation of parallel algorithms. Algorithms are formulated in terms of a *specification model* and their quality assessed with respect to an *evaluation model*. The objective is that the framework provided by the pair of models offers a space for algorithm design and development that is conceptually simple, expressive

and easy to use, yet it gently constrains design in a manner that fosters the development of algorithms that are effective and portable.

The specification model, denoted  $M(n)$ , is an abstract machine embodying  $n$  numbered nodes  $P_0, \dots, P_{n-1}$ , each equipped with a processor and a private memory module. The computation consists of a sequence of supersteps in which each node may first complete some local computation involving data held within the node and then transmit messages to one or more other nodes. An  $i$ -superstep is a superstep in which nodes are constrained to send/receive messages only to/from nodes whose node number shares the same  $i$  most significant bits.

Note that the numbering of nodes induces a natural decomposition of the nodes of  $M(n)$ : the  $n/2^i$  nodes that share the same  $i$  most significant bits form a cluster, referred to as an  $i$ -cluster, and each  $i$ -cluster contains two  $(i+1)$ -clusters, each containing two  $(i+2)$ -clusters and so on. The  $\log n$ -clusters are individual nodes.

It must be stressed that  $n$  is an algorithmic parameter, typically related to the size of the problem instance (e.g., matrix size in the case of a matrix algorithm) and not intended to reflect the number of processors of the machine ultimately used to execute the algorithm.

The evaluation model, denoted  $M(p, B)$ , incorporates two parameters:  $p$  denoting the number of processing elements (each consisting of a processor plus local memory and referred to subsequently simply as a “processor” for brevity) and  $B$  the block size. The model shares the same clustering structure as  $M(n)$  and as well as a superstep-based mode of algorithm execution. In the case of the  $M(p, B)$  model, however, messages are aggregated into blocks of size  $B$  for transmission at the end of each superstep.

Consider the problem of matrix multiplication on the  $M(n)$  model. Suppose that two  $\sqrt{n} \times \sqrt{n}$  matrices  $A$  and  $B$  are distributed in row-major fashion among the nodes of  $M(n)$ . The product  $C = A \cdot B$  may be computed in a manner akin to that employed in the cache-oblivious setting seen earlier. Let  $P(i, j)$  denote the node that holds entries  $A[i, j]$  and  $B[i, j]$  (and ultimately also  $C[i, j]$ ).

- Partition the nodes of  $M(n)$  into eight clusters of size  $n/8$ . Distribute the elements of matrices  $A$  and  $B$ , so that each cluster has the matrix elements

required for one of the eight  $\sqrt{n}/2 \times \sqrt{n}/2$  subcomputations listed on the right-hand side of [Eq. 1](#).

- Recursively complete the eight subcomputations within the individual clusters.
- Distribute the elements of the results of the eight subcomputations so that  $P(i, j)$  receives the data it requires to compute  $C[i, j]$ .

Note that the algorithm is expressed purely in terms of the problem size parameter  $n$  and is completely oblivious to any machine characteristics.

Note that an algorithm  $\mathcal{A}$  for  $M(n)$  may be interpreted as an  $M(p, B)$  algorithm  $\mathcal{A}'$  by assigning groups of  $n/p$  consecutive nodes of the former to each processor of the latter, whose role is to simulate the actions of the  $M(n)$  nodes assigned to it. For communication steps,  $i$ -supersteps where  $i \geq \log p$  are handled internally within the  $M(p, B)$  processors (as the nodes of each  $i$ -cluster of  $M(n)$  are mapped to a single  $M(p, B)$  processor). Supersteps where  $i < \log p$  are handled as processor–processor communication, but with messages bound for a common destination bundled together into blocks of size  $B$  prior to transmission.

The *communication complexity* of a superstep is defined to be the maximum, over all processors, of the number of message blocks transmitted/received. The communication complexity of an algorithm is the sum of the complexities of its constituent supersteps. It turns out that the communication complexity of the matrix-multiplication algorithm on  $M(p, B)$  is  $O(n/Bp^{2/3})$  (provided that  $B \leq n/p$ ).

An algorithm  $\mathcal{A}$  for  $M(n)$  is optimal if the communication complexity  $C$  for its simulation  $\mathcal{A}'$  on  $M(p, B)$  is optimal, i.e., every  $M(p, B)$  algorithm for the same problem has a communication complexity that is  $\Omega(C)$ .

The  $O(n/Bp^{2/3})$  communication complexity for the matrix multiplication algorithm is in fact optimal (subject to certain technical assumptions). So the algorithm, though not specifically tuned to number of processors, is optimal with respect to the  $M(p, B)$  model.

To explore the effectiveness and portability of the algorithm, consider the performance of this algorithm on a network of processors  $p$  with a two-dimensional  $\sqrt{p} \times \sqrt{p}$  mesh interconnection, in which each processor can exchange messages directly only with its immediate neighbors.

Step 1 of the algorithm involves a communication step for the  $n$  elements of matrix  $A$  in which each processor is the source of  $n/p$  elements and the destination of  $2(n/p)$  (since each matrix quadrant is replicated twice). This can be accomplished in  $O((n/p)\sqrt{p}) = O(n/\sqrt{p})$  time on the mesh. The same bound applies to the distribution of matrix  $B$  and the communication involved in Step 3. Overall the communication cost of the algorithm is captured by a recurrence of the form  $C(n, p) = C(n/4, p/8) + O(n/\sqrt{p})$ , which has a solution  $T(n, p) = O(n/p^{2/3})$ . (Note that once a cluster size of one is reached, subproblems can be completed internally with a processor and no further communication is required.)

It turns out that this  $O(n/p^{2/3})$  complexity is optimal for the matrix multiplication problem for the square mesh in the network of processors model. In fact the  $\Omega(n/p^{2/3})$  lower bound holds (subject to some technical restrictions) even for the  $M(p, 1)$  model that takes no account of the nonconstant cost of communication between distant processors in a network of processors such as the mesh.

Thus the algorithm, developed in a manner that is completely oblivious to the structure and communication cost of the target architecture (in this case the two-dimensional mesh), is nonetheless optimal with respect to that architecture. Intuitively, the optimality of the algorithm stems from communication locality implicit in the hierarchical decomposition into subproblems of geometrically decreasing size and the mapping of these subproblems to smaller and smaller clusters (submeshes) in a way that exploits the lower communication costs within smaller submeshes.

It turns out that this phenomenon of efficient network-oblivious algorithms yielding efficient algorithms when executed on specific machine architectures is not unique to matrix multiplication or to the two-dimensional mesh architecture. The phenomenon extends to a variety of algorithmic problems and a range of different network architectures as well.

Optimal network-oblivious algorithms are also known for matrix transposition, FFT and sorting, for example. In each case, the optimality in the network-oblivious framework guarantees good performance when executed on a  $\sqrt{p} \times \sqrt{p}$  mesh. Efficient network-oblivious algorithms have also been explored for a

variety of other problems such as prefix sum, various dynamic programming algorithms, some variants of Gaussian elimination, among others.

Furthermore, the effectiveness of optimal network-oblivious algorithms applies to a wide class of network architectures, loosely speaking ones that exhibit a hierarchical structure in their pattern of interconnections and in their communication costs. Bilardi et al. [2] used the DBSP model (Decomposable BSP), a hierarchical variant of the well-known BSP model, to address the issues of effectiveness and portability of network-oblivious algorithms.

The DBSP model shares both the superstep-based structure of algorithm execution and hierarchical clustering structure of the  $M(n)$  and  $M(p, B)$  models, but has a more refined model for the costs of communication that better captures the communication characteristics of various network architectures. Specifically, it captures a notion of proximity: that communication with “nearby” processors should be cheaper than with those that are more distant. An  $i$ -superstep, in which processors may transmit messages to other processors within the same  $i$ -cluster, has a cost of  $\lceil \delta/B_i \rceil g_i$ , where the parameters  $g_i$  and  $B_i$  model the bandwidth-latency characteristics of communication within an  $i$ -cluster and  $\delta$  denotes the maximum number of words transmitted/received by any processor during the superstep.

The hierarchical nature of both the processor clustering and cost model for communication allows the DBSP model, for suitable choices of parameters  $(g_0, g_1, \dots, g_{\log p - 1})$  and  $(B_0, B_1, \dots, B_{\log p - 1})$ , to model a wide variety of architectures, such as meshes of various dimensions as well as fat trees. For example, a  $d$ -dimensional mesh architecture can be modeled by a  $p$ -processor DBSP with  $g_i = (p/2^i)^{1/d}$  and  $B_i = (p/2^i)^{1/d}$ . It should be noted that meshes of various dimensions are a common choice for interconnection on modern parallel machines. It can be shown that, under some reasonable assumptions on the pattern of communication embodied within the algorithm, an optimal network-oblivious algorithm yields an optimal DBSP algorithm and so optimal network-oblivious algorithms translate into efficient algorithms on architectures modeled by the DBSP, such as multidimensional arrays.

## Future Directions

To broaden the range of problems analyzed within the network-oblivious framework is an obvious line of further enquiry. An investigation of how near-optimal network-oblivious algorithms perform on various architectures as well as negative results (that certain problems do not admit optimal network-oblivious algorithms) might be valuable in casting light on what algorithm characteristics foster or frustrate network obliviousness. Clearly, further work that broadens the space of architectures to which these results apply would also be worthwhile. Work that extends these ideas into other settings, such as the multicore work referred to below, also looks promising.

## Related Entries

- [Bandwidth-Latency Models \(BSP, LoGP\)](#)
- [FFT \(Fast Fourier Transform\)](#)
- [Models of Computation, Theoretical](#)
- [Parallel Prefix Algorithms](#)
- [PRAM \(Parallel Random Access Machines\)](#)

## Bibliographic Notes and Further Reading

For a survey of parallel algorithms for various processor network architectures, see Leighton's text [8]. For a sample of algorithms designed from the PRAM model, see JáJá [7]. The BSP model was introduced by Valiant [9]. The decomposable BSP variant DBSP was first introduced by de la Torre and Kruskal in [5], but the variant used here was presented in Bilardi et al. [1, 2]. The latter paper also includes an interesting discussion of the interplay between usability, effectiveness, and portability. Cache obliviousness was introduced by Frigo et al. in [6].

The concept of network obliviousness was first formulated and explored by Bilardi et al. in [3]. A recent paper by Chowdhury et al. [4] develops a concept of multicore obliviousness in the same spirit of cache-/network- obliviousness. The paper also presents a number of network-oblivious results.

## Bibliography

1. Bilardi G, Pietracaprina A, Pucci G (1999) A quantitative measure of portability with application to bandwidth-latency models for

parallel computing. In: Euro-Par '99: Proceedings of the 5th international Euro-Par conference on parallel processing, Toulouse. Springer, London, pp 543–551

2. Bilardi G, Pietracaprina A, Pucci G (2007) Decomposable BSP: a bandwidth-latency model for parallel and hierarchical computation. In: Reif J, Rajasekaran S (eds) *Handbook of parallel computing: models, algorithms and applications*. CRC Press, Boca Raton
3. Bilardi G, Pietracaprina A, Pucci G, Silvestri F (2007) Network-oblivious algorithms. In: Parallel and distributed processing symposium (IPDPS 2007), Long Beach. IEEE International, Piscataway, pp 1–10, Mar 2007
4. Chowdhury RA, Silvestri F, Blakeley B, Ramachandran V (2010) Oblivious algorithms for multicores and network of processors. Proceedings of the IEEE 24th international parallel and distributed processing symposium (IPDPS), Atlanta, 19–23 Apr 2010
5. de la Torre P, Kruskal CP (1996) Submachine locality in the bulk synchronous setting (extended abstract). In: Euro-Par '96: Proceedings of the second international Euro-Par conference on parallel processing, vol II, Lyon. Springer, London, pp 352–358
6. Frigo M, Leiserson CE, Prokop H, Ramachandran S (1999) Cache-oblivious algorithms. In: FOCS '99: Proceedings of the 40th annual symposium on foundations of computer science, Washington, DC. IEEE Computer Society, Los Alamitos, p 285
7. JáJá J (1992) *An introduction to parallel algorithms*. Addison Wesley Longman, Redwood City
8. Leighton FT (1992) *Introduction to parallel algorithms and architectures: array, trees, hypercubes*. Morgan Kaufmann, San Francisco
9. Valiant LG (1990) A bridging model for parallel computation. Commun ACM 33(8):103–111

## Network of Workstations

Network of workstations (abbreviated as NOW) are clustered commodity workstations intended to function together as a cost-efficient parallel processing system. Their effectiveness in parallel processing is often facilitated by special parallel communication and management software layers and enhanced by high-performance system area networks.

## Related Entries

- [Clusters](#)

## Bibliography

1. Anderson TE, Culler DE, Patterson DA, the NOW team (1995) A case for NOW (Networks of Workstations). IEEE Micro 15(1): 54–64

## Network Offload

►Collective Communication, Network Support For

## Networks, Direct

OLAV LYSNE, FRANK OLAF SEM-JACOBSEN  
The University of Oslo, Oslo, Norway

### Synonyms

Distributed switched networks; Hypercube; Interconnection networks; K-ary n-cube; Mesh; Network Architecture; Ring; Router-based networks; Torus

### Definition

A direct network is a network for interconnecting a set of nodes in such a way that no external switching components are required. These nodes are themselves usually programmable computers. A switch is an integral part of the nodes, so that the node itself can be directly connected to a set of neighbor nodes. Non-neighbor nodes can be reached through multiple hops in the network. The integration of the switch with the node implies that there is a set of network topologies associated with direct networks that is distinct from the set of topologies associated with indirect networks.

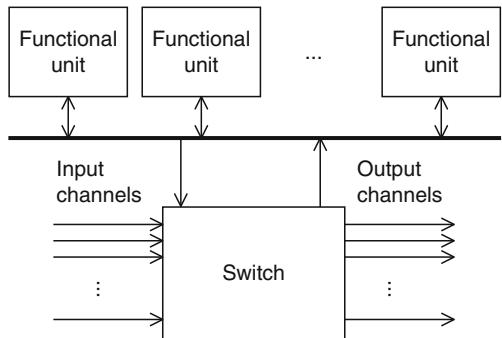
## Discussion

### Introduction

The preferred way to connect a small set of compute nodes to each other is to attach them all to a shared medium Bus. The main limitation of this approach lies in scalability. As the number of attached nodes increases, the total aggregate bandwidth of the Bus remains constant; thus, the Bus becomes a bottleneck.

A popular way to get around this scalability problem is to integrate switching capabilities into the compute nodes. This will allow a compute node to be directly connected to a subset of the other nodes in the system. Nodes that are directly connected to each other are generally referred to as *neighbors*.

A generic architecture of a node is depicted in Fig. 1. The Functional units depicted will typically be processors or memory, but other type units, such as graphic



**Networks, Direct.** Fig. 1 A generic architecture of a node for direct networks

processors or specialized vector processors are thinkable. Sometimes output channels and input channels are paired into bidirectional channels. In these cases, the channel interfaces denote the endpoint of bidirectional links. Still, the architectural layout of the node remains the same regardless of whether the links are unidirectional or bidirectional.

The direct network is constructed from a set of such nodes by repetitively coupling output channels of one node with input channels of another node until a topology of connected nodes has been built. There are many interconnection patterns that can be used in doing this, and each of them results in a defined class of topologies. The most well known classes of topologies are defined in a later section.

It is common to refer to the topologies generally used for direct networks as “direct network topologies,” and this chapter will follow the same convention. Although this wording hardly ever leads to misunderstandings, it is a bit inaccurate. All the topologies that can be applied for direct networks can equally well be the basis of the design of indirect networks (The opposite is, however, not always the case. Although it is possible to construct, e.g., a multistage network from switches that are integrated with nodes, the routing limitation of multistage networks makes it hard to obtain a fully connected and deadlock-free routing strategy for these cases.). Indeed, at the time of writing, there is a trend in industry to build switches as separate units, and in accordance with some communication standard, e.g., InfiniBand. Even if these switches are not integrated with a compute node, they are used to build systems with both direct-and indirect-network topologies.

Direct networks provide very good flexibility, both with respect to scalability and topology. The extension of a system with additional nodes is in many cases cost effective, as the needed extension in communication infrastructure is embedded in the node itself, and because the necessary connection points in the existing system are provided by the nodes already in the system.

## Topologies

*The network topology defines the interconnection pattern among nodes.*

The usual way of modeling network topologies is as a graph  $G(N, C)$  where the vertices, denoted by  $N$ , represent the set of processing nodes, and the edges  $C$  represent the set of links. The links may be defined to be unidirectional or bidirectional, based on the properties of the technology under study. In some cases it is necessary to model a bidirectional link of some technology as a pair of unidirectional links. In particular, this is necessary for the analysis of deadlock properties of routing algorithms.

In the study of topologies, there is a set of notions that are central. These stem partly from the requirements that the topology places on the nodes, and partly on the properties of the topology itself. *Node Degree* is a notion from the first category. This is the (minimal) number of external input and output channels that the topology requires of each node. Another example is *Forwarding Logic*. Topologies that are of a regular structure can be supported by a specialized logic for efficient and simple forwarding of packets tailored for that particular topology.

*Network diameter* is a direct property of the topology itself. It is the maximal length of the minimal path between any pair of nodes measured in number of intermediate nodes that have to be traversed. Much early research on topologies focused on solutions that minimized the diameter. The introduction of *Cut Through Switching* in the early 1980s did, however, make packet latency in a network to a large extent independent of the number of hops.

*Bisection Bandwidth* is a notion that captures the ability of a topology to handle bandwidth requirements. It is defined as the minimum aggregate bandwidth of a set of links that when they are removed divide the network in two equal-sized halves. Although this notion is important, its relevance is not independent of the

application that the network should support. The main strength of some direct-network topologies is that they support very well some applications that require communication only with neighboring nodes. In such a scenario, the bandwidth of a single link is important, but the aggregate bandwidth across the bisection is of less importance.

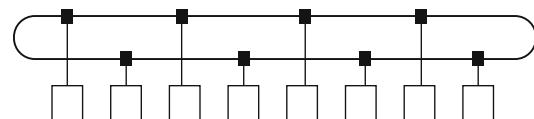
## Rings

The simplest extendible direct network topology is the ring (Fig. 2). The ring may be unidirectional or bidirectional. For rings, the nodes only need one input and one output channel. The forwarding logic can be made quite simple, in that every node compares the address of each packet passing by. If the address matches the node's own address, the packet is delivered to the internal parts of node itself. Otherwise, it is forwarded to the next node in the ring. Some technologies, like SCI, are tailor-made for ring structures, and thus take advantage of easy forwarding and low-node degree.

The bisection bandwidth and the network diameter of rings are generally considered as the weak points of these structures. The network diameter grows linearly with the number of nodes, and the bisection bandwidth remains constant whatever the size of the ring. This severely limits the size of the systems built as a single ring. Systems of rings can, however, be constructed to improve on this weakness. The most prominent example of this is the Torus-class of topologies. These are discussed below.

## Meshes

The  $n$ -dimensional mesh topology can be explained by imagining an  $n$ -dimensional grid, with a node in each cross-point. Each node has one bidirectional link connecting it to each of its closest neighbors in each dimension. This means that the nodes have at most  $2n$



**Networks, Direct. Fig. 2** This is an example of a ring with eight nodes. Note that the nodes on each side of the ring are interleaved, so that the length of the cable between each pair of nodes becomes uniform

neighbors, thus the switching element of the nodes must have at least  $2n$  ports to support an  $n$ -dimensional mesh. Note also that the nodes that are placed at the extreme of a dimension will have fewer than  $2n$  neighbors. The nodes that are at the extreme of all dimensions will have exactly  $n$  neighbors.

A more formal definition is as follows: An  $n$ -dimensional mesh has  $k_0 \times k_1 \times \dots \times k_{n-1}$  nodes, with  $k_i$  nodes along each dimension  $i$ . Let  $k_i$  be the *radix* of dimension  $i$ . Each node  $X$  in the topology is uniquely identified with a tuple of  $n$  coordinates identifying its position in the grid. If a node  $X$  in the topology is identified by  $\{x_0, x_1, \dots, x_{n-1}\}$ , then for all  $i$ ,  $0 \leq x_i < k_i$ . Two nodes are neighbors, and thus are directly connected to each other through a link, if and only if their identifying tuples of coordinates are identical in all places, except for exactly one coordinate  $i$  where they differ with exactly one (Fig. 3).

The bisection bandwidth of a mesh equals the link speed multiplied with the product of the number of nodes in each dimension and divided by the number of nodes in the dimension with the highest number of nodes. This means that when the radix of the dimensions is balanced,  $M$  is the number of nodes and  $n$  is the number of dimensions, the bisection bandwidth of a mesh grows with  $M$  divided by the  $n$ 'th root of  $M$ . The diameter of the mesh is the radix of the dimensions multiplied with the  $n$ 'th root of  $M$ .

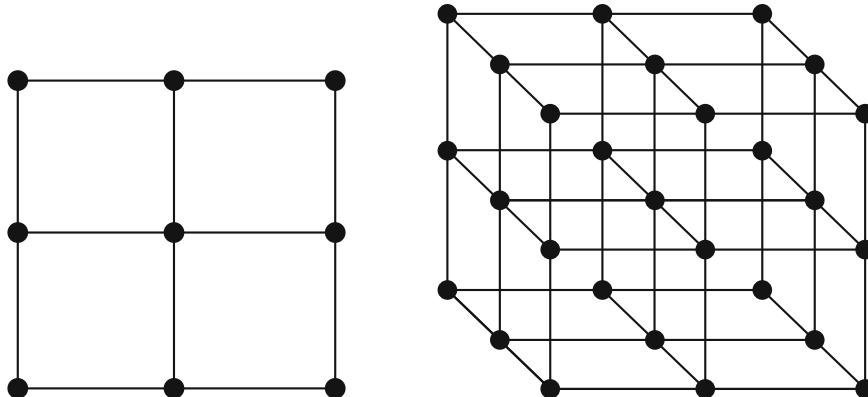
If specialized forwarding logic for meshes is implemented in a node, this generally means that the addressing scheme is based on the  $n$ -tuple that identifies the position of each node in the mesh. Typically then, each

forwarding step consists of moving the packet one step closer to its destination along one dimension. This is done by first identifying one digit in the  $n$ -tuple where there is a mismatch between the destination address of the packet and the address of the node that currently handles it. In the dimension identified by the digit, the packet is forwarded in the direction that decreases the mismatch. When all mismatches have been nullified, the packet has reached its destination.

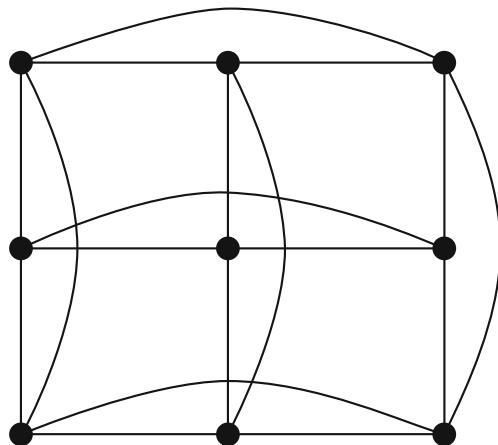
## Torus

Torus topologies are extensions of meshes with additional links. In the same way as for meshes, one can describe the placements of the nodes by referring to cross-points in a grid, and in the same way as for meshes, the nodes can be identified by tuples defining their placement in each of the  $n$  dimensions of the grid. The only topological difference is that there are additional links in a torus compared to a mesh. These links connect the first and the last node in a dimension, so that each sequence of nodes along each dimension forms a ring (*Some authors will define a torus such that the extra links are not there when the size of the dimension is two. Following that definition, the definition of torus and meshes coincide for this case. Our later discussion of bisection bandwidth for tori would not be valid for dimension-size of two under that definition.*) (Fig. 4).

The implication of this is that in a torus, each node has exactly the same number of neighbors, equaling twice the number of dimensions in the torus. It also means that the bisection bandwidth of the torus is



**Networks, Direct. Fig. 3** A two-dimensional and a three-dimensional mesh



**Networks, Direct.** Fig. 4 Example of a 2-dimensional torus

two times that of the mesh. The diameter of the torus approaches half that of the mesh when the network is large (*There is a subtlety in that the diameter of each ring in the torus is calculated differently depending on whether the number of nodes in the ring is odd or even. The effect of this subtlety approaches zero when the rings grow large.*). Specialized forwarding logic is possible for tori in the same way as for meshes. The notion of distance will, however, need to be more involved to capture the possibility for following the shortest path along wraparound links.

## Hypercube

The hypercube can be defined in several ways. One way is to define it as a mesh where the radix of each dimension is exactly two. The following recursive definition is more formal:

- A single node is a hypercube.
- Two identical hypercubes, where each isomorphic pair of nodes from the two hypercubes has a link between them, is a hypercube.

The construction of hypercubes from this definition is illustrated in Fig. 5.

Since hypercubes are a special case of meshes, the discussion of port requirements in the nodes, of forwarding logic in the nodes, the diameter, and the bisection bandwidth follows from the discussion above.

Still, it is worth reflecting over the implication of limiting the radix to two. First, it means that the only way to extend the network is to add another dimension.

This means that for each node that has been designed, the number of ports implemented in the node limits the size of the network that can be built. On the other hand, the bisection bandwidth grows linearly with the number of nodes, meaning that the bisection bandwidth per node remains constant. Furthermore, the diameter of the network equals the number of dimensions in the system, giving a diameter that is logarithmic to the number of nodes in the system.

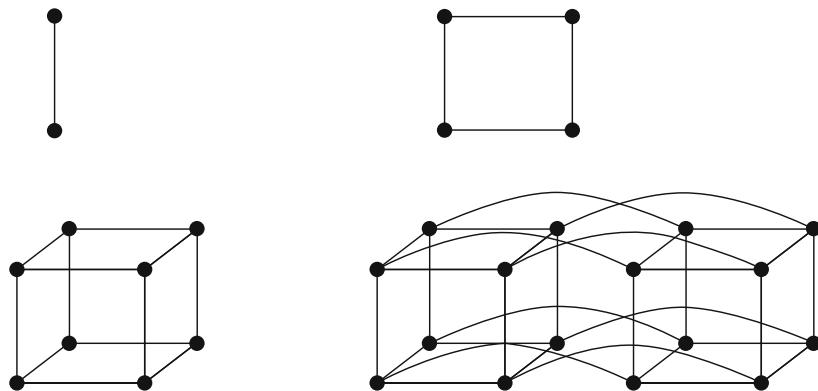
## Other Topologies

In addition to rings, meshes, tori, and hypercubes, a plethora of topologies have been defined in the literature. Most of these additional topologies were designed with the goal of minimizing the network diameter for a given number of nodes and/or a given node degree. With the introduction of pipelined switching techniques, like wormhole routing and Virtual Cut Through, end to end packet network latency is close to independent of network diameter. None of these topologies are therefore likely to be implemented. For a short introduction to some of these topologies, we refer to Sect. Popular Network Topologies of [1].

## Routing Functions

The routing functions in use for these topologies are quite similar to each other. At the base there is shortest path routing that follows one dimension at a time. Basically, the set of dimensions are ordered in a sequence, and the dimensions are traversed in this order. For meshes and hypercubes, this guarantees deadlock-free routing. For rings and tori, however, the cyclic structures in the topologies creates additional challenges, as shortest path routing will create cycles of dependencies between the buffers and the channels, and thus there is a danger of deadlock. In these cases, the solution is to introduce virtual channels, such that the cycle of dependencies between the different resources in the network is broken. More information on this is given in the entry on routing in this encyclopedia.

As the size of the networks grows, the importance of the system to remain in operation regardless of whether some of the nodes fail becomes important. Several routing strategies that cope with faulty nodes in direct network have been proposed, but very few of them have so far been implemented. More information on this can be found in the entry on fault-tolerant networks.



**Networks, Direct.** **Fig. 5** This is an illustration of the development of hypercubes from the recursive definition. Top left is a one-dimensional hypercube consisting of two nodes connected together. Top right is a two-dimensional hypercube, bottom left a three-dimensional hypercube, and bottom right is a four dimensional hypercube

### Direct Networks Today

For smaller networks, the choice of topology is relatively easy. Most solutions will work. Presently, one-dimensional meshes (or daisy chains) are typically used for HyperTransport, and simple rings are used for SCI and interconnecting the processing elements in the Cell Processor and the Intel Larrabee. The discussion does not really get involved until the system size increases to interconnect more than 10–12 nodes.

The properties of Hypercubes with respect to diameter and bisection bandwidth are very appealing. Still, real machines have to be built in the three available dimensions. This has limited the use of Hypercubes to medium-sized systems. For the recent systems that have been built with thousands of nodes, the preferred direct network topology has therefore been a three-dimensional torus.

When looking at the Top 500 list of supercomputers in the world, one finds both direct topologies, and indirect topologies – typically multistage networks. It is a well-established fact that for arbitrary traffic, multistage networks give superior price/performance compared to the direct topologies. Still, tori and meshes appear in the list. The reason for this is that many of the computational problems that run on such computers analyze physical systems in a three-dimensional world. A torus topology allows the physical world to be divided into three-dimensional cubes of equal size, and let these cubes be mapped isomorphically onto the compute nodes in the system. Since the physical influence between these cubes are between

neighbors, the communication in the computer system will mainly go between neighboring nodes, and thus the locality in the torus/mesh topology fits ideally with communication needs. Therefore, the current trend is that for multi-purpose installations, multistage topologies are used. For installations that are specialized to run a limited set of applications modeling the physical world, tori or meshes are still the topology of choice.

### Future Directions

There are few new developments in network topologies after the turn of the millennium. Still, the recent changes in technology have revitalized the field somewhat. The development in processor technology has made it possible to put many compute cores onto one chip. This has spawned the need for an on-chip interconnection network. Because of its square layout in two dimensions (or three when one considers different metal layers) have made meshes and tori look as obvious candidates for such networks.

The quest for simple integration of different components onto the same chip seems to indicate that the integration of switching capacity and functionality onto the same logical unit will increase. From this observation it is a reasonable guess that direct topologies will be preferred. There are, however, several reasons why our knowledge of direct network topologies will have to be extended to support on-chip networks.

The first reason is that the different functional units that are to be integrated onto the same chip will in many

cases not be uniform. They will have different characteristics, and different need for footprint on the chip. For that reason, it is not at all obvious that the on-chip network will be regularly structured in the same way as tori and meshes are. The second reason is that the traffic requirements will not be uniform either. On-chip networks will frequently be used in single application scenarios. The quest for minimizing power and area utilization will make designers deviate from regular structures whenever the communication needs will allow it.

Finally there are currently initiatives working in the direction of streamlining on-chip, on-board, and between-board communication. These initiatives may result in common addressing solutions. Still, the different requirements of these domains will most likely mean that there will be different network topologies in each of them. Efficient co-working of several networks with different topologies is therefore likely be a topic addressed in the future.

## Related Entries

- ▶ [Buses and Crossbars](#)
- ▶ [Cray XT4 and Seastar 3-D Torus Interconnect](#)
- ▶ [Ethernet](#)
- ▶ [Hypercubes and Meshes](#)
- ▶ [InfiniBand](#)
- ▶ [Myrinet](#)
- ▶ [Networks, Fault-Tolerant](#)
- ▶ [Network Interfaces](#)
- ▶ [Networks, Multistage](#)
- ▶ [Routing \(Including Deadlock Avoidance\)](#)
- ▶ [SCI \(Scalable Coherent Interface\)](#)
- ▶ [Switch Architecture](#)
- ▶ [Switching Techniques](#)
- ▶ [Topology Aware Task Mapping](#)
- ▶ [Quadrics](#)

## Bibliographic Notes and Further Reading

There have are several sources to the state of the art in direct networks. The two most important books on interconnection networks both contain good overviews [1, 2]. Furthermore, there is a treatment of topologies for interconnection networks in an appendix of the book by Hennessy and Patterson [3].

Direct networks were used on some of the earliest parallel computers [7, 8]. Later, in particular inspired by the Caltech Cosmic Cube [9], there were a series of approaches using hypercube-like structures [10, 11].

William Dally did much seminal work on direct networks in the late 1980s. The one publication most relevant to the discussion on topology here was [4], where he made the case for low dimensional, high radix networks to have better scalability properties than the high dimensional hypercubes. Other important contributions were done together with Charles Seitz on mechanisms for deadlock-free routing on tori [5, 6]. From this point, two and three-dimensional meshes and tori have been the direct network of choice [12, 13]. A recent example that demonstrates the lingering relevance of these topologies is the Red Sky machine at Sandia National Labs. This machine was delivered in November 2009 and made number 10 on the Top 500 list then. Its interconnection network is a three-dimensional torus.

## Bibliography

1. Duato D, Yalamanchili S, Ni L (2002) *Interconnection networks – an engineering approach*. Morgan Kaufmann, San Francisco
2. Dally WJ, Towles B (2004) *Principles and practices of interconnection networks*. Morgan Kaufmann, San Francisco
3. Duato D, Pinkston T (2007) Appendix E: Interconnection networks. In: Hennessy JL, Patterson DA (eds) *Computer architecture – a quantitative approach*, 4th edn. Morgan Kaufmann, San Francisco
4. Dally WJ (1990) Performance analysis of k-ary n-cube interconnection networks. *IEEE Trans Comput* 39(6): 775–785
5. Dally WJ, Seitz CL (1987) Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans Comput* C-36(5):547–553
6. Dally WJ, Seitz CL (1986) The torus routing chip. *J Distributed Syst* 1(3):187–196
7. Barnes GH (1968) The ILLIAC IV computer. *IEEE Trans Comput* 17(8):746–757
8. Slotnick DL, Borck, WC, McReynolds RC (1962) The Soloman Computer. In: *Proceedings of the AFIPS Spring Joint Computer conference*, vol 22, Spartan Books, New York, pp 97–107
9. Seitz CL (1985) The cosmic cube. *Commun ACM* 28(1):22–33
10. Close P (1988) The iPSC/2 node architecture. In: *Proceedings of the conference on hypercube concurrent computers and applications*, Pasadena, USA, pp 43–55, Jan 1988
11. Palmer JF (1986) The NCUBE family of parallel supercomputers. In: *Proceedings of the international conference on computer design*, pp 177–188, San Diego, USA, 1986

12. Kessler RE, Schwarzmeier JL (1993) Cray T3D: a new dimension for Cray Research. In: Proceedings of the IEEE Computer Society International Conference (COMPON), San Francisco, USA, pp 176–182, Feb 1993
13. Scott SL, Thorson GM (1996) The cray T3E networks: adaptive routing in a high performance 3D torus. In: Proceedings of the symposium on hot interconnects, Standord 1996, pp 147–156

## Networks, Fault-Tolerant

MARÍA ENGRACIA GÓMEZ REQUENA  
Universidad Politécnica de Valencia, Valencia, Spain

### Synonyms

[Reliable networks](#)

### Definition

A fault-tolerant network is a network that can provide full connectivity among all the processing nodes connected to the network without losing any messages despite the presence of faults in the network elements.

### Discussion

#### Introduction

Large parallel computers with thousands of nodes have been and are being built as can be seen in the top 500 list [1]. In such systems, the high number of components significantly increases the probability of failure. Each individual component can fail with a certain probability and, thus, the probability of failure of the entire system increases dramatically as the number of components increases. In addition, the failure probability of each individual component increases as transistor integration density increases. Hence, in these systems, it is critical to keep the system running even in the presence of faults. Moreover, faults in the interconnection network may isolate a large fraction of the machine, containing healthy processing nodes. Therefore, fault-tolerant mechanisms for the interconnection network are critical design issues for large parallel computers. Fault-tolerance in an interconnection network refers to the availability of the network to function in the presence of component faults.

However, techniques used to implement fault-tolerance in the interconnection network are often at

the expense of considerable performance degradation, which is not desirable. A good fault-tolerance mechanism should degrade network performance gradually with the number of faults. Also each fault-tolerance mechanism can tolerate a certain number of faults. Beyond this number, an additional fault may disconnect the network.

The solution adopted to tolerate faults in a given system depends on the types of faults that can occur and the assumed fault-model. The pattern of faults and expectations about the behavior of processors and switches in the presence of these faults determines the approaches to provide fault-tolerance. This information is gathered in the fault model.

#### Fault Models

In the design of fault-tolerant techniques for the interconnection network, a fault model is assumed. In what follows, the most commonly-used fault models are presented [2].

First, there are two main kinds of faults in the interconnection network: transient and permanent. Permanent faults remain in the system until it is repaired. On the other hand, transient faults are dynamic in nature as integrated circuit sizes continue to decrease and speeds continue to increase. As it can be read below, the approaches followed to deal with both fault types are completely different.

Another consideration is the level at which components are detected as faulty. Usually, detection mechanisms differentiate between switch faults and link faults since most types of faults will simply manifest as link or switch faults. In the case of a switch fault, all links incident on the faulty switch are also considered as faulty.

The fault model also specifies the area of the fault information available at each node to tolerate the faults. On one hand, there are techniques in which each node only has fault information about adjacent nodes. On the other hand, in other techniques, every node knows the fault status of every other node. The advantage of the latter techniques is that messages can be forwarded along the shortest feasible path in the presence of faults. However, in practice it is difficult to have global fault information in a timely manner. These techniques require storage and computation time which have a significant impact on performance, and the occurrence

of faults during update periods requires complex synchronization protocols. In case of local information, nodes only have fault information about adjacent nodes; so routing decisions are simpler and can be computed quickly, and updating the fault information about adjacent nodes can be performed in a very simple way. However, in this case, routing decisions are not optimal and messages may be routed to network zones with faults, leading to longer paths. In practice, fault-tolerant techniques are between purely local and purely global fault status information. Fault diagnosis deals with developing algorithms for getting and maintaining timely fault information about neighbor nodes.

On the other hand, depending on the actions performed after the fault detection, two different types of fault models can be defined: static or dynamic. In a static fault model, once a fault is detected, all the processes in the system are halted, the network is emptied, and a management application is run in order to deal with the faulty component. This application detects where the fault is and computes the information required by the nodes in order to tolerate the fault. This information is distributed by the management application and, then, the system is rebooted and the processes are resumed. This fault model needs to be combined with checkpointing techniques in order to be effective. Applying checkpointing minimizes the fault's impact on applications, because they are restarted from the latest checkpoint. In a dynamic fault model, once a new fault is found, actions are taken in order to appropriately handle the faulty component while the system keeps running. For instance, a source node that detects a faulty component through a path can switch to a different path that does not use the failed component.

### Transient and Permanent Faults

As commented above, faults can be classified as transient and permanent [3, 4].

Transient faults are usually due to electromagnetic interferences and alpha particle impacts and they are usually corrected by retransmitting the packet either at the link level or at the end-to-end level. A link level treatment is performed by encoding redundant information in the packet using an error control code (ECC). Simple parity is enough to detect a single error of a bit. However, most links use a cyclic-redundant check (CRC) of sufficient length in order to reduce the

probability of an error of several bits. The error check can be performed at different levels of granularity: flit or packet. When the sender transmits the packet or flit, it retains a copy until correct reception is notified by the receiver. The redundant information included in the packet or flit is checked when it is received. If the packet or flit pass the test, the receiver sends an acknowledgment indicating that it has been received correctly. Each packet or flit is tagged to facilitate its identification and this tag is used in the acknowledgment to identify the packet or flit. The acknowledgment also indicates the reception status, that is, received correctly or received in error. Upon receiving this acknowledgment, the transmitter discards its copy of the packet. If a packet is received in error, the sender retransmits it. In most cases, the error is transient and the packet is received correctly on the second attempt.

Transient faults can be also treated by end-to-end mechanisms that operate in a similar way to that of link-level retransmission, except that packets are retransmitted over the whole network path up to the final destination [4]. In this case, as in the link-level mechanism, the sender must store a copy of each packet pending of its correct reception. The destination sends an acknowledgement when the packet arrives correctly. It contains the tag assigned to the packet. In this case, due to having longer paths, a timer is usually used at the sender node for each sent packet. If the timer of a given packet expires before an acknowledgement of that packet arrives to the destination or if a negative acknowledgement is received, then the source resends the packet. However, two copies of the same packet can be received at the destination if the acknowledgement arrives at the source delayed after the timer has expired. This can be identified in an easy way since both packets have the same tag, and the second packet is discarded.

On the contrary, permanent faults are due to components that stop working within specifications and cannot be recovered by retransmitting packets with the help of some higher-layer software protocol. In order to tolerate this type of faults, an alternative physical path must exist in the network topology.

Three main groups of techniques are used to deal with permanent faults in the interconnection network: resource replication, network reconfiguration, and fault-tolerant routing.

## Replication

Replication provides tolerance to permanent faults by replicating network components. Those spare components are switched on in case of fault and the faulty components are switched off (or bypassed). As an example, the ServerNet interconnection network is designed with two identical switch fabrics. Only one of the switch fabrics is available at any given time. In case of fault in one of them, the other fabric is used.

This technique is used by a large amount of fault-tolerant proposals for MINs. They add either links between switches in the same stage, more links between stages, more switches per stage or extra stages to increase the number of alternative paths. In [14], a new topology that consists of two parallel fat-trees with crossover links between the switches in the same position in both networks is proposed. In this topology, when a packet encounters a fault in its path, it is forwarded through the crossover link to the other parallel fat-tree.

The main drawbacks of this approach are the high extra cost of spare components and the non-negligible probability of failure of the circuits required to switch to spare components.

This technique can also be implemented without using spare resources, leading to a degraded mode in the system operation after a fault. The IBM Blue Gene/L supercomputer, for instance, offers the ability to bypass faulty network resources while keeping its topology and routing algorithm. In case a fault is detected, all the nodes included in several midplane boards that contain the faulty node/link are bypassed to isolate the fault. In this way, the network topology remains the same and routing is not changed. The main drawback of this technique is the relatively large number of healthy resources (i.e., midplane node boards) that may be switched off after a fault.

## Network Reconfiguration

Network reconfiguration relies on the use of routing tables as the routing mechanism. It is a more general technique to deal with changes in the network topology due to failures or to some other cause. When a fault appears, the routing tables are reconfigured to be adapted to the new topology resulting after the network topology change. In order to perform the network reconfiguration, the fault-free portions of the topology

must first be discovered, then the new routing tables must be computed and finally they are distributed to the corresponding network components (i.e., switches and/or end node devices).

Network reconfiguration requires the use of programmable switches and/or network interfaces, depending on how routing is performed. It may also make use of generic routing algorithms (e.g., up\*/down\* routing) that can be configured for all the possible network topologies that may result after faults. This technique relieves the designer from having to supply alternative paths for each possible fault combination at design time and is extremely flexible because it allows tolerating faults while there is physical connectivity in the network. The drawbacks of this technique are that it uses generic routing algorithms that are valid for any topology, but these routing algorithms achieve poor performance when they are used in regular networks, and that programmable network components have higher cost and latency. Most standard and proprietary interconnection networks for clusters and SANs – including Myrinet, Quadrics, InfiniBand, Advanced Switching, and Fibre Channel – incorporate software for (re)configuring the network routing in accordance with the new topology.

Another issue related to network reconfiguration is whether the interconnection network supports hot swapping. That is, whether the interconnection network can continue operation while a new component is added to or removed from the network. Most networks allow for hot swapping which is usually supported by performing a dynamic network reconfiguration without stopping network traffic.

The main problem with network reconfiguration is guaranteeing deadlock-free routing while routing tables are updated in switches and/or end node devices, as more than one routing algorithm may be used in the network at the same time. Most WANs solve the possible deadlocks by dropping packets, but dynamic network reconfiguration is much more complex in lossless networks.

## Fault-Tolerant Routing

Fault-tolerant routing takes advantage of the multiple paths usually available in the network topology between each source-destination pair, in order to route packets through the fault-free paths. Adaptive routing is usually

used in this case. The faulty components are made unavailable to the adaptive routing algorithm and all the packets are routed using the remaining components. Nevertheless, it is also possible to achieve network fault-tolerance with deterministic routing by routing a packet in two or more phases and storing it at some intermediate nodes.

The main difficulty of these routing algorithms is guaranteeing that the routing algorithm will remain deadlock-free and livelock-free after the fault appears given that arbitrary fault patterns may occur.

The solution adopted to provide fault-tolerance depends on the type and pattern of the component faults and the network topology. In what follows, different routing algorithms for regular networks are presented and grouped depending on the type of the regular topology, that is, direct or indirect.

### Fault-Tolerant Routing in Direct Networks

A large number of fault-tolerant routing algorithms for multiprocessor systems have been proposed, especially for mesh and torus network topologies.

In multidimensional direct topologies, if the number of faulty components is less than the degree of a node, then the topology provides at least a physical path between each source-destination pair, and therefore the routing algorithm may communicate any two nodes of the system.

In direct networks, guaranteeing that the routing algorithm remains deadlock-free after the fault appears and when using the alternative paths to avoid the fault is especially difficult since the faults can comprise its regularity.

Some approaches used in direct networks use global status information of the network, whereas others use only local status information.

In case of local information, faulty components can be avoided by using a dimension orthogonal to the dimensions leading to the faulty components. Chien and Shin [5] propose such an algorithm for hypercubes. When all of the shortest paths to the destination from an intermediate node are blocked by faults, the message is transmitted using an extra dimension that is not in the minimal path. The algorithm is  $n-1$  fault-tolerant.

Another alternative approach is the use of randomization to produce probabilistically livelock-free,

non-minimal, fault-tolerant routing algorithms. The idea is to avoid repetitive sequence of link traversals in order to avoid livelocks. The Chaos router incorporates randomization. In this case, messages are normally routed along any profitable output port. When a message is stored in an input buffer for too long, blocked by busy or faulty output buffers, it is removed from the input buffer and stored in a local buffer that has a higher priority and allows routing through non-minimal paths.

The previously presented approaches assumed the availability of only information about neighboring links and switches. An approach to maintain global fault information is by encoding the node state. Lee and Hayes [6] introduce the concept of unsafe node. A non-faulty node is unsafe if it is adjacent to two or more faulty or unsafe nodes. Each node must know the state of its neighbors, so each node must transmit its state information to the neighboring nodes. The unsafe state indicates to the routing algorithm that going through another path is a better routing option, that is, going through a safe neighbor node.

When maximum flexibility must be ensured, routing algorithms based on graph search techniques are used. A message can be routed to traverse a set of links corresponding to a systematic search of all possible paths between a pair of nodes. The routing overhead is significant, but the probability of delivering messages is maximized. A main issue in this case is that of passing through a node more than once which results in an unnecessary overhead. Most approaches maintain information in the header to avoid passing through the same node several times.

The aforementioned fault-tolerant routing algorithms are defined for SAF or VCT switching. When wormhole switching is used, new challenges appear in the design of fault-tolerant routing algorithms. The use of small buffers makes that blocked messages span multiple nodes, hence the fault affects several nodes. In this case, the most common approach is to define fault regions and route packets avoiding those regions. Adjacent faulty links and faulty switches are joined into fault regions. The idea is not to introduce new dependencies in the routing algorithm in order to guarantee deadlock-freedom.

The difficulty of routing messages around faulty regions depends on the shape of the region. The fault tolerant routing algorithms usually impose constraints

to the regions. In particular, they usually assume that the fault regions are convex, since concave regions present more difficulties. Moreover, in some algorithms convex regions are further constrained to be block fault regions, that is, regions whose shape is rectangular. Given a set of faults in a direct network, rectangular fault regions can be constructed by marking some healthy nodes as faulty to fill out the entire region. In this case, the fault-tolerant routing algorithm usually needs additional resources as virtual channels to avoid deadlocks.

Assuming block fault regions, when a message encounters a block fault region, an alternative non-minimal path is followed by the message. The alternative path should not introduce new dependencies in the routing algorithm. A solution to avoid deadlocks in 2-D meshes was introduced by Chien and Kim [7] and is to use planar adaptive routing, which adds an additional virtual channel. The technique can also be applied to multidimensional meshes and tori. In the case of torus topologies, this solution requires six virtual channels for each physical link. Moreover, it presents problems when fault regions are adjacent or include nodes on the mesh boundary. Both situations are equivalent to the occurrence of a concave fault region. Ensuring that fault region remains convex will require marking more healthy nodes as faulty, which is not desirable.

One solution to reduce the number of healthy nodes that are included in the fault region is based on the idea of fault rings. A fault ring is the sequence of links and nodes that are adjacent to and surround a fault region. If the fault region is rectangular, then the fault ring is also rectangular. If a fault region includes boundary nodes, then a fault ring cannot be defined and a fault chain is used. This fault rings and fault chains are used to misroute messages that are blocked due to faults in their paths. Meshes with non-overlapping fault rings and with no fault chains require two virtual channels to avoid deadlocks. If there are overlapped fault regions or fault chains then four virtual channels are required.

Some fault-tolerant routing algorithms are able to work with non-convex fault regions. In particular, some of them assume that faults are covered by solid fault regions. A solid fault region in an  $n$ -dimensional mesh is a region where any 2-D cross section of the fault region produces a rectangular fault region. In this case, the concept of fault ring is used to route messages around

the fault region. In this case, four virtual channels are required to ensure deadlock-freedom.

In [15], the authors propose a new fault-tolerant routing methodology based on the use of intermediate nodes. In order to avoid faults, for some source-destination pairs, packets are first sent to an intermediate node and then from this node to the destination node. Deadlocks are avoided by moving packets to a different virtual channel when crossing the intermediate node. Fully adaptive routing is used along both subpaths. In order to increase the level of fault-tolerance intermediate nodes are used together with disabling adaptive routing and/or using misrouting on a per-packet basis. The methodology uses a static fault model but tolerates a reasonably large number of faults without disabling any healthy node and only requires an additional virtual channel.

The preceding fault-tolerant routing algorithms rely on the use of virtual channels; however, virtual channels affect the speed and complexity of the switches. Software-based fault-tolerant routing is an approach that is not based on the use of virtual channels. When a packet encounters a fault, it is ejected from the network and is later injected and routed through an alternative path. This mechanism is very flexible and supports many fault patterns, without requiring assuming healthy nodes as faulty nor requiring additional virtual channels. However, some packets may suffer high latencies due to the packet ejection and reinjection, and memory bandwidth at the node where the injection/reinjection is performed is consumed.

## Fault-Tolerant Routing in Indirect Networks

The fault-tolerant proposals can be classified depending on whether they are applicable to unidirectional or bidirectional MINs. In UMINs, if they use the minimal number of stages, there is a unique path for each source-destination pair. In BMINs, the topology provides several paths for each source-destination pair. In particular in BMINs topologies, if the number of faulty components is smaller than the arity of the switches, then the topology provides at least a path between each source-destination pair, and therefore the routing algorithm may communicate any pair of nodes. This is true except for switch and link failures of the first and last stages, where the nodes directly

attached to the faulty components get disconnected from the rest of the network. So defining deadlock-free fault-tolerant routing algorithms is easier in BMINs topologies.

Some fault-tolerant routing algorithms for UMINs are based on misrouting packets by routing them in multiple passes [8, 9]. The packets that would cross the faulty elements are routed through other nodes of the network. In this way packets can avoid the faults. The drawback of using such technique is that these algorithms rely on a static fault model, inject the packet into the network several times, provide non-minimal paths, and increase the average packet latency. This technique can also be applied in BMINs.

In MINs, there are also proposals of disabling healthy nodes or network elements with poor network connectivity with the aim of making the routing easier after the occurrence of the faults. This approach is followed in [10] where the authors propose to create a new, but equivalent, topology from the faulty one, thus preserving the routing algorithm at the cost of disabling some healthy nodes and network elements.

The previous proposals are based on a static fault-model. But there are other proposals that use a dynamic fault model. In particular, some proposals exploit the high number of alternative paths of BMINs to provide fault-tolerance, such as the proposal made in [11, 12]. In these papers, when a fault is detected, the network reconfigures the routing information at the switches or nodes, and paths that would use any of the faulty network elements are disabled. This provides a good fault-tolerance at almost no overhead in cost. There are several works that mix this proposal with replication of links to increase the number of routing options, increasing the cost of the network.

A more recent technique is based on misrouting the packets when they encounter a fault without ejecting the packet from the network [13] and without using additional hardware. The authors provide a wide analysis to ensure that the misrouted packets cannot form a deadlock.

## Related Entries

- [Flow Control](#)
- [Interconnection Networks](#)
- [Networks, Direct](#)
- [Networks, Multistage](#)

- [Routing \(Including Deadlock Avoidance\)](#)
- [Switch Architecture](#)
- [Switching Techniques](#)

## Bibliographic Notes and Further Readings

The main bibliographic source for the elaboration of this entry has been the book [2]. Most of the discussion about fault-tolerant routing in direct networks is inspired by this book. Other two main sources have been the appendix [4] and the book [3]. Both of them have been especially useful for writing the entry beginning.

The references [5–7] are basic papers for fault-tolerant routing in direct regular networks and references [8–13] are basic papers for fault-tolerance in indirect regular networks.

## Bibliography

1. <http://www.top500.org/>
2. Duato J, Yalamanchili S, Ni L (2003) Interconnection networks: An engineering approach. Morgan kaufmann Publishers, San Francisco, California
3. Dally W, Towles B (2004) Principles and practices of interconnection networks. Morgan kaufmann Publishers, San Francisco, California
4. Pinkston T, Duato J Appendix E: Interconnection networks. Computer Architecture: A Quantitative Approach
5. Chien M, Shin K (1990) Adaptive fault-tolerant routing in hypercube multicomputers. IEEE Trans Parallel Distrib Syst 53(12):1406–1416, December 1990
6. Lee T, Hayes J (1992) A fault-tolerant communication scheme for hypercube computers. IEEE Trans Comput C-41(10):1242–1256, October 1992
7. Chien A, Kim J (1992) Planar-adaptive routing: Low-cost adaptive networks for multiprocessors. In: Proceedings of the 19th international symposium on computer architecture. pp 267–277, May 1992
8. Karlin A, Nelson G, Tamaki H (1994) On the fault tolerance of the butterfly. In: Proceedings of the 26th ACM symposium on theory of computing. pp 125–133
9. Shen J, Hayes J (1984) Fault-tolerance of dynamic-full-access interconnection networks. IEEE Trans Comput 33(3):241–248
10. Leighton F, Maggs B, Sitamaran R (1995) On the fault tolerance of some popular bounded-degree networks. SIAM J Comput 542–552
11. DeHon A, Knight T, Minsky H (1991) Fault-tolerant design for multistage routing networks. In: Proceedings of the international symposium on shared memory multiprocessing. April 1991
12. Gómez C, Gómez M, López P, Duato J (2009) FT2EI: a dynamic fault-tolerant routing methodology for fat trees with

- exclusion intervals. *IEEE Trans Parallel Distrib Syst* 20(6): 802–817
13. Sem-Jacobsen F, Skeie T, Lysne O, Duato J (2006) Dynamic fault tolerance with misrouting in fat trees. In: Proceedings of the international conference on parallel processing. pp 33–44
  14. Sem-Jacobsen F, Skeie T, Lysne O, Torudbakken O, Rongved E, Johnsen B (2005) Siamese-twin: a dynamically fault-tolerant fat-tree. In: Proceedings of the international parallel and distributed processing symposium. Denver, Colorado
  15. Gómez ME, Nordbotten N, Flich J, Lopez P, Robles A, Duato J, Skeie T, Lysne O (2006) A routing methodology for achieving fault tolerance in direct networks. *IEEE Trans Comput* 55(4): 400–415

## Networks, Multistage

OLAV LYSNE, FRANK OLAF SEM-JACOBSEN  
The University of Oslo, Oslo, Norway

### Synonyms

[Butterfly](#); [Crossbar](#); [Distributed switched networks](#); [Fat tree](#); [Interconnection networks](#); [k-ary n-fly](#); [k-ary n-tree](#); [MIN](#); [Multistage interconnection networks](#); [Router-based networks](#)

### Definition

A multistage network is a network for interconnecting a set of nodes through a switching fabric. These nodes can either be programmable computers or memory blocks. The switching fabric consists of a set of switches interconnected to form a *topology* with defined connection points for the nodes. The switches are organized in stages. A switch may have zero, one or more connection points, thus differentiating *multistage* networks from *direct* networks where every switch must have at least one connection point and *indirect* networks in general which have a less strict switch organization.

### Discussion

#### Introduction

Multistage networks are an approximation to the ideal network in terms of *bisectional bandwidth*, communication latency, and the number of concurrently communicating pairs in the network. In order to best understand this it is helpful to view the multistage networks in terms of a telephone exchange system, which is where the concept first emerged.

An old-fashioned telephone switchboard had a number of input channels, output channels, and internal switching points (the leads used to connect to telephone lines). As long as the switchboard had enough internal switching points it was possible to connect any two free phone lines together and allow them to communicate. In that case the switchboard was *nonblocking*. However, without enough leads to connect to telephone lines two parties wishing to initiate a phone call might have had to wait until one of these leads were available before the call could be initiated. In that case the switchboard was *blocking*.

As the telephone network grew a single switchboard was not sufficient to connect all the endpoints without becoming prohibitively large. The solution was to create a network of the switchboards so that a telephone call might have to traverse multiple switchboards.

As the size of the switchboards and the number of switchboards increased the challenge became to maintain the nonblocking capabilities of the network to as large an extent as possible. This led to the advent of multistage networks (clos networks) which will be discussed below.

The concepts from the telephone network can be more or less directly transferred to computer networks, with the important addition of *packet switching* in addition to the *line/circuit switching* used by the telephone network. The rest of this chapter focuses on packet switched multistage networks, as they are the ones most commonly found in current computer systems.

Multistage networks are commonly found in tightly coupled networks such as networks used in computer clusters, within switches, or in single chips. Here the nonblocking concept is often modified to *statistically nonblocking* as discussed further down. It is in these domains that multistage networks have been most extensively studied.

#### Topologies

As for direct networks, the topology of a multistage network can be modelled as a graph  $G(V, C)$  where  $V$  is the set of switches and  $C$  is the set of links (unidirectional or bidirectional) interconnecting the switches. The processing nodes are usually not included in the model. In addition, a regular multistage network can be modelled as an  $N \times M$  network with  $N$  inputs and  $M$  outputs. It is not uncommon that  $M$  equals  $N$ .

Depending on the size of  $N$  and  $M$  there are practical limitations on how these can be interconnected in a topology. The ideal topology is the crossbar (discussed below), and for small  $N$  and  $M$  this is indeed a practical implementation. However, as  $N$  and  $M$  increase, implementing a crossbar becomes impractical. A common solution to this problem is to implement the network in multiple stages as a multistage network.

### Crossbar

The crossbar is perhaps the most ideal network topology. Given  $N$  nodes to be interconnected, an  $N \times N$  crossbar allows  $N$  one-to-one interconnections without contention, i.e., it is nonblocking. A node can send data to any other node that is not already receiving data without having to wait for capacity in the crossbar. This lack of contention allows the connection points in the crossbar to be implemented as very simple arbiters. The crossbar has very high bisectional bandwidth that scales linearly with the size of the crossbar, and because of its simplicity the time used to cross it is very low. A simple crossbar is visible in Fig. 1.

The high degree of connectivity of the crossbar makes it a quite expensive topology to implement. The cost of it is  $O(N^2)$ , indicating that it does not scale well beyond small sizes. Because of this, the crossbar is often used to interconnect the input and output ports of a switch itself, and the switches are again interconnected

to create larger networks to overcome the scaling problem of the crossbar. The largest single-chip crossbar switches in commercial production at the time of writing typically have around 36 ports.

### Multistage Networks

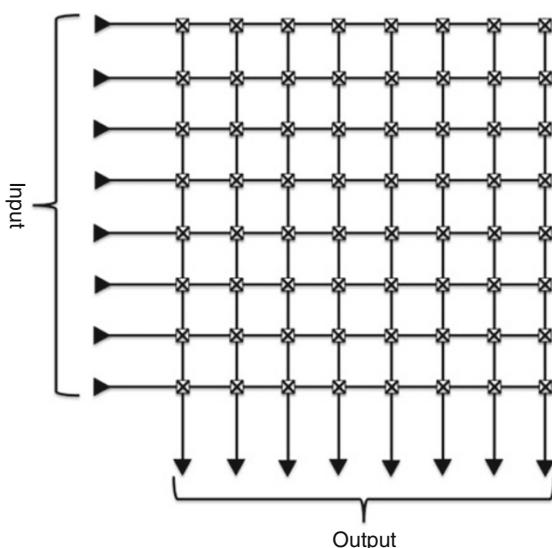
In order to solve the problem of scalability of the crossbar and still maintain its nonblocking properties, Charles Clos introduced nonblocking switching networks in 1952. While the work was originally intended for the telephone networks, the concept of nonblocking switching networks has, together with the crossbar, been transferred to the packet-switched computer networks domain. The types of networks he introduced are often called clos networks.

The basic idea behind clos networks is to connect switches in an odd number of switching stages, with one or more switches at each stage. This idea has been greatly extended since its inception, resulting in a class of networks called Multistage Networks. The common way of denoting a multistage network is by giving the number of stages in the network, the sizes of the switches that make up the stages, and the permutations that determine the interconnections between the stages.

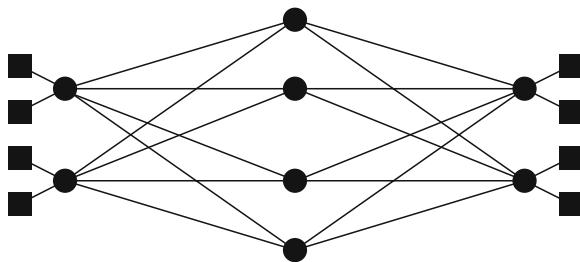
Going back to the original work by Clos, the minimum number of stages for a clos network is three; an input stage, an output stage, and a middle stage. The number of switches and the number of ports on each switch for the three different stages determines the number of nodes that can be connected and the communication properties of the network.

For the three stage networks, every switch at the input side of the multistage network has  $n_1$  input ports and  $m$  output ports. There are  $m$  middle stage switches with  $r_1$  input ports and  $r_2$  output ports. Finally, the output stage consists of  $r_2$  switches with  $m$  input ports and  $n_2$  output ports. It is the relationship between these variables that determines the blocking characteristics of the multistage network as discussed in the next section. Figure 2 shows a three-stage clos network with four middle-stage switches and  $n_1 = 2, m = 4, r_1 = r_2 = 2$ , and  $n_2 = 2$ .

The permutation defines how the switches are interconnected between the stages. This affects scheduling of the packets to cross the topology, and also any degree of redundancy that may be available for use for fault tolerance.



**Networks, Multistage.** Fig. 1 An 8 × 8 crossbar



**Networks, Multistage.** Fig. 2 A three stage clos network

For a fixed switch size (e.g.,  $2 \times 2$ ) increasing the number of inputs and outputs supported by the topology is achieved by increasing the size of the middle-stage switches. By recursively replacing the middle-stage switches with three-stage clos networks based on smaller switches, the large switches can be replaced by  $2 \times 2$  switches at the cost of increasing the number of stages in the network. Even though the number of stages in the network is increased this makes sense from a cost perspective because  $2 \times 2$  switches are quite inexpensive and fast. However, the overall latency of the network will in many cases increase.

### Blocking Characteristics

The switch sizes, the number of stages, and the number of switches at the middle stage also determine the blocking characteristics of the MIN. Pure clos networks are *strictly nonblocking*, i.e., regardless of any pre-existing paths set up through the network, it is always possible to find a path between two idle connection points. In terms of the parameters defined above it is required for the network to be strictly nonblocking that  $m \geq 2n - 1, n = n_1 = n_2$ . As the size (in terms of switches and switch sizes) of the middle stage decreases, the degree of blocking in the network increases. Some MINs are *rearrangeable* (e.g., Benes networks) where it is always possible to set up a connecting path between two idle nodes by moving pre-existing paths around in the network.

Other blocking characteristics also exist based on how and when pre-existing paths through the topology are moved around to accommodate further communication pairs.

### Unidirectional MINs

Multistage networks can either be unidirectional or bidirectional. This section considers the unidirectional topologies.

A unidirectional MIN consists of a series of two or more switching stages where packets always move from an injection point at the left side to an ejection point at the right side. To allow bidirectional communication between nodes each injection and ejection pairs are connected to a single node, usually by a short link to the ingress and a long link to the egress. MINs are usually designed in such a way that it is possible to move from any injection point to any ejection point in a single pass through the network.

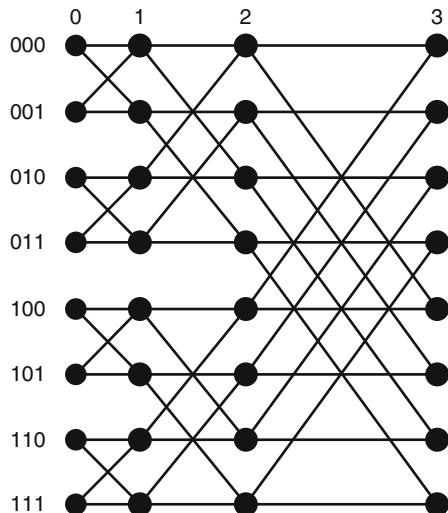
A large number of connectivity permutations (different ways of interconnecting switches at neighboring stages) have been proposed for unidirectional MINs. This includes the shuffle-exchange, the Butterfly, Baseline, Cube, Omega, and the Flip to name a few. All of the topologies mentioned here are isomorphic, or topologically equivalent. This means that any one topology can be converted to any of the other topologies simply by moving the switching elements and the endpoints up and down within their stages.

Aside from these a large number of other unidirectional MIN topologies also exist, such as the Benes network, which provides more than one path between each pair of nodes.

Each of the existing topologies represents a trade-off between implementation complexity, blocking versus nonblocking, the ability to route from a single source to multiple destinations in one pass through the topology, partition ability, fault tolerance, and the number of switching stages required to realize these properties.

An important characteristic of unidirectional multistage networks is that the path length to any destination is constant. This is an important characteristic with regards to scheduling packets as it helps to make the network very predictable.

**Butterfly** The Butterfly, or  $k$ -ary  $n$ -fly, is one of the most commonly mentioned topologies when discussing multistage networks.  $k$  and  $n$  represent the number of ports in each direction of every switch and the number of switching stages, respectively. The connection between any two switches at two neighbouring stages is defined



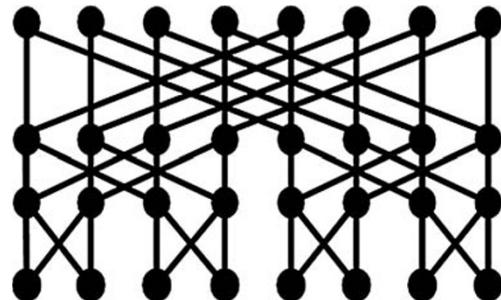
**Networks, Multistage. Fig. 3** A four stage butterfly

by the following function. The interconnection pattern is illustrated in [Fig. 3](#).

Let every outgoing switch port be identified by a  $n$ -digit number (tuple) where each digit  $x$  can have the values from 0 to  $k - 1$ ,  $\{x_{n-1}, x_{n-2}, \dots, x_0\}$ .  $\{x_{n-1}, x_{n-2}, \dots, x_1\}$  corresponds to the switch number and  $x_0$  is the port number on that switch. The connection from a port  $a_{i-1}$  at stage  $i-1$  (enumerated from 0 on the left) to a switch  $b_i = \{b_{n-1}, b_{n-2}, \dots, b_1\}$  at stage  $i$  is given by exchanging the first digit ( $x_0$ ) with the  $i$ 'th digit  $x_i$ . For instance, port 000, which is port 0 on switch 00 at stage 0, is connected to switch 00 at stage 1. Port 001, which is port 1 on switch 00 at stage 0, is connected to switch 01 at stage 1, and so on. In this manner, a Butterfly network of any size may be constructed.

### Bidirectional MINs

Unidirectional MINs are a very old development, and most modern-day systems use bidirectional channels. The bidirectional channels can be used to construct bidirectional multistage networks. These topologies have usually been derived from unidirectional MINs by “folding.” In simple terms, all end nodes are connected to only one side of the topology. In order to reach any other end node packets is moved to the right in the network until they reach a switch that is also reachable by the destination node. At this point the direction



**Networks, Multistage. Fig. 4** A common fat tree, a 2-ary 4-tree

reverses and the packet travels to the left, towards its destination.

The consequence is that, contrary to unidirectional MINs where all path lengths were equal, the path length from a specific source to different destinations may be different. In fact, the destinations can be grouped by locality with respect to a specific source, yielding a hierarchical representation of the distance of the destinations.

**Fat Tree** The most commonly used class of multistage networks are fat trees. The fat tree was introduced as a topology by C. Leiserson in 1985. In its original inception, the fat tree was presented as a tree topology with a single root (a tree topology is a multistage network). What made the topology unique was that the aggregate bandwidth of all links between any two tiers in the tree was constant. This was achieved through simply increasing the link bandwidth at every tier.

Increasing the link bandwidth and switch capacity is not a viable approach for large bandwidth systems, so fat trees are in practice most often implemented as folded clos topologies. Contrary to unidirectional networks, fat trees/folded clos networks are most often represented in a vertical manner instead of horizontal, with the end nodes connected to the bottom of the network. Depending on the exact interconnection pattern and the number of switches at the different tiers, the connections of a fat tree are often described as a  $k$ -ary  $n$ -tree or  $m$ -port  $n$ -tree, where  $m$  is the number of switch ports and  $k = m/2$ .  $n$  is the number of tiers in the tree. An example topology is depicted in [Fig. 4](#).

The fat tree is a very popular topology in modern supercomputers because of its universal nature. Averaged over all possible traffic patterns, the fat tree shows the best mean performance. This means that for general purpose computers designed to solve a multitude of different problems, the fat tree is well-suited. There exists, however, a number of specific problem domains such as physical simulations that are better handled by direct networks such as meshes or tori (see the Chap. Direct Networks).

## Routing Functions

Many of the multistage networks are *self routing*. This means that the path through the network to the destination is completely determined by the destination address. At each stage the output port is determined by the corresponding tuple in the destination address. This allows for very simple routing logic and helps to maintain the high performance of the networks.

Many direct networks must follow routing restrictions in order to guarantee deadlock freedom. This means that it is difficult to achieve the maximum theoretical performance from the topology. On the other hand, all the types of MINs described here support minimal deadlock-free routing without any limitations. This means that as long as a packet moves towards the destination, any output port can in principle be selected without risking deadlock.

For modern large-scale implementations, it is common to use routing tables since the systems often are built using commodity hardware that are routing table-based. So, even though any path is deadlock free, the exact path chosen for a given destination is determined by a specific *routing algorithm* in use, whether it is *adaptive* or *deterministic*, and in the latter case, the deterministic path that has been set up for the destination.

## Multistage Networks Today

Multistage networks are a very old research topic. Unidirectional MINs have been extensively studied in order to create topologies that support all desired properties for any given application. This interest has abated significantly in the last decade, mainly because the industrial forces have not shown enough interest in the developments in this field.

In contrast to this, the interest for bidirectional MINs, and specifically the fat tree, has grown in recent years. This is especially evident when looking at the list of the 500 fastest supercomputers in the world. The systems that are not specifically tailored towards applications that require tori-like structures are typically fat trees. This has led to a sustained research interest into how to most efficiently route traffic through the fat tree in a manner that achieves good balancing for a wide variety of traffic patterns.

Another challenge that is actively researched is the best way to efficiently utilize the large degree of redundancy in fat trees. This may be exploited either for increased performance through multipath routing or to achieve fault tolerance.

## Future Directions

Fat trees will continue to be an important topology for high-performance computing. As the number of ports supported by commodity off-the-shelf switching hardware increases, it becomes possible to build larger and larger fat trees without increasing the number of tiers of the topology. Alternatively, the increased switch sizes can be exploited to keep the same topology size and to reduce the number of tiers. This is equivalent to a reduction in latency, which is an important parameter in high-performance computing.

Another area where fat trees lend themselves well is in hierarchical communications schemes. A typical example of this is memory systems where a computing core is connected to different levels of cache. Each cache level is larger than the previous and is located further away. At the very end there are typically the RAM chips. By connecting the processing cores to one subset of the tree, and the memory modules to other subsets of the tree that are sequentially further and further away, the hierarchical nature of the memory is replicated to some degree. As the processing scale decreases and more and more memory fits onto a single chip, there might be a need for such approaches to allow scalability and flexibility in the memory management.

## Related Entries

- ▶ [Buses and Crossbars](#)
- ▶ [InfiniBand](#)
- ▶ [Myrinet](#)

- ▶ Networks, Direct
- ▶ Networks, Fault-Tolerant
- ▶ Network Interfaces
- ▶ Quadrics
- ▶ Routing (Including Deadlock Avoidance)
- ▶ SCI (Scalable Coherent Interface)
- ▶ Switch Architecture
- ▶ Switching Techniques

## Bibliographic Notes and Further Reading

The first presentation of multistage networks was done by Charles Clos [1]. There exists a huge number of publications on typical multistage networks, and a simple overview of the characteristics of clos networks is given in [4]. Other good introductions to the topic of multistage networks can be found in the books by Duato et al. [3] and Dally et al. [2].

The seminal paper on fat trees was published by Charles Leiserson [5]. This also spawned an extensive amount of work focusing on both reducing the redundancy to achieve cheap topologies with the same characteristics [9], and on utilizing the redundancy for fault tolerance [8] and high-performance. The fat tree structure is originally quite strict. This has been extended [10] to support a more flexible structure. This is important since most actual implementation of this topology deviates from the strict fat tree definition. The common multiple root definitions of fat trees are  $k$ -ary  $n$ -trees [7] and m-port n-trees [6].

## Bibliography

1. Clos C (1953) A study of non-blocking switching networks. Bell Syst Tech J 32:406–424
2. Dally W, Towles B (2003) Principles and practices of interconnection networks. Morgan Kaufmann, San Francisco
3. Duato J, Yalamanchili S, Ni L (2003) Interconnection networks: an engineering approach. Morgan Kaufmann, San Francisco
4. Jajszczyk A (2003) Nonblocking, repackable, and rearrangeable clos networks: 50 years of the theory evolution IEEE Commun Mag 41(10):28–33
5. Leiserson CE (1985) Fat-trees: universal networks for hardware-efficient supercomputing. IEEE Trans Comput C-34: 892–901
6. Lin X, Chung Y, Huang T (2004) A multiple LID routing scheme for fat-tree-based infiniband networks. In: Proceedings of IEEE international parallel and distributed processing symposiums

7. Petrini F, Vanneschi M (1995) K-ary N-trees: high performance networks for massively parallel architectures. Retrieved from cite-seer.ist.psu.edu/petrini97kary.html
8. Valerio M, Moser LE, Melliar-Smith PM (1995) Fault-tolerant orthogonal fat-trees as interconnection networks. In: Proceedings 1st international conference on algorithms and architectures for parallel processing, vol 2, pp 749–754
9. Valerio M, Moser LE, Smith PM (1994) Recursively scalable fat-trees as interconnection networks. In: Proceeding of 13th IEEE annual international phoenix conference on computers and communications
10. Øhring SR, Ibel M, Das SK, Kumar MJ (1995) On Generalised Fat-trees. In: Proceedings of the 9th international symposium on parallel processing, p 39

## NI (Network Interface)

- ▶ Network Interfaces

## NIC (Network Interface Controller or Network Interface Card)

- ▶ Network Interfaces

## Node Allocation

- ▶ Job Scheduling

## Non-Blocking Algorithms

DANNY HENDLER

Ben-Gurion University of the Negev, Beer-Sheva, Israel

## Synonyms

- Lock-free algorithms

## Definition

Non-blocking algorithms are shared-memory multiprocessor algorithms that can avoid race conditions and guarantee correctness without using mutual exclusion

locks. They are in general more resilient to asynchronous conditions than lock-based algorithms since they guarantee progress even if some of the threads participating in the algorithm are delayed for a long duration or even fail-stop. (The definition of the terms “non-blocking” and “lock-free” has changed in recent years, creating some terminology confusion. In the past, an algorithm was called “non-blocking” if it guaranteed global progress and “lock-free” if it did not rely on mutual exclusion locks. In recent years, most authors call an algorithm “lock-free” if it guarantees global progress and “non-blocking” if the delay of some threads cannot delay other threads indefinitely. This entry adopts the latter terminology since it is more widely used nowadays. Precise definitions are provided later in the sequel.)

## Discussion

### Introduction

To ensure correctness, shared-memory multiprocessor algorithms (also called *concurrent algorithms*) must employ inter-thread synchronization for precluding concurrency patterns that result in *race conditions* – situations in which correctness depends on the order in which steps by different threads are scheduled.

The traditional approach for inter-thread synchronization is to use *blocking* (lock-based) mechanisms, such as mutex locks, condition variables, and semaphores, for ensuring that conflicting critical sections of code do not execute concurrently. Synchronization based on only a small number of locks (*coarse-grained locks*), each protecting a large portion of data, may restrict parallelism too much and is not scalable. On the other hand, when numerous locks are being used (*fine-grained locks*), it becomes difficult to avoid correctness issues such as deadlocks, convoying, and priority inversion. Moreover, the failure of a thread that holds a lock may prevent other threads from making progress.

Non-blocking algorithms take a different approach and can ensure correctness without using locks. Non-blocking synchronization is more resilient to asynchrony conditions than blocking synchronization since thread delays or failures do not prevent other threads from making progress.

### Progress and Correctness Guarantees for Non-blocking Algorithms

Non-blocking algorithms can be characterized according to the type of progress they ensure. The key progress guarantees provided by non-blocking algorithms are the following:

1. *Wait-freedom* [15] ensures that a thread will always complete its operation in a finite number of its steps regardless of the progress of other threads. Wait-free algorithms thus guarantee the individual progress of any non-failed thread.
2. *Lock-freedom* [15] ensures that *some* thread will complete its operation after a finite number of steps is taken by the participating threads. Lock-free algorithms thus guarantee global progress, but individual threads may starve (they might never complete their operation).
3. *Obstruction-freedom* [21, 22] ensures that any thread that *runs in isolation* (i.e., it is the only thread taking steps) from some point on will complete its operation in a finite number of its steps. If no thread runs solo long enough, then obstruction-free algorithms may suffer from *live-lock*, where threads constantly interfere with one another and none of them completes its operation.

It is easily seen that wait-freedom implies lock-freedom which, in turn, implies obstruction-freedom. The converse implications are false: an obstruction-free algorithm is not necessarily lock-free, and the latter is not necessarily wait-free.

The *correctness* condition of a concurrent data structure specifies how to derive the semantics of concurrent implementations of the data structure from the semantics of the corresponding sequential data structure. This requires to disambiguate the expected results of concurrent operations on the data structure.

Two common ways to do so are *sequential consistency* [25] and *linearizability* [19]. Both require that the values returned by the operations appear to have been returned by a sequential execution of the same operations; sequential consistency only requires this order to be consistent with the order in which each individual thread invokes the operations, while linearizability further requires this order to be consistent with the real-time order of nonoverlapping operations.

The vast majority of the non-blocking synchronization literature considers linearizable (also called *atomic*) implementations.

## Base Objects and Primitive Operations

Concurrent algorithms in general, and non-blocking algorithms in particular, are used to implement *concurrent objects* such as concurrent counters, queues, hash-tables, and so on. Concurrent objects are typically implemented from simpler *base objects*. The simplest base objects in a given system are its memory locations. The system's hardware or operating system provide *primitive operations* that can be applied to memory locations.

The familiar atomic *read* and *write* primitive operations are assumed to be provided by all systems. In the shared-memory nomenclature, a memory location that supports the read and write operations only is called a *register*. It has been shown [15] that registers alone cannot support non-blocking implementations of even the most basic data structures, like queues and stacks. This and other evidence on the relative weakness of reads and writes have led much of the research on non-blocking algorithms to assume the existence of stronger primitive operations. The most notable of these is *compare and swap* (often denoted CAS), which takes three arguments: a memory address, an expected value, and a new value (Fig. 1). If the address stores the expected value, it is replaced with the new value; otherwise it is unchanged. The success or failure of this operation is then reported back to the program. It is crucial that this operation is executed *atomically*; thus an algorithm can read a datum from memory, modify it, and write it back only if no other thread modified it in the meantime. CAS can be used, together with reads and writes, to implement any object in a wait-free manner [15]. It has consequently become a synchronization primitive

```
compare-and-swap(w,expected,new)
 if(*w==expected)
 then
 *w=new,return success
 else
 return failure
fetch-and-add(w,Δ)
 curVal=*w
 *w=curVal+Δ
 return curVal
```

**Non-Blocking Algorithms. Fig. 1** The *compare and swap* and *fetch and add* primitive operations

of choice, and hardware support for it is provided in many multiprocessor architectures.

Another widely available primitive operation is the *fetch and add* primitive (denoted FAA), which takes two arguments: a memory address storing an integer value and an integer  $\Delta$ . The FAA operation atomically increases the integer stored at the argument address by  $\Delta$  and returns its previous value. The pseudo-code of the CAS and FAA primitive operations is shown below.

To exemplify the use of the CAS and FAA operations, consider the implementations of a concurrent *counter* object shown in Fig. 2. A counter supports a single operation *fetch-and-inc*, which atomically increments the counter value by 1 and returns its previous value. The *fetch-and-inc* operation can be trivially implemented in a wait-free manner by applying the FAA operation with  $\Delta = 1$ .

Using CAS, the *fetch-and-inc* operation can be implemented by reading the current counter value (this is the expected value), adding 1, and then using CAS for attempting to update the counter. If no other thread modifies the counter between the read and CAS operations, the CAS will succeed, and the counter will be updated. However, if a concurrent modification does occur, the CAS will fail, and the thread will repeatedly retry the update (by first reading the new value of the counter) until it is successful (if it ever is). This algorithm is not wait-free since other threads may keep incrementing the counter and make a specific thread repeatedly fail. It is lock-free since a failure of a CAS operation implies that a CAS operation by a different thread succeeded between when the expected value was read and the failing CAS was attempted.

N

## Theoretical Foundations

As shown above, the FAA operation can be used to implement a counter in a wait-free manner. Can a wait-free counter be implemented by using only read and write operations? Can the CAS operation be implemented in a wait-free manner by using only read, write, and FAA operations? It turns out that the answer to both these questions is negative. The foundations of the rich theory underlying non-blocking synchronization were laid down in 1991 by a seminal paper of Herlihy [15]. Quoting from [15]: “The fundamental problem of wait-free synchronization can be phrased as follows: given two concurrent objects X and Y, does there exist

```

class Counter {
 private int val=0
 int fetch-and-inc()
 return fetch-and-add(&val,1)
 }
}

class Counter {
 private int val=0,exp
 int fetch-and-inc()
 do
 {exp=val}
 until CAS(&val,exp,exp+1)=success
 }

```

**Non-Blocking Algorithms.** Fig. 2 Implementing a counter object from *fetch and add* and from *compare and swap*

```

class Consensus {//Code for thread $i \in \{0,1\}$
 private Counter c initially 0
 private register vals[0..1]
 boolean decide(Vv){
 vals[i]=v
 if c.fetch-and-inc()==0
 return v
 else
 return vals[1-i]
 }
}

```

**Non-Blocking Algorithms.** Fig. 3 A two-thread consensus implementation from counter objects and registers

a wait-free implementation of X by Y?" Herlihy introduced powerful techniques for obtaining the answers to such questions. These techniques are based on reductions to the consensus object.

A *consensus object* supports a single operation called *decide*, which every thread may call with its input at most once. The *decide* operation must satisfy the following requirements.

- *Agreement*: all terminating calls of *decide* (that is, all calls made by threads that do not fail while executing the *decide* operation) must return the same value. In other words, all threads agree on the same value.
- *Validity*: the common decision value is the input of some thread.

A *consensus number* is associated with each object type, specifying the maximum number of threads for which the object type can implement consensus in a wait-free manner. More specifically, if object type X has consensus number  $CN(X)$ , then a wait-free consensus object, shared by  $CN(X)$  or fewer threads, can be implemented by objects of type X and registers; however, a wait-free consensus object that is shared by more than  $CN(X)$  threads cannot be implemented by objects of type X and registers.

For instance, Herlihy proves [15] that the consensus number of a register is 1. This means that registers can (trivially) implement wait-free consensus for only a single thread but cannot be used to implement wait-free consensus for two or more threads. As shown in Fig. 3, a counter object can implement wait-free consensus for two threads. Herlihy proves also [15] that wait-free consensus shared by 3 or more threads cannot be implemented by any number of counter objects and registers. Thus, the consensus number of a counter is 2. It follows that a wait-free implementation of a counter from registers, shared by two threads or more, is impossible.

An object type X is called *universal* if any object can be implemented from instances of X and registers in a wait-free manner. Herlihy proves that an object is universal in a system of  $n$  threads if and only if it has consensus number at least  $n$ . The universality of consensus is established by presenting a *universal construction* algorithm (using consensus objects and registers) that can implement any object in a wait-free manner. Consensus is not the only universal object: Herlihy shows that the CAS object has consensus number  $\infty$ . It follows that CAS can be used, together with read and write, to implement any object (shared by any number of threads) in a wait-free manner. The CAS operation has consequently become a synchronization primitive of choice, and hardware support for it is provided in many multiprocessor architectures.

The concepts of consensus-numbers and universality imply that there is a hierarchy of objects (called the *wait-free hierarchy*), where objects are placed in hierarchy levels according to their consensus numbers, such that (1) an object at level  $l$  cannot implement objects at level  $m > l$ , shared by more than  $l$  threads, in a wait-free manner, and (2) an object at level  $l$  (together with registers) is universal in  $l$ -thread systems.

The table below presents the consensus numbers of several concurrent objects.

| Consensus number | Objects                                                        |
|------------------|----------------------------------------------------------------|
| 1                | Read/write registers                                           |
| 2                | Fetch and add, swap, test and set, stack, queue                |
| :                | :                                                              |
| m                | m-enqueues augmented queue                                     |
| :                | :                                                              |
| $\infty$         | Compare and swap, load-link/store-conditional, augmented queue |

CAS, the *load-link/store-conditional* pair of operations (a weak version of which is implemented on some IBM architectures), and the augmented queue object (a queue that supports a *peek* operation that returns the value of the first queue item, in addition to *enqueue* and *dequeue* operations) all have consensus number  $\infty$  and are therefore universal in any system. It is noteworthy that all levels of the hierarchy are populated (An *m-enqueues augmented queue* is an augmented queue that allows at most  $m$  enqueue operations and starts returning a  $\perp$  response after the  $(m + 1)$ 'st enqueue operation is applied to it [7]).

The impossibility and universality results presented in [15] and summarized above hold also for lock-free implementation but do not hold for obstruction-free implementations: there are obstruction-free implementations of any object from read and write operations only [22]. An objects hierarchy is called “robust” if objects at lower levels cannot be used to implement objects at higher levels. Jayanti [24] showed that the wait-free hierarchy is not robust if consensus numbers are defined so that implementations cannot use registers or can only use a single object of the lower type.

## Algorithms and Design Considerations

Non-blocking synchronization has been the focus of intense research in recent years, and numerous non-blocking algorithms have been published. A few such algorithms are described in the following, mainly for highlighting the design principles underlying them.

A register is called *safe* if a *read* operation that does not overlap a *write* operation returns the value

written by the most recent *write* operation (or the initial value if there are no preceding *write* operations) but may return any value if there is a concurrent *write* operation. A register is *single-writer single-reader* if it can only be used for unidirectional communication between two specific threads, one of which may write the register and the other may read it. A register is *multi-writer multi-reader* if multiple threads may read and write to it. Lamport [26, 27] defined various register types based on the number of threads that are permitted to read and write them and their correctness guarantees. A series of constructions established the rather surprising result that binary safe single-writer single-reader registers can be used to implement the much stronger atomic multi-writer multi-reader registers in a wait-free manner. (They are simply referred to as *atomic registers* in the rest of the text.)

The atomic *snapshot* object [2] allows a thread to take an instantaneous snapshot of an array of registers. It supports two operations: the *update* operation writes a value to the calling thread's register, and the *scan* operation returns an instantaneous snapshot, that is, a state of the array that existed at some point during the execution of the *scan* operation. The ability to obtain an instantaneous snapshot in a wait-free manner can simplify the design of non-blocking algorithms. Indeed, the snapshot object is used as a building block of non-blocking implementations of key objects such as *k-set consensus*, *approximate agreement*, and *renaming* (see [7], Chapter 16).

N

## Helping

A wait-free implementation of the snapshot object from atomic registers is given in [2]. In this implementation, *update* operations help *scan* operations by performing an embedded scan themselves and storing the resulting snapshot value along with the value they write. This is an example of a general design principle underlying many wait-free implementations, called *helping*: operations may need to assist other operations to prevent starvation. A helping mechanism is also used in Herlihy's universal wait-free construction [15], where each thread, before performing its own operation, chooses another thread and checks if it requires help; if it does, it first performs the other thread's operation and only then performs its own operation.

## The ABA Problem and Safe Memory Reclamation

Many non-blocking implementations of concurrent objects based on strong synchronization operations have been presented, including (but not limited to) counters, stacks, queues and double-ended queues, linked lists, sorting networks, hash tables, skip lists, priority queues, and renaming objects (See [18] for a comprehensive discussion of such implementations). Most of these implementations use *unary* atomic operations (i.e., operations that are applied to a single memory location) such as CAS, FAA or *test-and-set*, but some implementations are based on *multi-word* operations such as *double compare and swap*.

Since the CAS operation is available on most modern multiprocessor architectures, and because it can be used to implement any object in a wait-free (hence also lock-free) manner, CAS is used by many non-blocking algorithms. CAS-based algorithms often have to deal with a problem of CAS called the *ABA problem*. The ABA problem occurs when a thread  $t$  reads some value  $A$  from a memory address  $m$ , then some other threads change  $m$ 's value from  $A$  to a different value, say  $B$ , and back again to  $A$ , and finally  $t$  attempts to swap  $m$ 's value from  $A$  to some other value by using CAS; the CAS will then succeed. The correctness of the implementation may be compromised if it depends on the assumption that the CAS must fail if  $m$ 's value changes between the read and the CAS applied to it by  $t$ .

A simple solution to the ABA problem is to associate a reference counter with the application value and to increment it whenever the application value is modified by CAS. This requires packing the application value and reference counter within a single machine word, reducing the number of bits that can be used for storing application values. For some applications, and especially on 64-bit machines, this may be a satisfactory solution; for other applications, however, reducing the domain size of application values is infeasible.

The ABA problem is closely related to the problem of safe memory reclamation. A thread must not recycle (i.e., initialize and then re-use) a dynamic memory structure, even after it was logically removed from the data-structure, while other threads may still be holding references to this structure, which they may later use to access it. Violating this rule may compromise the implementation's correctness. Specifically, unsafe

memory reclamation may result in the ABA problem occurring for CAS operations that attempt to swap a pointer value; in such scenarios, the swap may succeed instead of failing and corrupt the implementation's data-structure.

General schemes for safe non-blocking memory reclamation exist [23, 29]. Michael [29] proposed a wait-free safe memory reclamation technique that uses *hazard pointers*. (The scheme described in [23] uses similar ideas.) Informally, an access to a dynamic memory structure is called *hazardous* if the structure may be concurrently recycled by other threads. With Michael's scheme, each thread owns some (typically small) number of hazard pointers. Before making a hazardous access to structure  $s$ , a thread  $t$  sets one of its currently unused hazard pointers to point to  $s$ ; then,  $t$  verifies (in some application-dependant manner) that  $s$  is still logically part of the algorithm's data-structure. If the verification succeeds,  $t$  can safely access  $s$  as the scheme now guarantees that  $s$  will not be recycled as long as  $t$ 's hazard pointer points to it. If the verification fails, implying that  $s$  is no longer part of the data-structure,  $t$  proceeds to handle this situation in an application-dependant manner. After a dynamic memory structure is logically removed from the data-structure, it is designated as a candidate for memory recycling but will only be recycled when no hazard pointers point to it.

## Memory Contention and Adaptive Algorithms

The number of steps performed by a thread in the course of performing an operation on the implemented data-structure is not the only factor that contributes to the time complexity of non-blocking algorithms. In practice, the performance of non-blocking algorithms is often limited by *memory contention* [12], the extent to which multiple threads access widely-shared memory locations simultaneously. The degradation in performance that is caused by contention is the result of limitations on the bandwidth of both memory and processor-to-memory interconnect.

Important properties of a non-blocking algorithm are therefore (1) the extent by which threads executing the algorithm may encounter memory contention as they execute their steps, and (2) whether the algorithm's step complexity is a function of the total number of processes that *may participate* in the execution or

a function of the number of threads that *actually participate* in the execution (as captured by the memory contention encountered by an operation).

There are several ways to measure contention during the interval of an operation  $op$ . The *total contention* [5] of an execution is the number of threads that ever took steps in the execution. The *interval contention* [3] of  $op$  is the number of threads whose operations are concurrent with  $op$ . The *point contention* [6] of  $op$  is the maximum number of operations *simultaneously* concurrent with  $op$ . Finally, the *step contention* [10] of  $op$  is the number of threads that take steps during  $op$ 's execution interval. An implementation is *adaptive* to (total, interval, point or step) contention if the step complexity of an operation is a function of the contention during the operation interval. By definition, algorithms with adaptive step complexity are wait-free. Yet another measure of contention is *hot-spot contention* [12], defined as the maximum number of threads simultaneously poised to access the same memory location.

Reducing memory contention and designing algorithms that are adaptive to different measures of contention has been the focus of the design of many non-blocking data structures. Some examples include: *Counting networks* [4, 20], which are highly concurrent wait-free implementation of a counter that eliminates sequential bottlenecks and reduce hot-spot contention; adaptive non-blocking implementations of *Collect* [1, 6, 9, 14], and adaptive wait-free implementations of *Renaming* [5].

## Non-blocking Algorithms: Pros and Cons

Non-blocking algorithms are one approach for implementing concurrent data-structures, but alternative approaches exist as well. In this section, the key alternative paradigms for concurrent multi-threaded programming are briefly described and the advantages and disadvantages of the non-blocking approach in comparison with these alternatives are discussed.

The traditional approach for inter-thread synchronization is to use mutual-exclusion locks. Fine-grained lock-based algorithms can be highly efficient but are error-prone and difficult to program. Coarse-grained lock-based algorithms, on the other hand, are much easier to program but, in general, are not scalable.

The key advantage of non-blocking synchronization in comparison to lock-based programming is its

resilience to asynchrony, since, in general, threads executing a non-blocking algorithm can make progress even when other threads are delayed or even fail. On the other hand, the design of wait-free and (to a lesser extent) lock-free algorithms is notoriously hard. Due to their design difficulty, obtaining efficient wait-free or lock-free implementation of basic data-structures is an active research topic in recent years.

Another disadvantage of wait-free and (to a lesser extent) lock-free algorithms is that they often incur high overhead in terms of time-complexity. As shown by Attiya et al. [8], there are problems for which the step-complexity of wait-free implementations is inherently higher than that of non-wait-free implementations. Wait-free and lock-free algorithms often use strong synchronization operations, such as CAS, that are slower than read and write operations, which also hurts their performance.

Designing obstruction-free algorithms, on the other hand, is easier and they have the potential of being faster in the lack of contention. However, obstruction-free algorithms may be susceptible to live-lock scenarios and require contention-management mechanisms for overcoming them.

*Transactional memory* (TM) [16, 31] is an emerging concurrent programming abstraction. Memory transactions allow a thread to execute a sequence of shared memory accesses whose effect is atomic: similarly to database transactions, a TM transaction either has no effect (if it fails) or appears to take effect instantaneously (if it succeeds). Some software transactional memory (STM) implementations are non-blocking, while others are lock-based. A key advantage of the TM programming abstraction is its simplicity: due to the atomicity provided by transactions, programmers can apply sequential reasoning and only need to consider executions that consist of transaction sequences for ensuring correctness.

On the other hand, since TM is a generic mechanism, it is likely that optimized direct non-blocking implementations of specific data-structures will be more efficient than TM-based implementation (assuming both implementations guarantee the same progress and correctness guarantees) but will be much harder to derive.

It is now accepted that no single programming methodology is a panacea for multi-threaded

programming. Instead, multiple approaches must be considered and possibly combined, depending on the application type and the target architecture.

## Related Entries

- [Asynchronous Iterative Algorithms](#)
- [Atomic Operations](#)
- [Processes, Tasks, and Threads](#)
- [Race Conditions](#)
- [Shared-Memory Multiprocessors](#)
- [Synchronization](#)
- [Transactional Memories](#)

## Further Reading

The textbooks by Attiya and Welch [7], Herlihy and Shavit [18], Lynch [28] and Taubenfeld [32] contain a wealth of information about Distributed Computing in general, and the theory and practice of non-blocking algorithms in particular.

Numerous lower bounds and impossibility results for both deterministic and randomized non-blocking algorithms, under various measures, were published over the last 20 years. One of the key developments of this very active area of research is the use of algebraic topology tools for proving impossibility results on the computability of wait-free decision tasks from read and write operations [11, 17, 30]. A detailed description of this research area is out of the scope of this entry, and the interested reader is referred to a comprehensive survey by Fich and Ruppert [13].

## Bibliography

1. Afek Y, De Levie Y (2007) Efficient adaptive collect algorithms. *Distrib Comput* 20(3):221–238
2. Afek Y, Attiya H, Dolev D, Gafni E, Merritt M, Shavit N (1993) Atomic snapshots of shared memory. *J ACM* 40(4):873–890
3. Afek Y, Stupp G, Touitou D (2002) Long-lived adaptive splitter and applications. *Distrib Comput* 15(2):6786
4. Aspnes J, Herlihy M, Shavit N (1994) Counting networks. *J ACM* 41(5):1020–1048
5. Attiya H, Fouren A (2001) Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J Comput* 31(2):642–664
6. Attiya H, Fouren A (2003) Algorithms adaptive to point contention. *J ACM* 50(4):444–468
7. Attiya H, Welch J (2004) Distributed computing: fundamentals, simulations and advanced topics. Wiley, Hoboken
8. Attiya H, Lynch NA, Shavit N (1994) Are wait-free algorithms fast? *J ACM* 41(4):725–763
9. Attiya H, Fouren A, Gafni E (2002) An adaptive collect algorithm with applications. *Distrib Comput* 15(2):87–96
10. Attiya H, Guerraoui R, Hendler D, Kuznetsov P (2009) The complexity of obstruction free implementations. *J ACM* 56(4):1–33
11. Borowsky E, Gafni E (1993) Generalized FLP impossibility result for t-resilient asynchronous computations. In: STOC 1993, San Diego, pp 91–100
12. Dwork C, Herlihy M, Waarts O (1997) Contention in shared memory algorithms. *J ACM* 44(6):779–805
13. Fich F, Ruppert E (2003) Hundreds of impossibility results for distributed computing. *Distrib Comput* 16(2–3):121–163
14. Gafni E, Merritt M, Taubenfeld G (2001) The concurrency hierarchy, and algorithms for unbounded concurrency. In: PODC 2001, Newport, pp 161–169
15. Herlihy M (1991) Wait-free synchronization. *ACM Trans Program Lang Syst* 13(1):124–149
16. Herlihy M, Moss E (1993) Transactional memory: architectural support for lock-free data structures. In: ISCA 1993, San Diego, pp 289–300
17. Herlihy M, Shavit N (1999) The topological structure of asynchronous computability. *J ACM* 46(6):858–923
18. Herlihy M, Shavit N (2008) The art of multiprocessor programming. Morgan Kaufman, Amsterdam
19. Herlihy M, Wing J (1990) Linearizability: a correctness condition for concurrent objects. *ACM Trans Program Lang Syst* 12(3): 463–492
20. Herlihy M, Shavit N, Waarts O (1996) Linearizable counting networks. *Distrib Comput* 9(4):193–203
21. Herlihy M, Luchangco V, Moir M, Scherer III WN (2003) Software transactional memory for dynamic-sized data structures. In: PODC 2003, Boston, pp 92–101
22. Herlihy M, Luchangco V, Moir M (2003) Obstruction-free synchronization: double-ended queues as an example. In: ICDCS 2003, Providence, pp 522–529
23. Herlihy M, Luchangco V, Martin PA, Moir M (2005) Nonblocking memory management support for dynamic-sized data structures. *ACM Trans Comput Syst* 23(2):146–196
24. Jayanti P (1997) Robust wait-free hierarchies. *J ACM* 44(4): 592–614
25. Lamport L (1979) How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Trans Comput* C28(9):690–691
26. Lamport L (1986) On interprocess communication. Part I: basic formalism. *Distrib Comput* 1(2):77–85
27. Lamport L (1986) On interprocess communication. Part II: algorithms. *Distrib Comput* 1(2):86–101
28. Lynch N (1996) Distributed algorithms. Morgan Kaufman, San Francisco
29. Michael M (2004) Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Trans Parallel Distrib Syst* 15(6):491–504
30. Saks ME, Zaharoglou F (2000) Wait-free k-set agreement is impossible: the topology of public knowledge. *SIAM J Comput* 29(5):1449–1483
31. Shavit N, Touitou D (1997) Software transactional memory. *Distrib Comput* 10(2):99–116

32. Taubenfeld G (2006) Synchronization algorithms and concurrent programming. Pearson/Prentice Hall, Harlow

## Nondeterminator

- ▶ [Race Detectors for Cilk and Cilk++ Programs](#)

## Nonuniform Memory Access (NUMA) Machines

- ▶ [NUMA \(Non Uniform Memory Access\) Machines](#)
- ▶ [NUMA Caches](#)

## NOW

- ▶ [Clusters](#)

## NUMA Caches

ALESSANDRO BARDINE, PIERFRANCESCO FOGLIA,  
COSIMO ANTONIO PRETE, MARCO SOLINAS  
Università di Pisa, Pisa, Italy

### Synonyms

[Nonuniform memory access \(NUMA\) machines](#)

### Definition

NUMA is the acronym for Non-Uniform Memory Access. A NUMA cache is a cache memory in which the access time is not uniform but depends on the position of the referred block inside the cache. Among NUMA caches, it is possible to distinguish: (1) the NUCA (Non-Uniform Cache Access) architectures, in which the memory space is deeply sub-banked, and the access latency depends on which sub-bank is accessed; and (2) the shared and distributed cache of a tiled Chip Multiprocessor (CMP), in which the latency depends on which cache slice has to be accessed.

## Discussion

### Introduction

For the past decades, microprocessors' overall performance has been improved thanks to the continuous reduction of transistor size obtained in silicon fabrication technology. This scaling contributed in both (1) allowing designers to put on a chip more and more transistors, and thus to implement on the same die more and more complex microarchitectures, up to arrive to single Chip Multiprocessor (CMP) systems, and (2) increasing processors' clock frequencies.

As the memory bandwidth requirements of cores have increased as well, the increased number of on-chip transistors has also been used to integrate on the same die deeper and larger memory hierarchies. However, reducing feature sizes has also introduced the *wire-delay* problem: sizes of on-chip wires have been also reduced, resulting in larger delay for signals propagation [1]. In particular, considering a huge traditional on-chip cache, the wire delay and the increase of clock frequency lead to an increase of the latency experienced on each access. The bulk of access time involves signals' routing to and from cache banks and not the bank access themselves. Exploiting a non-uniform access time for on-chip caches is a viable solution to the wire-delay problem.

By allowing non-uniform access time also to cache memories, it is possible to design a cache composed by independently accessible banks, connected via a scalable connection fabrics, typically a Network-on-Chip (NoC) [2, 4, 5]. The overall cache access latency is proportional to the physical distance between the requesting unit and the accessed bank. This organization is called NUCA (Fig. 1b and c): banks close to the requesting unit can be accessed faster than a traditional monolithic cache. If the cache management policies succeed in keeping the most performance impacting blocks in such banks, the overall performance will be boosted. NUCA architectures have been proposed for both single-core [2, 3, 6, 7] and multi-core [4, 5, 8–13] environments.

Needs for scalability in CMP designs has lead to the design of CMP tiled systems that are composed by many identical nodes, called tiles. Each tile is composed by one cpu, its private cache levels, and the LLC. Nodes are typically connected through a NoC. In these systems,

the LLCs can be used as a globally shared cache, the LLC inside each node being a slice of the whole cache. Hence, the access time depends on respective position of the requesting node and of the memory slice involved in the access, and this results in a non uniform access time. Those systems results to be implicitly NUMA caches as a consequence of their tiled architecture, differently from NUCA architectures whose design explicitly focuses on the non-uniformity in access time in order to tolerate the wire-delay effects.

### NUMA Caches in the Single-Core Environment: NUCA

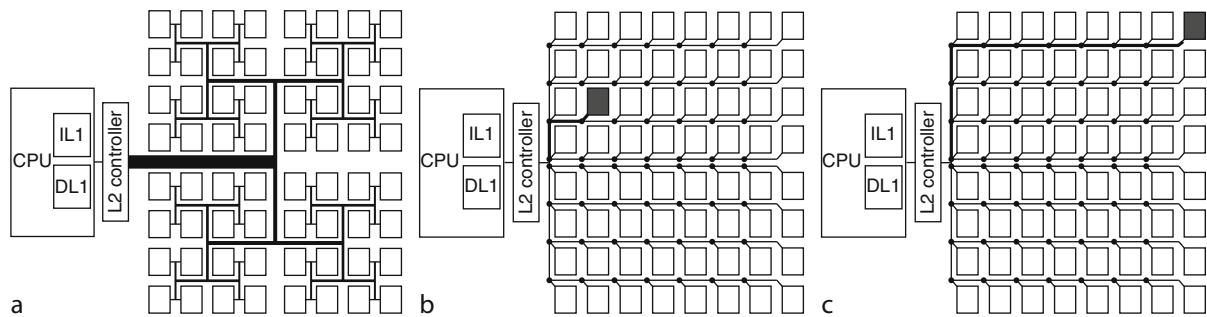
In a traditionally organized cache, each bank is connected to the controller via a fixed length path, usually organized according to the H-Tree model (Fig. 1a). Thus, in a wire delay-dominated context, the access latency of each bank is dominated by the constant length of the path followed by control and data signals, and it is independent from the physical position of the bank. This organization is referred as UCA (Uniform Cache Access) as its access time is *uniform*.

The basic idea of NUCA, first proposed by Kim et al. [2], is to design the cache so that banks are connected among them and with the controller via a connection fabric, typically a NoC. Figure 1b and c show a system equipped with a Level 2 NUCA, partitioned in 128 independent banks, connected with a partial 2-D mesh NoC.

The adoption of a NoC allows to access each bank independently from the others. In particular, in a NUCA, there is a different access latency for each bank depending on its physical distance from the controller. Being in a wire delay-dominated environment, the signal propagation delay dominates overall access latency. Hence banks that are closer to the controller (Fig. 1b) can be accessed faster than the others that are farther (Fig. 1c). As a consequence, the access time is *non-uniform* as it is proportional to the physical distance to be traversed by signals. This architectural design is complemented by data management policies that maintain critical blocks in faster banks. This allows better performance with respect to UCA architectures by obtaining a lower average cache access latency, thus hiding wire-delay effects.

### Data Management Policies: S-NUCA and D-NUCA

When designing a NUCA, one of the main choices is related to the *mapping rule* that dictates which bank can hold a given memory block. The simplest mapping rule is the *static mapping*: a block can exclusively reside in a single predetermined bank, basing on its memory address. A NUCA cache adopting the static mapping is called Static NUCA (S-NUCA). An alternative rule is the *dynamic mapping*: a block can be hosted in a set of banks, called *bankset*, and can be moved inside the bankset. A NUCA cache adopting the dynamic mapping is called Dynamic NUCA (D-NUCA) [2].



**NUMA Caches.** Fig. 1 Examples of systems adopting a traditional sub-banked UCA cache (a) and a NUCA cache (b and c). In all the designs the memory space is partitioned in 64 banks (white squares in the picture). UCA adopts the H-Tree connection model, while NUCA uses a network on chip made up of routers (black circles in the picture) and links. In this way, each bank is independently accessible from the others and the access latency of a block that is in a bank near the controller (b) is lower than access latency that is in a far bank (c), because the network path that must be traversed in order to reach the farther bank is longer

## S-NUCA

In an S-NUCA made up of  $N$  banks, the bank to be accessed is selected basing on the value of  $\log(N)$  bits of the address. Proposed policies for static mapping are *sequential mapping*, which uses the most significant bits of the index field of the physical address, and the *interleaved mapping*, which uses the low-order bits of the same field in order to distribute among different banks blocks that are memory contiguous, thus reducing bank contentions.

When searching for a block, the controller injects into the NoC, a request that will be delivered to the pertaining bank. If the block is present, i.e., the cache access results in a *hit*, the bank sends it to the controller. If the block is not present, i.e., there is a *miss*, the block is retrieved from the main memory, then is stored in the cache bank and, finally, is delivered to satisfy the request.

Despite the simple design and management policies, an S-NUCA usually exhibits lower average access latencies with respect to traditional UCA caches. However, as the data mapping doesn't take into account the data usage frequency, it may happen that more frequently used data are mapped in banks that are far from the controller, thus resulting in higher access latencies and poor performances.

## D-NUCA

In a D-NUCA, banks are grouped into banksets, and each block can be stored in any of the banks that belong

to the pertaining bankset: for example, in the systems of Fig. 1b and c, a bankset could be made up of all banks belonging to the same row. The bankset is chosen basing on the address of the block itself, using  $\log(\text{number of bankset})$  bits of the index field, adopting either the sequential or the interleaved mapping. Blocks can move from one bank to another of the same bankset thanks to the *block migration* mechanism. Migrating most frequently used data from farthest banks to closest banks allows to reduce the average access latency, and thus to achieve better performance with respect to S-NUCA.

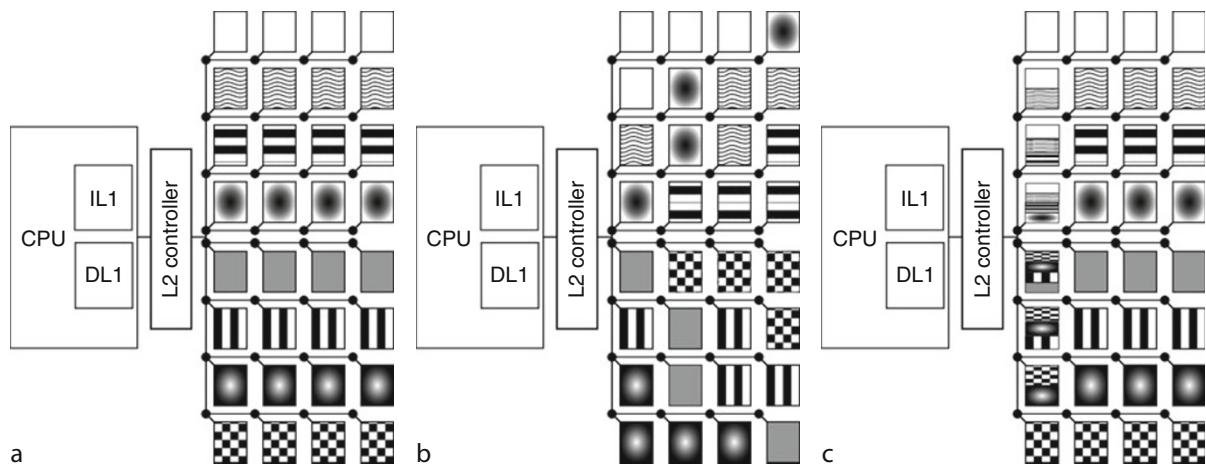
The three main topics in designing a D-NUCA are:

- *Bank Mapping*: how to map banksets to physical banks?
- *Data Search*: how the possible banks are searched to find a block?
- *Movement and replacement*: how and when the data should migrate? In which bank of the bankset a new block should be placed?

## Bank Mapping

The bank mapping policy of a D-NUCA dictates how to group the available physical banks into banksets. Many different policies can be adopted, each offering a different trade-off in terms of implementation complexity and fairness, among the various banksets, of the latency distribution. Proposed mapping policies are [2]: (1) *simple mapping*, (2) *fair mapping*, and (3) *shared mapping*.

The simple mapping (Fig. 2a) maps a bankset to all the banks belonging to the same row. Such policy is



NUMA Caches. Fig. 2 Three examples of bank mapping: simple mapping (a), fair mapping (b) and shared mapping (c)

simple to be implemented, but there are very different access latencies between banksets that are mapped to the central rows of the cache and those that are mapped in the other rows; to access them, a longer network path along the vertical direction must be traversed. In the fair mapping (Fig. 2b), this is avoided at the cost of additional complexity: the banks are allocated to banksets so that the average access times to all the banksets are equalized. In the shared mapping (Fig. 2c), the banksets share the banks closest to the controller so that a fast bank-access is provided to all the sets.

## Data Search

When the search is performed, all the banks of the bankset that can contain a referred block must be searched. The controller injects in the network a request that will be delivered to all the pertaining banks, and the block is searched inside each bank. If all the banks miss the searched block, then there is a *cache miss*, and the block must be retrieved by the next level in the memory hierarchy. Alternatively, one bank contains the searched block, there is a *cache hit*, and the block is sent to the controller.

Examples of proposed search policies for D-NUCA caches are *multicast search* and *incremental search* [2]. In the multicast search, the request is delivered in parallel to all the possible banks, and each of them starts the search as soon as possible. Figure 3 shows an example of memory operation in a D-NUCA adopting the simple mapping scheme and the multicast search. As opposite, in the incremental search, the banks of a set

are sequentially searched from the closest to the farthest. The delivery of the request to each bank is performed only if the previous bank has missed the block. In this way, the number of bank accesses and the associated energy consumption are reduced at the cost of an increase of the total latency.

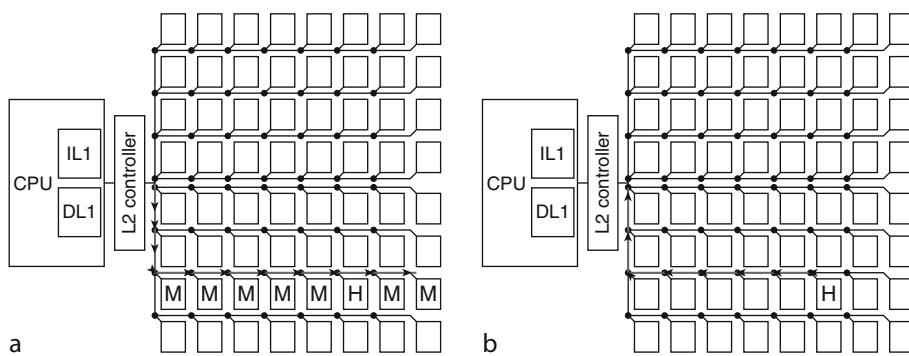
Other solutions have been proposed [5] mixing the two techniques: the banks of one bankset are divided in two or more groups; inside each group the multicast search is adopted, while the various groups are searched incrementally.

## Data Movement and Replacement

Basing on the locality principle, chances there are that, when a block is referred, it will be referred again in a short time. So, when a block is accessed, it makes sense to move it in another bank of the bankset that is closer to the controller. This movement is called *data promotion* [2].

An ideally desirable policy would be to use LRU ordering for the blocks in the banksets, with the closest bank holding the MRU block, second closest holding second most-recently used, etc. Maintaining this ordering would require a heavy block redistribution among banks on each access, with consequent impact on network traffic and energy consumption.

For this reason, more practical low-impact policies are employed usually based on *generational promotions* [2] of blocks basing on their access pattern. For example, in the D-NUCA cache of Fig. 1c on every hit, the accessed block is moved in the left direction by one



**NUMA Caches. Fig. 3** The multicast search operation in a D-NUCA adopting the simple mapping: the request first traverses the vertical links to the pertaining bankset, then traverses the horizontal links and routers where is replicated to reach all the banks of the bankset (**a**). The bank containing the block sends a reply to the controller (**b**) that follows the inverse path of the request message

bank. In this way, the next access to the same block will incur a lower latency. This migration policy is called *per-hit promotion*.

More generally, designing a generational data promotion, in the case of a D-NUCA, requires the definition of three elements: the *promotion trigger* (number of hits after which a promotion must be triggered), the *promotion distance* (number of banks that the promoting block must be advanced), and the *initial placement* of a newly loaded block after a cache miss.

Another important design decision involves what to do with the block that, in the new bank, is eventually occupying the line in which a promoting block must be inserted. Basic choices for this design issue are the *demotion* of the block in the bank previously hosting the promoted block or its *eviction* from the cache. Demotion requires more bank accesses and network traffic than eviction; however, the latter affects performance as it evicts blocks without taking into account their actual usefulness.

Different choices result in different power, network traffic, and performance trade-off; however, the majority of the proposed D-NUCA designs [4–7, 9, 10] adopt the per-hit promotion (i.e., one hit as promotion trigger and one bank as promotion distance) and the insertion of new blocks in the farther from the controller bank. Increasing the value of promotion trigger has been shown to affect performances too much as it limits the promotion of newly loaded data that, basing on the temporal locality principle, are likely to be requested at the next accesses. At the same time, increasing the promotion distance involves the rapid demotion of data that, because of the spatial locality principle, may be needed again in the next future. Using different insertion point for newly loaded data means evicting from the cache blocks that are still useful as they occupy banks that are close to the controller.

## NUMA Caches in the CMP Environment

The increase in the number of available on-chip transistors and the demand for increasing performance have driven the design of CMP systems that include more elaborating cores and a multilevel cache memory hierarchy in a single die. The adoption of a large on-chip multilevel cache allows to guarantee fast memory access to each core.

CMP caches are usually organized in two or more levels in which the Last-Level-Cache (LLC) can be shared among all on-chip cores, while all the other levels are private for each core. As the LLC is usually quite large, it is likely to be affected by the wire-delay problem. For this reason, a CMP system can adopt a shared NUMA cache as its LLC [4, 5, 9, 10].

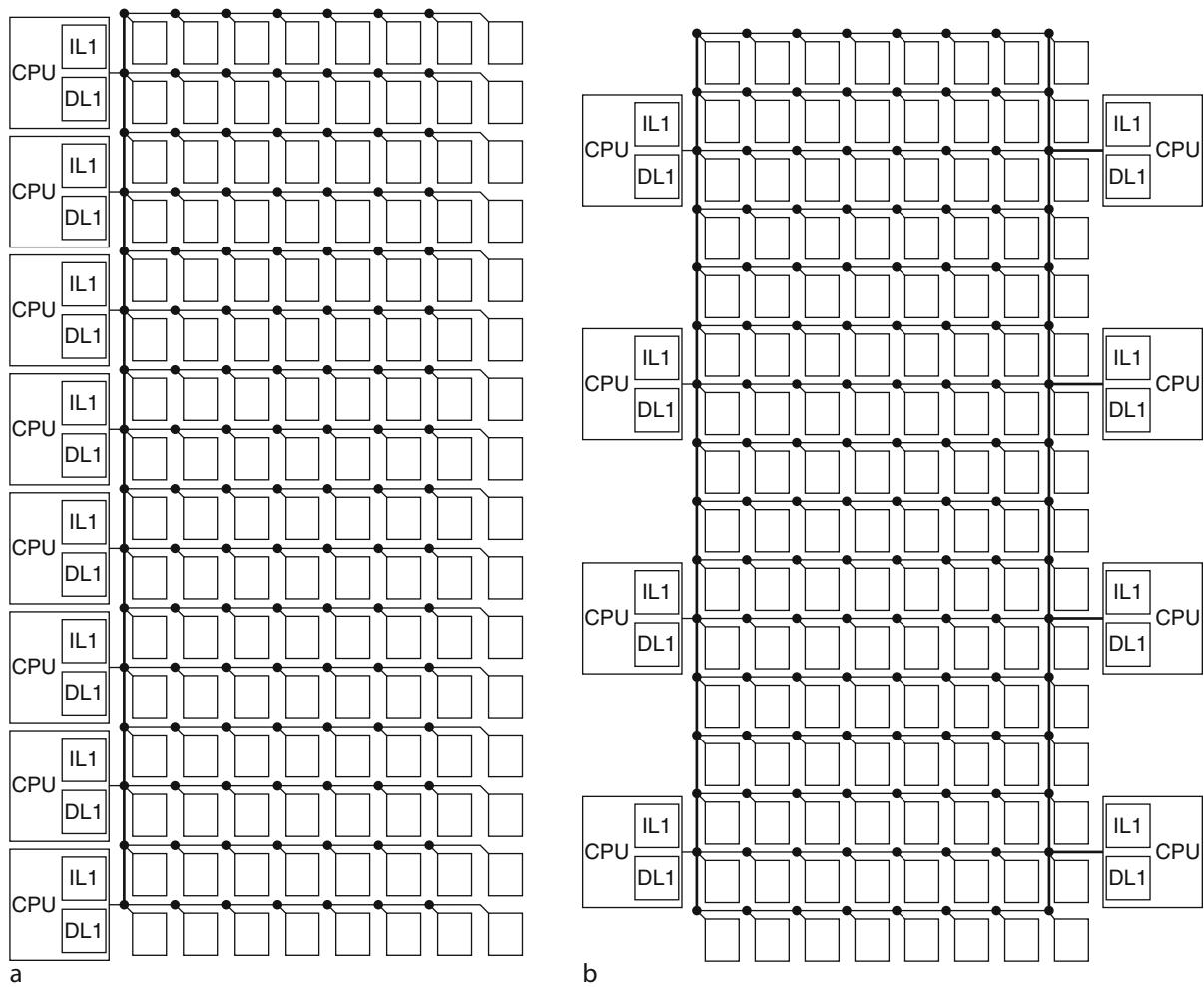
## System Topologies

Taking into account the respective position of cache banks and cpus, CMP systems can be classified in two main topology families: *dancehall* and *tiled*.

In the dancehall, the shared LLC is concentrated in an area of the chip, while cpus stay at one or more sides of the cache. In a dancehall architecture cpus and cache banks can be connected via a NoC (or other connection fabric), resulting in NUCA CMP designs [4, 5, 9, 10, 12] that are analogous to the single core systems previously discussed. Figure 4a shows an example of dancehall CMP in which cpus are plugged to the same side of the NUCA. Variations of this simple configuration have half of the cpus plugged at one side of the shared NUCA, and the others plugged to the opposite side (Fig. 4b), or all the cores distributed along all the sides of the banks matrix [5].

The tiled topology, as shown in Fig. 5, is composed by many identical nodes, called tiles. Each tile is composed by one cpu, its private cache levels, and a LLC that can be used as a private cache or as a local slice of a globally shared LLC [11–13]. In the last case, the access time to the shared LLC depends on the position of both the requesting node and the looked-up LLC slice. In particular, for a core, an access to a remote slice will result in a greater latency, with respect to an access to the local slice. This is due to network paths that depend on the physical distance from the local to the remote tile.

When designing a CMP system, choosing between dancehall and tiled introduces a trade-off in terms of performance and scalability. The three main components of the NUMA access time are: (1) the number of NoC hops to be traversed to reach the hosting bank, (2) the latency of each hop, and (3) the access time to the banks that stores the block. Hence, if the bank to be accessed is small and close to the requesting node, the access latency will be limited. If the interested bank is huge and far from the requesting node, the cost of the cache access will proportionally increase.



**NUMA Caches. Fig. 4** NUCA based CMP systems adopting the *dancehall* topology, with 8 cpus with private L1 caches, and a shared L2 NUCA cache composed by  $16 \times 8$  banks. All the cpus, together with their private caches, can be plugged at the same side of the NUCA **(a)**, or at two opposite sides **(b)**

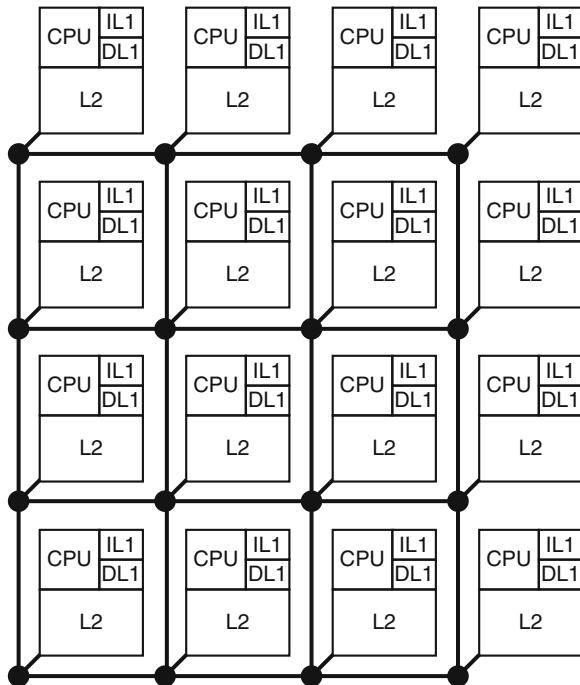
Adopting the *dancehall* topology results in a large number of small banks, each characterized by small response latencies. Moreover, NoC links, being designed to surround banks, are short. Thus they are characterized by a limited link traversal delay. Instead, in the tiled organization, there is a small number of greater and slower banks. Links are longer since they surround the entire tile, and consequently, their traversal delay is higher. Consequently, a request to any non-local slice traverses a small number of higher cost network hops.

Adopting a tiled topology will increase scalability at the cost of high cache access latency due to huge

cache slice and high network hop traversal cost. Instead, adopting a *dancehall* topology minimizes the latency of each hop, as well as the access time to each single bank, but presents a smaller degree of scalability than the tiled case.

### Coherency in CMP Systems Adopting NUMA Caches

In a CMP system, private cache levels must be kept coherent. Given that NUMA LLCs adopt a communication fabric which does not guarantee sequencing of coherence traffic (e.g., the NoC), a directory-based coherence protocol, similarly to the ones proposed for



**NUMA Caches. Fig. 5** A NUCA based CMP system adopting the *tiled* topology, with 16 cpus with private L1 cache, and a shared NUCA cache composed by 16 slices, one for each tile

classical DSM systems, must be adopted. The *directory* is a node of the system that tracks which of the private caches hold a copy of each block. In CMP systems with NUMA LLC, the directory can be *centralized* or *distributed*.

A centralized directory is a monolithic node of the system separated from the remaining nodes. This solution presents some scalability issues because all the nodes must access it to guarantee the correctness of memory operations. When the number of such nodes increases, then directory contentions increase as well, resulting in higher response time and thus affecting performance.

The distributed directory is typically implemented inside each NUMA cache bank. Directory information is stored near the TAG field of each cached block, using some specific status bits. As a consequence, directory accesses are distributed among all banks. When a *miss* in the last private level cache occurs, an unique access is used to retrieve both the block and the directory information.

## Mapping Policies

Also for the NUMA cache in CMP systems, the two possible mapping policies are static and dynamic. Due to the presence of many traffic sources, in both cases the NoC traffic increases proportionally to the number of cores per chip. At the same time, the contention of shared blocks introduces new design issues that are related to both mapping policy and topology.

### Static Mapping

Static mapping has been adopted in both *tiled* [11] and *dancehall* [9, 12] designs. The static mapping associates each memory address to a single cache bank, which is able to satisfy both read/write and coherence requests coming from any of the private next-level cache.

The path followed by request and coherence messages to reach the interested bank depends on the respective position of requestor and bank. If the bank is far from the requestor, the message has to traverse many network hops before reaching the bank. If the bank is close to the requestor, the network path will be short. As the bank to be accessed is chosen basing on the block address, the banks that a processor accesses depend on how the *working set* of the running process/thread is mapped on memory, i.e., which are the physical addresses that are accessed. So, if the running process/thread's *working set* is mapped to remote banks, then the cache access latency will be high; otherwise it will be small. Consequently, both topology aspects and memory mapping rules contribute in affecting or enhancing overall performance.

### Dynamic Mapping

Block migration can be used to improve performance also in CMP systems adopting a NUMA LLC. *Dancehall* [4, 5, 8, 10, 12] and *tiled* [11–13] designs have been proposed in which cache blocks are able to move among banks in order to reduce response latency.

As for the *dancehall* case, the design results in a D-NUCA similar to the single core case. Also in this context, the shared D-NUCA is usually considered as partitioned in many banksets, and cache blocks are able to migrate among banks belonging to the same bankset [4, 5, 9, 10]. For example, in the systems shown in Fig. 4a and b, banksets may be constituted by the rows of the NUCA bank matrix. If D-NUCA succeeds in bringing the most frequently accessed blocks near to the

referring cpu(s), overall performance will be boosted. However, the performance gain that could be obtained with migration comes with some design issues, as in a CMP there are many traffic sources that complicate the managing of the block migration process.

Blocks that are shared by two or more threads can be accessed by different nodes if the sharers are running on different cpus. This is not necessarily a disadvantage but depends on the overall system topology and process scheduling. By considering the system of Fig. 4a, as all the cpus are placed at the same D-NUCA side, all the blocks migrate toward the same direction because the faster way is the same for all cpus. Instead, in the system of Fig. 4b, the faster way for half of the cpus is the slowest for the others, and vice versa. If the sharers are running on cpus plugged at opposite D-NUCA sides, shared blocks will alternatively migrate in both directions, and none of the cpus will succeed in bringing such blocks in the respective faster way. This phenomenon is known as *ping-pong*, and if not properly managed, reduces the performance gain deriving from block migration.

Another design issue of D-NUCA CMP systems regards the *initial placement* of a new block after a cache miss (i.e., choosing which bank of each bankset will store incoming blocks). Considering the *dancehall* configuration of Fig. 4a, choosing the fastest way as the block entry point would interfere with most referred blocks, and similar considerations can be done as in the single core case. Instead, for the system shown in Fig. 4b, the farthest way for half of the cpus is the closest for the other half, so blocks loaded by half of the cpus would interfere with most frequently used blocks of the other half. Consequently, a possible trade-off could be choosing the *initial placement* in the central banks.

Alternative schemes, for both the *dancehall* and *tiled*, have been proposed [11, 12], in which blocks are initially placed near the first requestor and stays there until they are requested by a different cpu. In this case, the blocks migrate toward a bank (or a slice) that is determined according to its memory address basing on a static mapping policy. On subsequent requests, the block is not further moved. Dually, migration schemes have been proposed so that blocks are initially placed basing on their memory address and

subsequently migrated basing on the position of the referring cpu [13].

Dynamic block movement designs must face the *false miss* problem. This is a particular race condition that can occur when a subsequent access to a migrating block arrives when the block is still *on-the-fly*. A block migration involves two banks: the *source* and the *destination*. If the new request is received by the *source* after the block has been sent and by the *destination* before the migrating block has arrived, both the accesses will result in a miss; actually, the block is present in cache and must not be retrieved by the next level memory. This would cause two problems: (1) performance loss, because off-chip accesses are always more expensive than cache accesses; (2) the correctness of memory operations, as there would be two (or more) unmanaged copies of the same block. Techniques for resolving such problem have been proposed, for example, a directory holding all the cached block addresses can be accessed in order to recognize false misses [5], or a specific false miss avoidance algorithm can be designed [10].

## Research Directions

Promising lines of studies for NUMA caches are:

1. Power-consumption saving techniques. While being able to reduce wire-delay effects, NUCA caches still suffer of another typical problem of nanometers technologies: the static power consumptions due to the increase of leakage currents. Besides, due to the additional NoC traffic and bank access, D-NUCA exhibit an increase also in the dynamic power consumption [14]. Even if NUCA caches are made up of traditional cache banks, the direct adoption of well-known techniques like, for example, drowsy memories and decay cache lines, may not be effective particularly in D-NUCA because of the data movement that modify bank access pattern. An alternative proposed technique [16] exploits different usage of banks due to block migration to power-gate farther and less used banks, and has been introduced for both the single-core case and the CMP environment in which all the cpus are connected to the same cache side. However, when cpus are distributed around more NUCA sides due to the problems that arise in D-NUCA based CMP systems, the concept

of farthest banks is ambiguous and has to be redefined; thus the raw application of such technique is no longer possible.

2. Performance improvement via remapping policies. Memory remapping at compile time is traditionally used to improve cache memory performance, in particular for reducing conflict misses. In the context of NUCA caches, similar techniques can be used also for placing data block in banks (or banksets) that reduce their average access latency [16]. Typical samples of used metrics in mapping policies are the data usage frequency and their being shared or private. Remapping can be implemented also via hardware, for example, by designing specific protocols that dynamically change the hosting bank (in the case of S-NUCA, similarly to [12]) or bankset (in the case of D-NUCA).
3. Efficient block migration schemes are an open issue for CMP caches. Generational promotion schemes as in the case of single-core D-NUCA are effective in *dancehall* systems in which cores are all plugged at the same cache side. In all the other cases, such simple schemes are no longer effective due to the *ping-pong* problem that can arise on shared data. Alternative solutions are likely to take into account the position of cores that share a specific block in order to migrate it in a position that optimizes overall performance (similarly to [13]). Moreover, blocks replication could be evaluated as a viable solution to the *ping-pong* problem so that each copy can migrate toward a specific requestor [18]. Such scheme must take care of the need of each application. In fact, while the convenience of replicating Shared-Read-Only Blocks is obvious, there may be classes of applications, in which replicating also Shared-Read-Write blocks can contribute in improving performance.

## Related Entries

- ▶ [Cache Coherence](#)
- ▶ [Interconnection Networks](#)
- ▶ [Memory Models](#)
- ▶ [Memory Wall](#)
- ▶ [Routing \(Including Deadlock Avoidance\)](#)
- ▶ [Shared-Memory Multiprocessors](#)

## Bibliographic Notes and Further Reading

The first NUCA cache architecture was proposed in [2], and demonstrates that a dynamic NUCA structure achieves an IPC 1.5 times higher than a traditional Uniform Cache Architecture (UCA) when maintaining the same size and manufacturing technology. Leveraging on alternatives data mapping policies, NuRapid [4] and Triangular D-NUCA cache [6, 7] have been proposed as alternative designs for NUCA architectures in order to optimize the trade-off between area occupation and performance. An energy model for NUCA architecture is proposed in [14] together with an energy/performance trade-off evaluation for NUCAs and its comparisons with UCA. The Way Adaptable D-NUCA architecture is proposed in [16] shows that it is possible to dynamically adapt the cache size to the application needs, thus consistently reducing NUCA static power consumption while marginally affecting the performances. The impact of NoC elements parameters on NUCA performances have been analyzed in [17]. The same work proposes an alternative design for NUCA caches that relaxes the constraints imposed on network routers latency by clustering multiple banks around each router.

In the context of on-chip multiprocessors, the behavior of D-NUCA as shared L2 caches has been analyzed in [4, 5, 8]; these works evaluate the performance of different data mapping and migration policies. Topology studies for S-NUCA and D-NUCA, related to the dancehall configurations, can be found in [9, 10]. These works also propose a directory coherence protocol specifically suited for CMP NUCA environment and a solution to some design issue of block migration policies in D-NUCA-based CMP systems. The analysis in [11] is based on the tiled architecture and employs an optimized block placement and migration scheme, taking into account both block usage characteristic (Read-Only or Read-Write) and threads migration. In [12] is proposed a scheme called SP-NUCA in which private blocks are dynamically recognized and then stored in S-NUCA banks that are very close to the owner. The system proposed in [13] is based on a tiled architecture and implements a migration mechanism that places blocks in slices depending on the sharers' position.

## Bibliography

1. Matzke D (1997) Will physical scalability sabotage performance gains? *IEEE Comput* 30(9):37–39
2. Kim C, Burger D, Keckler SW (2003) Non uniform cache architectures for wire-delay dominated on-chip caches. *IEEE Micro* 23(6):99–107
3. Chisti Z, Powell MD, Vijaykumar TN (2003) Distance associativity for high-performance energy-efficient non-uniform cache architectures. Proc. 36th int. symp. on microarchitecture. San Diego, CA, pp 55–66
4. Huh J, Kim C, Shafi H, Zhang L, Bourger D, Keckler SW (2005) A NUCA substrate for flexible CMP cache sharing. Proc. of the 19th int. conf. on supercomputing. Cambridge, MA, pp 20–22
5. Beckmann BM, Wood DA (2003) Managing wire delay in large chip-multiprocessors caches. Proc. of 37th int. symp. on microarchitecture. San Diego, CA, pp 55–66
6. Foglia P, Mangano D, Prete CA (2005) A cache design for high performance embedded systems. *J Embedded Comput* 1(4): 587–598
7. Foglia P, Mangano D, Prete CA (2005) A NUCA model for embedded systems cache design. IEEE 2005 workshop on embedded systems for real-time multimedia (ESTIMEDIA). New York Metropolitan Area, USA, pp 41–46
8. Chisti Z, Powell MD, Vijaykumar TN (2005) Optimizing replication, communication, and capacity allocation in CMPs. Proc. of the 32nd int. symp. on computer architecture. Madison
9. Foglia P, Panicucci F, Prete CA, Solinas M (2009) An evaluation of behaviors of S-NUCA CMPs running scientific workload. Proc. of the 12th euromicro conference on digital system design (DSD). Patras, Greece, pp 26–33
10. Foglia P, Panicucci F, Prete CA, Solinas M (2009) Analysis of performance dependencies in NUCA-based CMP systems. Proc. of the 21st international symposium on computer architecture and high performance computing (SBAC-PAD), Sao Paulo
11. Hardavellas N, Ferdman M, Falsafi B, Ailamaki A (2009) Reactive NUCA: near-optimal block placement and replication in distributed caches. Proc. of the 36th international symposium on computer architecture (ISCA-09). Austin, pp 184–195
12. Merino J, Puente V, Prieto P, Gregorio JA (2008) SP-NUCA: a cost effective dynamic non-uniform cache architecture. *ACM SIGARCH Computer Architecture News* 36(2):64–71, New York
13. Hammoud M, Cho S, Melhem R (2009) ACM: an efficient approach for managing shared caches in chip multiprocessors. Proc. 4th int. conf. on high performance embedded architectures and compilers (HiPEAC-09). Paphos, Cyprus, pp 355–372
14. Bardine A, Foglia P, Gabrielli G, Prete CA (2007) Analysis of static and dynamic energy consumption in NUCA caches: initial results. Proc. of the MEDEA 2007 workshop. Brasov, Romania, pp 105–112
15. Bardine A, Comparetti M, Foglia P, Gabrielli G, Prete CA (2010) Way-adaptable D-Nuca caches. *International Journals of High Performance Systems Architecture* 2(3/4):215–228
16. Bartolini S, Foglia P, Prete CA, Solinas M (2010) Feedback driven restructuring of multi-threaded applications for NUCA cache performance in CMPs. 22nd international symposium on computer architecture and high performance computing. Petropolis, Brazil
17. Bardine A, Comparetti M, Foglia P, Gabrielli G, Prete CA (2009) Impact of on-chip network parameters on NUCA cache performance. *IET Computers & Digital Techniques* 3(5):501–512
18. Foglia P, Monni G, Prete CA, Solinas M (2010) Re-Nuca: boosting CMP performances through block replication. In: 13th EUROMICRO conference on digital system design, architectures, methods and tools (DSD2010), Lille, 1–3 Sep 2010. IEEE CS, Los Alamitos, pp 199–206. ISBN: 978-0-7695-4171-6

## Numerical Algorithms

- [Algebraic Multigrid](#)
- [Asynchronous Iterative Algorithms](#)
- [Dense Linear System Solvers](#)
- [Eigenvalue and Singular-Value Problems](#)
- [Linear Algebra, Numerical](#)
- [Linear Least Squares and Orthogonal Factorization](#)
- [Multifrontal Method](#)
- [Preconditioners for Sparse Iterative Methods](#)
- [Rapid Elliptic Solvers](#)
- [Reordering](#)
- [SPAI \(SParse Approximate Inverse\)](#)

## Numerical Libraries

- [ATLAS \(Automatically Tuned Linear Algebra Software\)](#)
- [BLAS \(Basic Linear Algebra Subprograms\)](#)
- [ILUPACK](#)
- [LAPACK](#)
- [libflame](#)
- [MUMPS](#)
- [PARDISO](#)
- [PETSc \(Portable, Extensible Toolkit for Scientific Computation\)](#)
- [PLAPACK](#)
- [SPIKE](#)
- [ScaLAPACK](#)
- [SuperLU](#)

## Numerical Linear Algebra

► [Linear Algebra](#), [Numerical](#)

## NVIDIA GPU

MICHAEL GARLAND

NVIDIA Corporation, Santa Clara, CA, USA

### Synonyms

[Graphics processing unit](#)

### Definition

NVIDIA's GPUs (Graphics Processing Units) are a family of microprocessors that provide a unified architecture for both visual and parallel computing. They generate and process complex 2-D and 3-D imagery, typically via application programming interfaces such as Microsoft's DirectX or OpenGL. NVIDIA's CUDA architecture provides a platform for executing massively parallel computations, written in standard languages such as C or Fortran, on these processors.

### Discussion

Graphics Processing Units, or GPUs, are massively parallel microprocessors. They are present in every workstation, game console, desktop computer, and laptop in use today. They are also increasingly common in mobile devices such as smartphones. The GPU is responsible for generating the 2-D and 3-D graphics required by modern applications, and it provides a platform for real-time image and video processing. These visual computing capabilities are typically accessed via standard application programming interfaces (APIs) such as OpenGL or Microsoft's DirectX. Modern NVIDIA GPUs also support the CUDA parallel programming architecture, and can be programmed directly in standard languages such as C and Fortran. This makes them a platform for parallel computing in general.

### Accelerating Computer Graphics

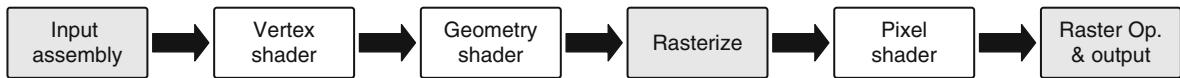
Historically, the GPU has evolved to meet the needs of applications that generate and process complex

computer graphics. To understand the design of a GPU processor, it is instructive to understand the graphical tasks that they have been designed to accelerate.

The real-time rendering of 3-D computer graphics is a computationally challenging task. Essentially all APIs designed for real-time rendering adopt a pipeline execution model. As an example, Fig. 1 shows a high-level block diagram of the Direct3D 10 pipeline [2]. Geometric primitives – typically points, lines, and polygons – are generated by the application and enter the graphics pipeline at the left. Primitives and their vertices are independently processed by the geometry and vertex shader stages, respectively. The rasterization stage converts each primitive into a set of pixel fragments, one for each pixel on the screen touched by the primitive. Each of these pixel fragments is processed in the pixel shader stage, and the results are composited into the framebuffer at the end of the pipeline.

The gray-colored stages of the pipeline are fixed-function components; their operation is restricted to a set of functionality defined in the relevant API standards. The others allow the application to provide custom procedures – typically referred to as “shaders” – that process individual primitives, vertices, and pixels. Shader programs are typically written in C-like domain-specific languages such as Cg [6], OpenGL's GL Shading Language (GLSL), or the DirectX High-Level Shading Language (HLSL).

The design and evolution of GPU processors has been influenced in a number of ways by the needs of interactive graphics. Foremost among these is the fact that graphics represents a workload with tremendous inherent parallelism. Even the simplest of 3-D scenes typically contain thousands of polygon primitives, and scenes containing a million or more primitives are not uncommon. At high-definition resolutions, a display device contains 2 million pixels. Rendering the scene involves processing each primitive, each vertex of every primitive, and each pixel touched by every primitive. The graphics pipeline model mandates that every primitive/vertex/pixel can be processed independently of every other primitive/vertex/pixel. Thus, at any one instant in time, a GPU may be given a workload with many millions of independent tasks, all of which could be processed in parallel given the necessary computational resources.



NVIDIA GPU. Fig. 1 Block diagram of the graphics pipeline execution model of Direct3D 10

Interactive computer graphics is also a fundamentally throughput-oriented problem. Applications typically render their scenes at a fixed frame rate, such as 30 frames per second. The amount of work that can be completed in 1/30 of a second is thus of primary importance, since this will largely determine the richness of the resulting visual experience. This has led GPU processors toward aggressive throughput-oriented designs; they are generally willing to sacrifice performance of a single task in order to increase total throughput over all tasks.

### Parallel Computing Architecture

Current GPUs, as illustrated in Fig. 2, are designed around a large array of parallel processors and various fixed-function units for tasks such as work distribution, rasterization, and image processing. The pipeline execution model of the graphics APIs is mapped onto the machine by circulating work through these processors. Shader programs provided by the application for the various programmable stages of the pipeline execute on the processor array and fixed function logic handles the scheduling of these tasks. This basic architecture has been used in all NVIDIA GPUs beginning with the G80 processor first released in the GeForce 8800 GTX in late 2006. This first-generation processor was later followed by the GT200 processor, which first introduced double precision floating point support. NVIDIA recently released its third major generation of unified processors, a GPU architecture code-named “Fermi.”

In addition to executing all shader programs for graphics applications, this parallel processor array can also be programmed directly. NVIDIA’s CUDA architecture provides a programming model and application interface for executing parallel programs – referred to as “kernels” – on the GPU. This programming model can be adapted to most typical sequential programming languages, and compilers for CUDA C, C++, and Fortran are currently available.

The programmer organizes a CUDA program into a sequential host program, intended to run on the CPU, and a collection of parallel kernels, that are

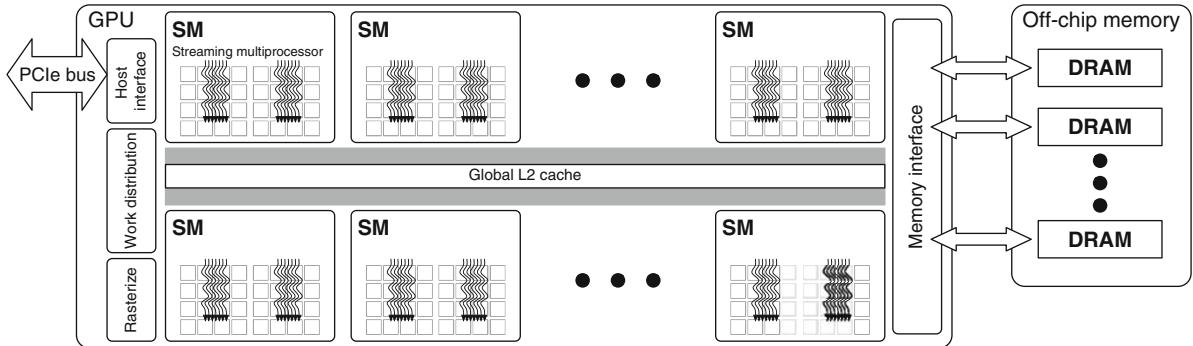
intended to run on the GPU. A single kernel is a blocked SPMD (Single Program Multiple Data) parallel computation. It consists of a collection of concurrent threads, all executing the same program. These threads are further grouped into thread blocks. When launching a kernel, the application specifies both the number of thread blocks and the number of threads per block to instantiate.

Figure 3 shows a simple example of a CUDA C program fragment for computing  $y \leftarrow \alpha x + y$  for a scalar  $\alpha$  and two  $n$ -vectors  $x$  and  $y$ . This is the so-called SAXPY kernel of the BLAS linear algebra library.

The `parallel_saxpy` routine is part of the host program, as indicated by the `_host_` attribute attached to its declaration. Given pointers to the  $x$  and  $y$  vectors allocated in the GPU memory, it invokes the parallel kernel. This code is designed to process each corresponding pair of elements with a single thread. Therefore, the host program selects a thread block size of 256 threads – an arbitrary choice in this case since there is no inter-thread communication – and determines the number of 256-thread blocks needed to create at least 1 thread per element. The function call-like `saxpy_kernel<<<B, T>>>()` actually launches the kernel with  $B$  blocks of  $T$  threads each.

The procedure `saxpy_kernel` is a kernel entry point, indicated by the `_global_` attribute. Although each thread may subsequently follow its own execution path, all threads of the kernel begin their execution at the beginning of this procedure. Thus, the kernel procedure itself is a standard sequential program describing the action of a single thread.

Individual threads within a kernel are differentiated by unique integer indices. The thread blocks of a kernel are numbered on a 1-D or 2-D grid. CUDA C exposes these indices through special variables visible within kernel functions: `blockIdx.x`, `blockIdx.y` give the index of the current block and `gridDim.x`, `gridDim.y` provide the number of blocks in each dimension. Similarly, each thread is given a unique index within its block. These indices may be one-, two-, or three-dimensional. A thread’s index is available via



NVIDIA GPU. Fig. 2 High-level structure of a modern discrete GPU

```

global__ void saxpy_kernel(int n, float a, const float *x, float *y)
{
 // Each thread processes 1 element, determined from the thread's index.
 int i = blockIdx.x*blockDim.x + threadIdx.x;

 if(i<n) y[i] = a*x[i] + y[i];
}

host__ void parallel_saxpy(int n, float a, const float *x, float *y)
{
 // Create kernel with 256-thread blocks.
 int blocksize = 256;

 // We will need [n/256] blocks to process n elements.
 int nblocks = ceil(n/blocksize);

 saxpy_kernel<<<nblocks, blocksize>>>(n, a, x, y);
}

```

NVIDIA GPU. Fig. 3 CUDA C implementation of  $y \leftarrow \alpha x + y$  given two  $n$ -element vector  $x$  and  $y$

the special variables `threadIdx.x`, `threadIdx.y`, and `threadIdx.z`, while the dimensions of a block are `blockDim.x`, `blockDim.y`, and `blockDim.z`.

The example shown in Fig. 3 uses only 1-D indexing; all other dimensions are ignored. It begins by computing a linear index  $i$  from the block and thread indices of the current thread. This assigns unique elements  $x_i$  and  $y_i$  to the thread. The conditional test that  $i < n$  is required since there will be more threads than elements when  $n$  is not a multiple of the block size, which is 256 in this case. Those threads that are within the bounds of the array conclude their computation by performing the element-wise update  $y_i \leftarrow \alpha x_i + y_i$ .

Figure 3 illustrates the basic mechanics of a CUDA program, but it also highlights an important qualitative difference between CPU and GPU programming. Typical CPUs tend to treat threads as heavyweight

objects of which there are relatively few. Concurrency runtimes like OpenMP and Intel's Threading Building Blocks typically create a persistent pool of worker threads proportional in size to the number of cores available. On current machines, this would typically imply a thread count of up to roughly 16, and these threads most likely persist over the application's entire lifetime. Their schedulers are then responsible for mapping tasks created by the application onto the available threads. In contrast, GPUs treat threads as extremely lightweight objects that may also have extremely brief lifetimes. Given two vectors of ten million elements, this single SAXPY kernel will launch, execute, and retire ten million threads. This significantly different viewpoint is largely a result of the GPU's graphics background. Graphics applications require GPUs to execute many millions of shader invocations per frame. Unlike CPUs,

which tend to assume that parallelism in their given workload is fairly scarce, GPUs assume that they are provided with workloads containing abundant parallelism.

### The Streaming Multiprocessor

The parallel processor array at the heart of the GPU is composed of a number of multithreaded Streaming Multiprocessors (SMs). When a kernel is launched on the GPU, the hardware work distributor enumerates its thread blocks and launches them on SMs as resources are available. If no SM can accommodate any additional blocks, the work distributor waits for some running block to retire, at which point it may launch a pending block on that SM. The SM is thus “streaming” in the sense that thread blocks “stream” through the multiprocessor: they launch, run, and then retire without being suspended and evicted to memory.

Each multiprocessor supports on the order of 1,000 coresident threads, with the precise number varying between processor generations. [Table 1](#) provides specific data for the three most recent generations of NVIDIA GPUs. The SM manages the creation, execution, and scheduling of its threads directly in hardware.

### Memory Organization

All threads running on the GPU have access to a global random-access memory. For discrete GPUs – those housed on PCIe cards – global memory is serviced by a high-bandwidth memory system that can deliver roughly 150 GB/s of bandwidth on the highest end cards. For GPUs in other form factors, such as GPUs integrated with the motherboard chipset, this memory may be the same system memory used by the CPU.

**NVIDIA GPU. Table 1** Capacity of each SM for three GPU generations

|                        | G8x/G9x | GT2xx  | “Fermi” |
|------------------------|---------|--------|---------|
| Scalar cores           | 8       | 8      | 32      |
| Registers (32-bit)     | 8192    | 16,384 | 32,768  |
| Coresident threads     | 768     | 1024   | 1536    |
| Shared memory (KB)     | 16      | 16     | 48/16   |
| L1 cache (KB)          | –       | –      | 16/48   |
| L2 cache (KB per chip) | –       | –      | 768     |

An SM is provisioned with a large register file so that every resident thread can be given its own dedicated register set. For example, the register file of a Fermi SM has 32,768 32-bit registers, for a total of 128 KB of register space. At its maximum occupancy level of 1,536 resident threads, the Fermi SM can allocate 21 registers to every thread. These registers provide primary storage for the local variables of the thread, with excess variables spilling to external memory.

In addition to the register file, which provides for private per-thread storage, the SM is also equipped with a small on-chip shared memory. This memory is high speed and low latency. It is allocated on a per-block basis, thus providing a space for sharing data among all the threads of a block and for intra-block communication.

The Fermi architecture augments this on-chip RAM with an L1/L2 cache hierarchy. Each SM maintains its own private L1 cache. The L1 shares a fixed 64 KB memory space with the shared memory RAM. This memory may be divided into either 16 KB L1 and 48 KB shared memory, or 48 KB L1 and 16 KB shared memory. The L2 cache is a global cache across all SMs and is 768 KB in size.

### Thread Synchronization

Parallel threads running on the GPU may synchronize with each other at one of three levels. At the highest level, there is an implicit barrier between successive kernels. By default, no thread of a given kernel may be launched until all threads of previously launched kernels have completed. For truly independent kernels, CUDA provides an additional mechanism to launch kernels in separate streams, thus specifying that they may be run concurrently.

The threads of a single thread block may synchronize with each other by calling a barrier function. This is a lightweight barrier that translates to a single SM instruction. It thus incurs essentially no overhead, except for the time required to physically wait for all threads to reach the barrier.

Individual threads may also perform atomic operations on memory addresses. These operations include, but are not limited to, atomic compare and swap, atomic increment/decrement, and atomic reduction for commutative operations such as addition, min/max,

and binary and/or/xor. Atomic operations may be performed in both global and on-chip shared memory.

### SIMT Execution

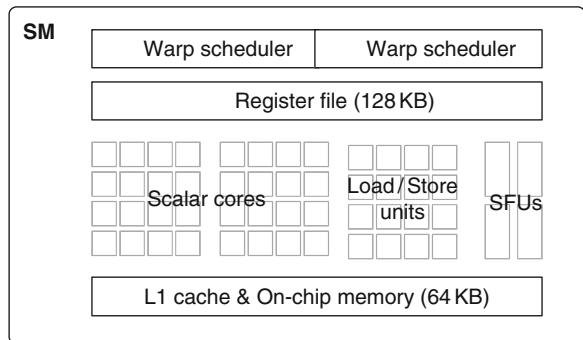
To manage its large population of threads efficiently, the SM employs a SIMT – Single Instruction, Multiple Thread – architecture. The threads of a given block are grouped into 32-thread units called “warps.” A warp of threads can execute a single instruction at a time across all its threads. Since each thread may follow its own independent path of execution, threads within a warp may potentially follow different code paths. The hardware automatically tracks and handles such execution divergence. When a warp of threads executes a divergent conditional, for instance, it will execute first one branch and then the other of the conditional. Threads of the warp will be activated only in the branch of the conditional which they chose to follow.

The SIMT architecture used by the GPU is an instance of the broader class of SIMD (Single Instruction, Multiple Data) architectures. However, it is qualitatively different from the vector SIMD extensions prevalent on other platforms. The action of each thread is described by a normal sequence of scalar instructions. When executing the same code path, these threads will automatically execute together and realize the benefits of SIMD execution. When they follow different paths, the execution of the program remains correct, although there is a performance loss proportional to the number of diverging paths. In contrast, a vector SIMD model embeds vector instructions directly in the instruction stream. They must be inserted explicitly, either directly by the programmer or by compiler autovectorization of loops.

Warps are the fundamental unit of scheduling in the SM. On each clock cycle, an SM warp scheduler is free to select from its available pool any runnable warp to issue an instruction. Runnable warps are simply those not waiting for some pending operation to complete. Thus, the scheduler interleaves the execution of separate warps at a very fine level. When there are multiple runnable warps, this has the effect of switching between warps at each instruction.

### Processing Elements

The SM contains a collection of processing elements that execute the instructions issued by running threads.



NVIDIA GPU. Fig. 4 Basic structure of a single streaming multiprocessor in a Fermi GPU

The precise configuration of these elements varies between GPU generations. Figure 4 sketches the configuration for a Fermi SM.

Each SM is built around 32 scalar cores. Each of these cores provides fully pipelined integer and floating point arithmetic units. The floating point units support IEEE 754-2008 single (32-bit) and double (64-bit) precision arithmetic based on the FMA (Fused Multiply Add) operation.

The scalar cores are grouped in two sets of 16, and each group is associated with a separate warp scheduler. Thus, two separate warps may be executing simultaneously. A 32-thread warp is executed on one set of 16 cores by running two half-warps of 16 threads each, back to back on the scalar cores. The scalar cores are augmented with a set of 16 Load/Store units, that handle address calculations and dispatch for memory transfers, and four Special Function Units (SFUs) that execute transcendental instructions such as sine, cosine, reciprocal, and square root.

N

### Throughput-Oriented Design

As observed above, one of the key characteristics of the workloads generated by interactive graphics applications is that they are fundamentally throughput oriented. The goal is to finish as many tasks as possible within the fixed time period of a single frame. Since the number of tasks that must be completed easily reaches into the tens of millions, the amount of time required to complete any particular task is essentially insignificant. For such workloads, the amount of available parallelism is obviously abundant. This abundance of parallelism, coupled with the throughput-oriented nature of the

problems it is typically targeted for, combines to drive a number of GPU architectural design decisions.

One major consequence of these assumptions is that the GPU relies extensively on multithreading to hide the latency of long-running operations. This is true even of relatively short latencies, such as the time required for instructions to make their way through the instruction pipeline, but it is particularly true for accesses to global memory. On all modern processors, the time that elapses between issuing a memory request to external DRAM and its completion is typically hundreds of clock cycles. No high-performance processor can afford to remain idle for such a length of time. Traditional processors, like CPU cores, which emphasize running a single sequential task as quickly as possible attempt to avoid long-latency memory operations through large, sophisticated on-chip caches. In contrast, GPUs largely rely on multithreading to hide this latency; it is perfectly acceptable for some threads to be blocked waiting for memory transactions to complete as long as many others are ready to run. NVIDIA GPUs prior to the Fermi architecture did not have any caches for load/store operations on external memory. Even the Fermi architecture, which employs a full cache hierarchy, still relies heavily on multithreading to hide latency.

Another consequence of the throughput-oriented design of GPUs is that they use relatively simple processor control schemes. Modern superscalar CPUs employ a variety of sophisticated techniques to accelerate sequential programs. These include out-of-order execution, branch prediction, and speculative execution. GPUs do not employ any of these techniques. The scalar cores of the SM are simple in-order pipelined processors. They perform neither branch prediction nor speculative execution. While this means that a single thread may take longer to run, simpler control logic translates into less chip area devoted to control. Hence, there is more area available for functional units which leads to higher total throughput for a fixed chip area.

The use of SIMD execution in the SM similarly increases total peak throughput by amortizing the cost of control over many functional units. This comes at the cost of lower performance on completely divergent tasks. However, in practice, a massively parallel computation that provides a pool of tens of thousands of concurrent tasks is likely to show a high degree of uniformity. Rather than 10,000 disparate tasks, it is generally

much more likely that there will be a fairly small number of task types, each of which provides ample uniformity for efficient SIMD execution.

## Related Entries

- ▶ [Barriers](#)
- ▶ [BSP \(Bulk Synchronous Parallelism\)](#)
- ▶ [Shared-Memory Multiprocessors](#)
- ▶ [SPMD Computational Model](#)

## Bibliographic Notes and Further Reading

The physical design of early hardware systems for 3-D graphics acceleration, exemplified by the Silicon Graphics Reality Engine [1], tended to mimic the structure of the graphics pipeline execution model. The hardware itself was a feed-forward pipeline of fixed-function hardware units. Over time, graphics hardware evolved, introducing programmable units for vertex, pixel, and geometry processing.

Lindholm et al. [5] provide an overview of the original G80 chip architecture, and an NVIDIA whitepaper outlines the basic characteristics of the upcoming Fermi architecture [8]. Nickolls et al. [7] explain the basics of the CUDA programming model. More complete information can be found in the official CUDA Programming Guide [9] and a recently published book on parallel programming by Kirk and Hwu [4]. Owens et al. [10] survey the field of GPU computing and its historical development. Garland et al. [3] report on experiences with CUDA in a collection of real-world applications.

An extensive collection of documentation, training materials, examples applications, and the CUDA development environment itself is all available online at <http://www.nvidia.com/CUDA>.

## Bibliography

1. Akeley K (1993) Reality engine graphics. In: Proceedings of SIGGRAPH '93, ACM, Spencer, pp 109–116
2. Blythe D (2006) The direct 3D 10 system. ACM Trans Graph 25(3):724–734
3. Garland M, Grand SL, Nickolls J, Anderson J, Hardwick J, Morton S, Phillips E, Zhang Y, Volkov V (2008) Parallel computing experiences with CUDA. IEEE Micro 28(4):13–27
4. Kirk DB, Hwu WMW (2010) Programming massively parallel processors: a hands-on approach. Morgan Kaufmann, Burlington

5. Lindholm E, Nickolls J, Oberman S, Montrym J (2008) NVIDIA Tesla: a unified graphics and computing architecture. *IEEE Micro* 28(2):39–55
6. Mark WR, Glanville RS, Akeley K, Kilgard MJ (2003) Cg: a system for programming graphics hardware in a C-like language. In: *Proceedings of SIGGRAPH 2003*, ACM, San Diego, pp 896–907
7. Nickolls J, Buck I, Garland M, Skadron K (2008) Scalable parallel programming with CUDA. *Queue* 6(2):40–53
8. NVIDIA's next generation CUDA compute architecture: Fermi (2009) <http://www.nvidia.com/fermi>
9. NVIDIA Corporation (2009) NVIDIA CUDA Programming Guide, July 2009. Version 2.3
10. Owens JD, Houston M, Luebke D, Green S, Stone JE, Phillips JC (2008) GPU computing. In: *Proceedings of the IEEE 96*, No. 5, Santa Clara

## NWChem

N. GOVIND, ERIC J. BYLASKA, WIBE A. DE JONG,  
K. KOWALSKI, TJERK P. STRAATSMA, MARAT VALIEV,  
H. J. J. VAN DAM  
Pacific Northwest National Laboratory, Richland,  
WA, USA

### Synonyms

Computational chemistry; Quantum chemistry

### Definition

NWChem (Northwest Computational Chemistry Package) is DOE's premier massively parallel computational chemistry package. It is an open-source high-performance platform with extensive capabilities for large-scale quantum and classical atomistic simulations that can be applied to a broad range of application areas.

### Discussion

Computational atomistic modeling has emerged as a powerful approach for fundamental studies in chemistry, materials science, geosciences, and biology. It has the ability to provide detailed insights into complex phenomena and, over the years, has grown into a powerful tool not only to validate experiments, but also to predict new phenomena. In order to harness computational modeling to the fullest, superior computational tools are required that not only offer a broad spectrum of capabilities, but that are also capable of taking advantage of modern massively parallel computer architectures efficiently to solve real-world problems.

The NWChem program suite is an example along these lines. It offers a range of capabilities to perform first-principles electronic structure calculations on molecular and periodic systems, classical molecular dynamics simulations, and hybrid QM/MM simulations that permit one to combine quantum and classical methods within a calculation.

The code is designed to run on high-performance parallel supercomputers as well as conventional workstation clusters with the goal to provide scalable solutions for large-scale atomistic simulations. It has been ported to almost all high-performance computing platforms, workstations, PCs running LINUX, as well as clusters of desktop platforms or workgroup servers. The package is scalable, both in its ability to treat large problems efficiently and in its utilization of available parallel computing resources. The parallel framework for NWChem is provided by the Global Array (GA) toolkit developed at PNNL.

NWChem is developed and maintained by the Environmental Molecular Sciences Laboratory (EMSL) located at the Pacific Northwest National Laboratory (PNNL) in Washington State. The latest release (version 6.0) of the code is distributed under the terms of the Educational Community License (ECL) version 2.0.

### Overview

NWChem [46] provides a number of quantum mechanical or electronic structure approaches for computing various properties of molecules, finite clusters, and periodic systems [46]. These include self-consistent field (SCF) or Hartree–Fock (HF) theory, density functional theory (DFT) including support for a broad range of state-of-the-art exchange-correlation (xc) functionals, plane-wave-based DFT methods for Car–Parrinello molecular dynamics (CPMD) and band structure calculations of periodic systems, higher-order many body methods like second-order Moller–Plesset (MP) perturbation theories and a full range of coupled cluster (CC) theories, relativistic approximations (DKH, ZORA), excited state approaches like time-dependent density functional theory (TDDFT), and equation of motion coupled cluster (EOMCC) theories. Most of the above approaches can be used to perform geometry optimizations (structure minimization and transition state location), frequency calculations, and/or molecular response calculations either analytically or

numerically. Macromolecular simulations and free-energy computations can be performed using classical molecular dynamics with parameterized classical force fields. These methodologies may also be combined to perform multi-physics mixed quantum mechanics/molecular mechanics (QM/MM) simulations. Other capabilities include hybrid approaches like ONIOM, continuum solvation models like COSMO, electron transfer (ET), as well as interface to a number of other programs.

## Core Capabilities

Since an exhaustive discussion of all the capabilities is beyond the scope of this essay, only a brief overview of the core capabilities is provided here. The interested reader is referred to published review articles [1, 10, 19, 46, 48] for a complete description.

## Self-consistent Field and Density Functional Theories

The Hartree–Fock (HF) or self-consistent field (SCF) method is a single determinant-based theory [45] and is an essential in any computational chemistry suite. It forms the basis for higher level electronic structure theories like Moller–Plesset perturbation theory (MP), coupled cluster (CC) theory-and other post-HF approaches. Density functional theory (DFT) [21, 35, 37] provides an alternative route to the many-electron problem and offers an excellent balance between accuracy and computational performance. It has emerged as a powerful theory that is broadly applicable. In analogy with the HF equations, DFT is rooted in the Kohn–Sham (KS) equations with the only difference being in the way the exchange and correlation is treated. In DFT, the HF exchange is replaced with a (in general nonlocal) density-dependent exchange and correlation contribution which encapsulates the complexities of the many-electron interactions. Since the exact form of the exchange and correlation in DFT contribution is an unsolved problem, a number of approximations of increasing complexity have been developed over the years [39], most of which are available in NWChem.

The HF/KS equations are very similar in structure and are typically solved using an iterative procedure with the help of basis set expansions [16, 25, 26, 45].

Two popular basis set approaches are implemented in NWChem: (1) Gaussians and (2) Plane waves.

### Gaussian Basis Set Approach

Within the Gaussian basis set approach [16, 45], both HF/KS can be expressed as a set of matrix equations:

$$\mathbf{FC} = \mathbf{SC}\epsilon \quad (1)$$

where  $\mathbf{F}$ ,  $\mathbf{C}$ ,  $\mathbf{S}$ , and  $\epsilon$  represent the Fock, coefficient, overlap, and diagonal orbital energy matrices, respectively. The problem of solving the HF/KS equations thus boils down to solving a generalized eigenvalue problem. The Fock matrix for both HF and KS formalisms can be encompassed under a single framework as:

$$F_{\mu\nu} = H_{\mu\nu}^{\text{core}} + G_{\mu\nu}^j + \alpha G_{\mu\nu}^k + \beta G_{\mu\nu}^{x-dft} + \gamma G_{\mu\nu}^{c-dft} \quad (2)$$

where  $H^{\text{core}}$  is the one-electron contribution (kinetic and ion-electron),  $G^j$  and  $G^k$  represent the two-electron (Coulomb and explicit exchange),  $G^{x-dft}$  and  $G^{c-dft}$  are the DFT exchange and correlation parts, respectively. The mixing coefficients  $\alpha$ ,  $\beta$ , and  $\gamma$  help span the HF and DFT limits. With  $\alpha = 1$ ,  $\beta = 0$ ,  $\gamma = 0$  one gets the pure HF limit, while the pure DFT limit is obtained with  $\alpha = 0$ ,  $\beta = 1$ ,  $\gamma = 1$  and  $\alpha < 1$ ,  $\beta < 1$ ,  $\gamma = 1$  covers the phase space of nonlocal hybrid-DFT forms.

The time-consuming steps in any standard implementation are computation of the two-electron integrals, construction of the two-electron contribution to the Fock matrix, computation of the exchange-correlation contribution to the Fock matrix in the case of the DFT, diagonalization of the Fock matrix, and computation of the density matrix using the molecular orbital coefficients. Within this context, only the parallelization approaches implemented in NWChem will be discussed in the following paragraphs. The reader is also referred to other detailed work on the subject [10, 12, 15, 18].

The calculation of the two-electron integrals is the most expensive and formally scales as  $O(N^4)$  even though it can be shown that the scaling is  $\approx O(N^2)$  for large molecular systems. In most parallel implementations, including NWChem, the relevant matrices are either handled in a replicated or distributed fashion or a combination [12, 15]. The replicated data approach offers a straightforward way to achieve task parallelism and low communication. Each processor maintains a copy of the necessary data. In the case of

the two-electron contribution, the Fock ( $F$ ) and density ( $D$ ) matrices are replicated over all the processors and the integral quartets are assigned to each processor as blocks. A partial  $F$  matrix is constructed on each processor and then consolidated into a full  $F$  matrix using a global sum operation. For a reasonable work distribution and an efficient global summation, this approach is perfectly parallelizable. It is also efficient for large systems because of the amount of parallel work available especially if the integrals are evaluated on-the-fly (also known as the direct approach). However, there is a potential  $O(N^2)$  memory bottleneck. The distributed approach avoids the memory bottleneck by distributing the  $F$  and  $D$  matrices, thus placing a smaller constraint on the available local memory. However, this approach is potentially more challenging from an implementation standpoint because of the extra bookkeeping. Figure 1 illustrates the strong scaling of the Gaussian basis set DFT module.

Both ground and excited state calculations on finite systems can be performed using the Gaussian basis set-based (HF/DFT) module in NWChem.

### Plane Wave Approach

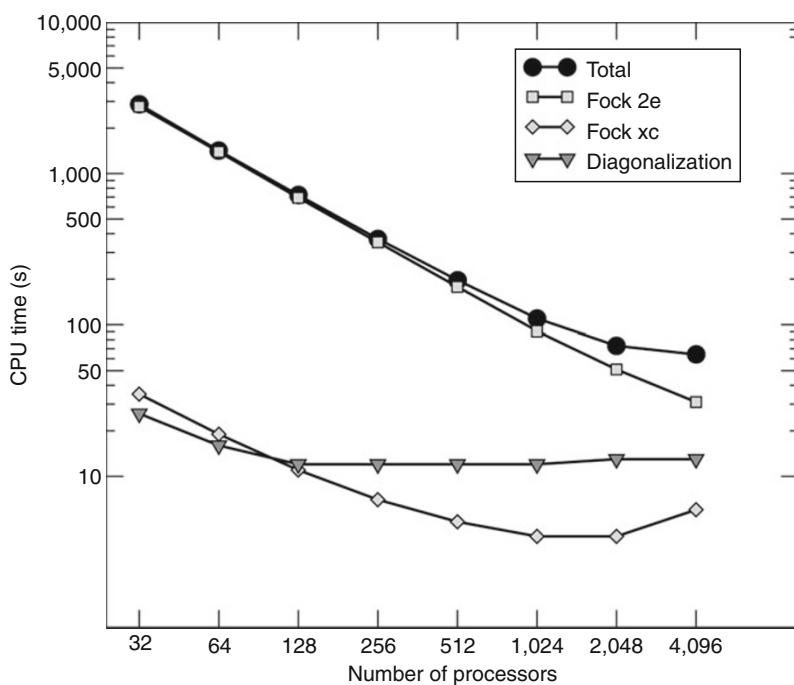
The plane wave DFT module in NWChem is composed of three components that use a common infrastructure: PSPW (a pseudopotential-based  $\Gamma$ -point code for isolated and periodic systems), BAND (a code to calculate the band structure of crystals and surfaces), and PAW (a projector-augmented-wave-based  $\Gamma$ -point code for isolated and periodic systems). Within the plane wave approach the single-particle wavefunctions of a periodic system can be expanded in general as [4, 25, 26]:

$$\psi_{jk}(\mathbf{r}) = e^{i\mathbf{k}\cdot\mathbf{r}} \sum_{\mathbf{G}} \tilde{\psi}_{jk}(\mathbf{G}) e^{i\mathbf{G}\cdot\mathbf{r}} \quad (3)$$

where  $\mathbf{k}$  is a vector in the first Brillouin zone,  $\mathbf{G}$  is the reciprocal lattice vector, and  $j$  represents the orbital index, respectively. For isolated systems like molecules, the Brillouin zone sampling is limited to the  $\Gamma$ -point ( $\mathbf{k}=0$ ) which in turn yields:

$$\psi_j(\mathbf{r}) = \sum_{\mathbf{G}} \tilde{\psi}_j(\mathbf{G}) e^{i\mathbf{G}\cdot\mathbf{r}} \quad (4)$$

The size of the plane wave basis set expansion is determined by the maximum kinetic energy cutoff ( $E_{cut}$ ) via



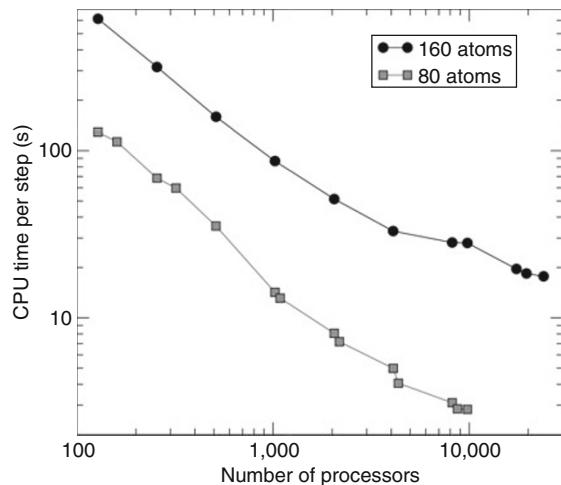
**NWChem. Fig. 1** Parallel scaling of the Gaussian basis set HF/DFT implementation on the  $C_{240}$  system. Calculations were performed using the PBE0 hybrid exchange-correlation functional and the 6-31G\* basis with a total of 3,600 basis functions for the whole system. Integral evaluations were performed using the on-the-fly (or direct) approach and Fock matrix replication. The figure shows the scaling of the various components

the following relation:

$$\frac{1}{2}|\mathbf{G}|^2 < E_{cut} \quad (5)$$

Some favorable features of the plane wave expansion include: being able to treat periodic systems in a seamless way, efficient calculation of the expansion coefficients using Fast Fourier Transform (FFT) techniques, independence of the basis with respect to nuclear positions which makes it immune to superposition, and over-completeness problems which are critical issues in local basis set approaches.

Because plane wave basis sets are typically much larger than a local basis, explicit Fock matrix construction and diagonalization is avoided in favor of direct optimization approaches like conjugate gradient minimization [38]. The most time-consuming steps of plane wave-based algorithms are the evaluation of the pseudopotential (specifically the nonlocal contribution), wavefunction orthogonalization, and exact exchange (if needed) in the description of the exchange-correlation. Several parallelization strategies [4, 5, 7, 26, 29] have been implemented in NWChem to mitigate issues such as parallel data distribution over the wavevectors, distribution of the orbitals, and distribution of the real and reciprocal space grids. The wavevector distribution is a popular approach for solid-state applications, but the scalability is hindered by the size of the Brillouin zone and the method is not applicable to the pure  $\Gamma$ -point calculations. Orbital distribution has been shown to scale favorably, but this entails storage of the entire one-electron orbital on a single node and is not practical for large molecular systems which typically require a large plane wave expansion. The most commonly implemented strategy, including NWChem, is the spatial distribution of the real and reciprocal grid points (typically along a single dimension) are stored on different processors. This approach is not only scalable, but also fairly economical from a memory management standpoint. The trade-off here is the increased communication costs due to parallel three-dimensional FFTs which plateau beyond  $\sqrt[3]{N_1 N_2 N_3}$  processors ( $\approx 100$  for typical applications). This limitation may be avoided by distributing the spatial and orbital degrees of freedom on a two-dimensional processor grid [14]. This approach is also available in NWChem.



**NWChem. Fig. 2** Parallel scaling of hybrid DFT plane wave calculations on the  $\text{Fe}_2\text{O}_3$  system for unit cells containing 80 and 160 atoms, respectively

The plane wave DFT module in NWChem also has a highly scalable algorithm [4, 10, 46] for exact exchange which makes it possible to perform calculations using hybrid exchange-correlation functionals. Using this approach, scaling to over 20,000 CPUs for modest size problems has been achieved. This is shown in Fig. 2 where the parallel scaling is demonstrated for  $\text{Fe}_2\text{O}_3$  with unit cells containing 80 and 160 atoms, respectively.

### Coupled Cluster Theory

Coupled cluster theory [3, 6, 36] (CC) is a method rooted in many-body perturbation theory (MBPT) and is considered by many to be the gold standard for accurate quantum mechanical calculations of ground and excited states. Over the years, it has grown into one of the most reliable approaches for describing correlation effects in atoms and molecules. In CC theory the wavefunction is expressed as an exponential ansatz:

$$|\Psi_0\rangle = e^{\hat{T}}|\Phi\rangle \quad (6)$$

where  $|\Phi_0\rangle$  is the ground-state function.  $\hat{T}$  is the cluster operator which produces a linear combination of excited determinants when acting on the reference function  $|\Phi\rangle$  (typically represented by the HF determinant). In CC theory, the energy is obtained as:

$$E = \langle\Phi|e^{-T}He^T|\Phi\rangle \quad (7)$$

The systematic inclusion of excitations of higher rank in the cluster operator results in a series of more accurate approximations. In the basic CCSD [40] approach (CC with singles and doubles) the cluster operator is approximated as  $T \approx T_1 + T_2$ . Similarly, in the more accurate CCSDT [34] approach (CC with singles, doubles, and triples) the cluster operator is represented as  $T \approx T_1 + T_2 + T_3$ . The higher accuracy of the CCSDT energies comes at a steep price because of the extra numerical complexity and scales as  $N^8$  with system size. Consequently, many non-iterative approaches which mediate the cost between the CCSD ( $N^6$ ) and CCSDT ( $N^8$ ) methods have dominated this field, the CCSD(T) [41] approach, which scales as  $N^7$  with system size, being the most well known and most frequently used.

Excited state calculations can be accomplished within single-reference CC theory via the linear response CC [29] (LR-CC) and equation-of-motion CC formalisms (EOMCC). Analogous to ground-state calculations, various iterative (EOMCCSD [8], EOMCCSDT [24]) and non-iterative approaches have been implemented in NWChem [10, 46].

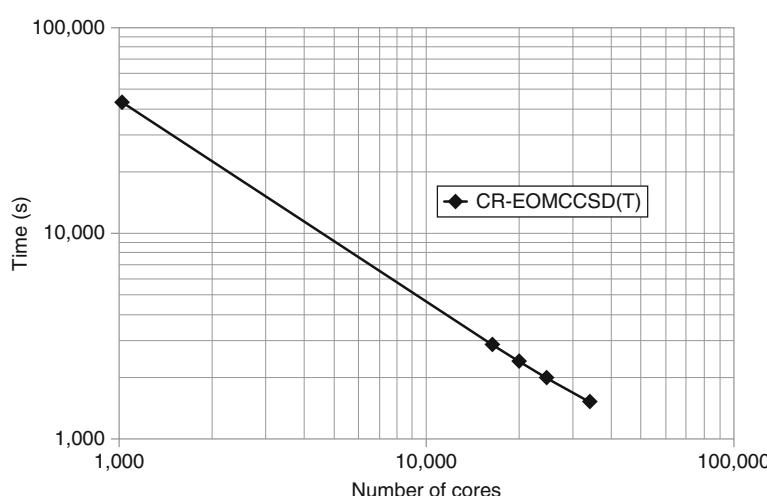
Since the CC equations involve contractions between tensors corresponding to the one- and two-electron integrals and cluster amplitudes, the task of writing efficient parallel CC code is extremely challenging. The main issues that define the overall parallel efficiency are related to (a) data storage and data transfer, (b) efficient load balancing, and (c) data flow during

the execution phase. Another significant issue is associated with the most efficient reduction of the numerical complexity of a given CC method.

NWChem provides a complete implementation of CC theory geared toward efficient execution on high-performance parallel architectures. Two classes of parallel CC codes have been implemented, a spin-free code for closed-shell systems [20] and a code based on the tensor contraction engine (TCE) [17]. It was recently demonstrated that the non-iterative (T) part of the spin-free code can be made to scale across 250,000 processors [2]. The TCE-based codes are automatically generated spin-orbital codes that can be used for a broad range of reference functions (RHF, ROHF, UHF, DFT). Here the block or tile structure of all the tensors is extensively used in the code for dynamic load balancing. This code is also parallelized using the GA library which is used to store all the tensors. The scaling of the CR-EOMCCSD(T) method is shown in Fig. 3. The reader is also referred to more recent applications that demonstrate the scaling of the CC module [22, 23].

### Classical Molecular Dynamics

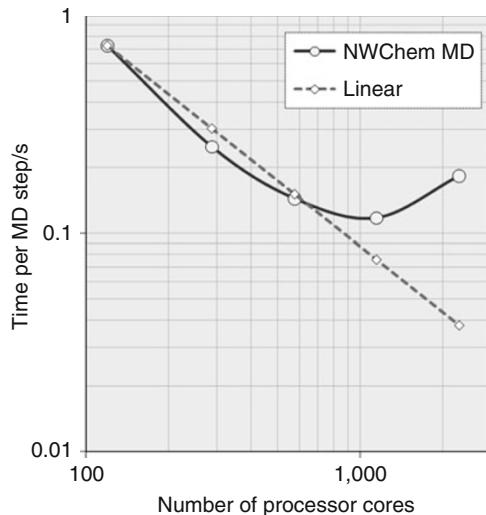
NWChem provides a highly scalable parallel framework for classical molecular dynamics (MD) simulations based on classical potentials (Amber [9]). The parallelization strategy is based on a domain decomposition approach which provides the best theoretical scalability of memory use and communication cost. In the domain



**NWChem. Fig. 3** Parallel scaling of the non-iterative triples correction ( $N^7$  (T)) in CR-EOMCCSD(T) calculations of the green fluorescent protein (GFP) with the cc-pVTZ basis set

decomposition model [43, 44] the physical space occupied by the system is distributed and the atoms are assigned to a processor based on their location. This is accomplished by dividing the total finite or periodic system being studied into slices along the three dimensions. Communication is made more efficient by logically arranging the processors in a three-dimensional grid. This permits one or more sub-domains to be assigned to a processor and preserves locality of the sub-domains which is essential for efficient communication. The domain decomposition has some advantages and disadvantages. The former includes: avoiding atomic data replication on each processor, memory reduction to hold the entire system in core. The latter includes: the need for periodic reassessments of the atoms that move between sub-domains, the need for a sophisticated dynamic load-balancing scheme for heterogeneous systems. Other challenges include dealing with systems with incommensurate spatial decompositions. The MD implementation in NWChem takes advantage of one-sided asynchronous capabilities provided by the GA toolkit, which allows for more sophisticated dynamic load-balancing algorithms to handle these issues.

Electrostatic contributions are evaluated using the smooth Particle Mesh Ewald (sPME) method [11] which employs the efficient three-dimensional Fast Fourier Transform (3D-FFT) approach. The data access patterns in the 3D-FFT, however, hinders efficient parallelization over large numbers of processors. To address this issue, a slab-based 3D-FFT is used so that the transform in two dimensions takes place on a single processor, thereby avoiding interprocessor communication. This leaves one dimension in which communication is required. The main scaling difficulty of this algorithm is that it only allows as many processors to be used for the 3D-FFT as there are slabs in the grid. To remedy this limitation and to achieve strong scaling to a larger number of processors than the number of slabs used to define the charge grid, the reciprocal space calculations are performed on a subset of the total number of processors used to carry out the simulation. Simultaneously with this subset of processors (or subgroup) performing the FFTs, the potentially much larger group of remaining processors handle the direct space interactions. The number of processors to be used for the FFTs can be specified in the input. The use of subgroups



**NWChem. Fig. 4** Time per molecular dynamics step for a SPC/E water simulation in the microcanonical ensemble with 648,000 atoms

is a standard feature of the Global Arrays toolkit. Figure 4 illustrates the time per molecular dynamics step for a SPC/E water simulation in the microcanonical ensemble with 648,000 atoms.

## Combined Quantum Mechanical Molecular Mechanics

The combined quantum mechanical molecular mechanics (QM/MM) module in NWChem provides an effective tool to study localized molecular transformations in large-scale systems such as those encountered in solution chemistry or enzyme catalysis. This approach entails using an accurate quantum mechanical (QM) method(s) in a small region of interest embedded in an environment which is treated using a lower level approximate theory (e.g., classical molecular mechanics). With this decomposition, the total energy of the system is given by the sum of the quantum ( $E_{qm}$ ) and classical energies:

$$E = E_{qm}[\mathbf{r}, \mathbf{R}; \psi] + E_{mm}[\mathbf{r}, \mathbf{R}] \quad (8)$$

where  $\mathbf{r}$ ,  $\mathbf{R}$  represents the coordinates of QM and MM regions and  $\psi$  denotes the ground or excited electronic wavefunction of the QM region. The QM contribution can be further decomposed into internal and external

contributions as [47]:

$$E_{qm} = E_{qm}^{\text{int}}(\mathbf{r}; \psi) + E_{qm}^{\text{ext}}(\mathbf{r}, \mathbf{R}; \rho) \quad (9)$$

where the internal part  $E_{qm}^{\text{int}}(\mathbf{r}; \psi)$  is the gas phase energy expression and the external part contains the electrostatic interactions of the classical charges ( $Z_I$ ) of the MM region with the electron density ( $\rho$ ):

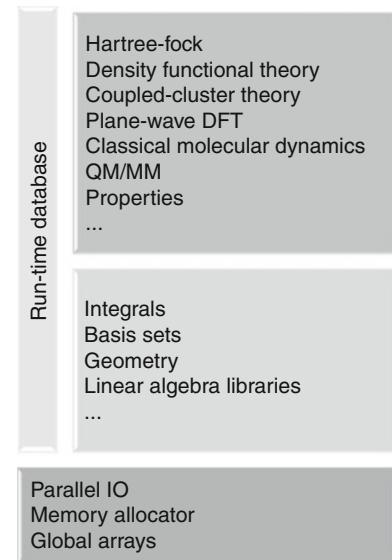
$$E_{qm}^{\text{ext}}(\mathbf{r}, \mathbf{R}; \rho) = \sum_I \int \frac{Z_I \rho(\mathbf{r}')}{|\mathbf{R}_I - \mathbf{r}'|} d\mathbf{r}' \quad (10)$$

All the classical interactions as well as the van der Waals and electrostatic interactions with the quantum region are subsumed in the classical energy term ( $E_{mm}$ ). Since a clean separation between the quantum and classical regions is not always possible, different schemes (e.g., link atoms, bond capping schemes) have to be utilized [13, 49]. The NWChem implementation supports two different treatments of link atoms – hydrogen and pseudo-carbon capping schemes.

The QM/MM module in NWChem is built as a top-level interface between the classical MD module and various QM modules and supports a wide variety of calculations like optimizations, dynamics, free energy calculations, to name a few. The scaling of QM/MM calculations is largely determined by the scaling of the underlying QM and MM implementations. We refer the reader to published review articles for further details regarding the implementation of this module [46, 47].

## Parallel Framework

NWChem uses the Global Array (GA) toolkit to scale and perform on parallel platforms [32, 33]. Figure 5 illustrates how the various modules are integrated into a single framework. Since the GA programming model includes message passing as a subset, the libraries within the toolkit can interoperate with MPI [27] for message passing. In other words, the programmer can make full use of MPI on GA and non-GA data. The GA toolkit comprises of the Memory Allocator (MA) which provides access to local memory, the Global Arrays (GA) which provides the necessary portable shared-memory programming tools, the Aggregate Remote Memory Copy Interface (ARMCI) [30] for portable and efficient remote memory copy operations (also known as one-sided communication) optimized for noncontiguous (strided, scatter/gather, I/O vector) data transfers,



**NWChem. Fig. 5** Schematic showing how the various modules in NWChem are integrated within a single framework along with the parallel libraries

and the Parallel I/O (ParIO) [31] tool to extend the nonuniform memory architecture model to disk.

The GA toolkit combines the best features of both the shared and distributed memory programming models. It implements a shared-memory programming model on distributed memory architectures in which data locality can be managed directly by the programmer. This is accomplished by explicit function calls that transfer data between a global address space (e.g., a distributed array) and local storage. This makes the GA model similar to distributed shared-memory (DSM) models that provide explicit acquire/release protocols. However, the GA model acknowledges that remote data is slower to access compared with local data and therefore allows data locality to be explicitly managed. By optimizing and selectively moving the data requested by the user, the GA model avoids problems like redundant data transfers. The GA model exposes the programmer to the hierarchical memory of modern high-performance computer systems and promotes data reuse and locality by recognizing the communication overhead for remote data transfer.

The GA library can be used in C, C++, Fortran 77, Fortran 90, and Python programs. The GA toolkit offers

support for both task and data parallelism. Task parallelism is supported via one-sided or non-collective copy operations that transfer data between global memory (distributed/shared array) and local memory. Each process can directly access data in a section of a Global Array that is logically assigned to it. Atomic operations are provided for synchronization and to assure correctness of an accumulate operation (e.g., floating-point addition that combines local and remote data) which is crucial to several electronic structure algorithms.

The data parallel computing model is supported through a set of functions that are collectively called that operate on either entire arrays or sections of Global Arrays. This set comprises BLAS-like operations (copy, additions, transpose, dot products, and matrix multiplication). Some are defined and supported for all array dimensions (e.g., addition), whereas operations like matrix multiplication are limited to two-dimensional arrays. Nevertheless, multiplication may be performed on two-dimensional subsets of higher dimensional arrays. The GA toolkit also includes interface to parallel linear algebra libraries like PeIGS and ScaLAPACK [42].

## Acknowledgments

The NWChem software and its documentation are developed at the EMSL located at PNNL, a multiprogram national laboratory, operated for the US Department of Energy by Battelle under the Contract Number DE-AC05-76RL01830. The code development is supported by the Department of Energy Office of Biological and Environmental Research, Office of Basic Energy Sciences, and the Office of Advanced Scientific Computing. Calculations reported here were performed using the Molecular Science Computing Capability (MSC) in the EMSL as well as resources of the National Energy Research Scientific Computing Center (NERSC) located at the Lawrence Berkeley Laboratory (LBL), California. The EMSL is funded by the Office of Biological and Environmental Research in the US Department of Energy.

## Acronyms

ARMCI: Aggregate Remote Memory Copy Interface  
 CC: Coupled Cluster Theory  
 CPMD: Car–Parrinello Molecular Dynamics

DFT: Density Functional Theory  
 DKH: Douglas–Kroll–Hess Approximation  
 EOMCC: Equation-of-Motion Coupled Cluster Theory  
 FFT: Fast Fourier Transform  
 GA: Global Arrays Toolkit  
 HF: Hartree–Fock Theory  
 MA: Memory Allocator  
 MBPT: Many-Body Perturbation Theory  
 MP: Müller–Plesset Perturbation Theory  
 MPI: Message Passing Interface  
 NWChem: Northwest Computational Chemistry Package  
 QM/MM: Quantum Mechanics–Molecular Mechanics  
 RHF: Restricted Hartree–Fock  
 ROHF: Restricted Open-Shell Hartree–Fock  
 SCF: Self-consistent Field Theory  
 TCE: Tensor Contraction Engine  
 TDDFT: Time-Dependent Density Functional Theory  
 UHF: Unrestricted Hartree–Fock  
 ZORA: Zeroth-Order Regular Approximation

## Related Entries

- ▶ [Amdahl's Law](#)
- ▶ [Anton, a Special-Purpose Molecular Simulation Machine](#)
- ▶ [BLAS \(Basic Linear Algebra Subprograms\)](#)
- ▶ [Car–Parrinello Method](#)
- ▶ [Computational Sciences](#)
- ▶ [Eigenvalue and Singular-Value Problems](#)
- ▶ [Exascale Computing](#)
- ▶ [FFT \(Fast Fourier Transform\)](#)
- ▶ [FFTW](#)
- ▶ [Global Arrays Parallel Programming Toolkit](#)
- ▶ [LAPACK](#)
- ▶ [NAMD \(NAnoscale Molecular Dynamics\)](#)
- ▶ [N-Body Computational Methods](#)
- ▶ [Non-Blocking Algorithms](#)
- ▶ [ScaLAPACK](#)
- ▶ [Spiral](#)

## Bibliographic Notes and Further Reading

References [2, 4, 10, 16, 18, 22, 25, 46] provide the interested reader with basic and advanced background to understand and utilize the various electronic structure methods implemented in NWChem.

## Bibliography

1. Aprà E, Bylaska EJ, Dean DJ, Fortunelli A, Gao F, Kristic PS, Wells JC, Windus T (2003) *Comp Mat Sci* 28:209
2. Aprà E, Harrison RJ, de Jong WA, Rendell AP, Tippuraj V, Xantheas SS, Olsen RM (2009) Liquid Water: Obtaining the right answer for the right reasons. In: Proceedings of the ACM/IEEE supercomputing 2009 conference, New York
3. Bartlett RJ, Musial M (2007) *Rev Mod Phys* 79:291
4. Bylaska E, Tsemekhman K, Govind N, Valiev M (2011) Large-scale plane-wave-based density functional theory: formalism, parallelization, and applications. In: Reimers JR (ed) Computational methods for large systems: electronic structure approaches for biotechnology and nanotechnology. Wiley, New York
5. Bylaska EJ, Valiev M, Kawai R, Weare JH (2002) *Comput Phys Comm* 143:11
6. Cizek J (1966) *J Chem Phys* 45:4256
7. Clarke LJ, Stich I, Payne M (1992) *Comput Phys Comm* 72:14
8. Comeau DC, Bartlett RJ (1993) *Chem Phys Lett* 207:414
9. Cornell WD, Cieplak P, Bayly CI, Gould IR, Merz KM, Ferguson DM, Spellmeyer DC, Fox T, Caldwell JW, Kollman PA (1996) *J Am Chem Soc* 118:2309
10. de Jong WA, Bylaska EJ, Govind N, Janssen CL, Kowalski K, Müller T, Nielsen IMB, van Dam HJJ, Veryazov V, Lindh R (2010) *Phys Chem Chem Phys* 12:6896
11. Essmann U, Perera L, Berkowitz ML, Darden T, Lee H, Pedersen LG (1995) *J Chem Phys* 103:8577
12. Foster IT, Tilson JL, Wagner AF, Shepard RL, Harrison RJ, Kendall RA, Littlefield RJ (1996) *J Comput Chem* 17:109
13. Gao JL, Truhlar DJ (2002) *Ann Rev Phys Chem* 53:467
14. Gygi F (2008) *IBM J Res Dev* 52:137
15. Harrison RJ, Guest MF, Kendall RA, Bernholdt DE, Wong AT, Stave M, Anchell JL, Hess AC, Littlefield RJ, Fann GI, Nieplocha J, Thomas GS, Elwood D, Tilson JL, Shepard RL, Wagner AF, Foster IT, Lusk E, Stevens R (1996) *J Comput Chem* 17:124
16. Helgaker T, Jorgensen P, Olsen J (2004) Molecular electronic-structure theory. Wiley, Chichester
17. Hirata S (2003) *J Phys Chem A* 107:9887
18. Janssen CL, Nielsen IMB (2008) Parallel computing in quantum chemistry. CRC Press/Taylor & Francis Group, Boca Raton
19. Kendall RA, Aprà E, Bernholdt DE, Bylaska EJ, Dupuis M, Fann GI, Harrison RJ, Ju J, Nichols JA, Nieplocha J, Straatsma TP, Windus TL, Wong AT (2000) *Comp Phys Comm* 128:260
20. Kobayashi R, Rendell AP (1997) *Chem Phys Lett* 265:1
21. Kohn W, Sham LJ (1965) *Phys Rev* 140:A1133
22. Kowalski K, Hammond JR, de Jong WA, Fan PD, Valiev M, Wang D, Govind N (2011) Large-scale plane-wave-based density functional theory: formalism, parallelization, and applications. In: Reimers JR (ed) Computational methods for large systems: electronic structure approaches for biotechnology and nanotechnology. Wiley, New York
23. Kowalski K, Krishnamoorthy S, Villa O, Hammond JR, Govind N (2010) *J Chem Phys* 132:154103
24. Kowalski K, Piecuch P (2001) *J Chem Phys* 115:643
25. Martin RM (2004) Electronic structure: basic theory and practical methods. Cambridge University Press, Cambridge
26. Marx D, Hutter J (2009) Ab initio molecular dynamics: basic theory and advanced methods. Cambridge University Press, Cambridge
27. Message Passing Interface Forum (2008) MPI: a message-passing interface standard. <http://www.mpi-forum.org>
28. Monkhorst HJ (1977) *Int J Quantum Chem* 421
29. Nelson JS, Plimpton SJ, Sears MP (1993) *Phys Rev B* 47:1765
30. Nieplocha J, Carpenter C (1999) ARMCI: a portable remote memory copy library for distributed Array Libraries and Compiler Run-time Systems. In: Proceedings of 3rd Workshop on Runtime Systems for parallel programming (RTSP) of international parallel processing symposium IPPS/SPDP iEi 99. Lecture notes in computer science, vol 1586. Springer, Berlin/Heidelberg/NewYork
31. Nieplocha J, Foster I, Kendall R (1998) *Int J Supercomput Appl High Perform Comput* 12:260
32. Nieplocha J, Harrison RJ (1997) *J Supercomput* 11:119
33. Nieplocha J, Palmer B, Tippuraj V, Krishnan M, Trease H, Aprà E (2006) *Int J High Perform Comput Appl* 20:203
34. Noga J, Bartlett RJ (1987) *J Chem Phys* 86:7041
35. Nogueira F, Marques M (2003) A primer in density functional theory (Lecture Notes in Physics) (v. 620). In: Fiolhais C, Nogueira F, Marques M. A. L (eds). Springer, Berlin/Heidelberg
36. Paldus J, Li XZ (1999) Critical assessment of coupled cluster method in quantum chemistry. *Adv Chem Phys* vol 110, Wiley, pp iEi 175
37. Parr RG, Yang W (1989) Density-functional theory of atoms and molecules. Oxford University Press, New York
38. Payne MC, Teter MP, Allan DC, Arias TA, Joannopoulos JD (1992) *Rev Mod Phys* 64:1045
39. Perdew JP, Schmidt K (2001) Large-scale plane-wave-based density functional theory: formalism, parallelization, and applications. In: Van Doren V, Van Alsenoy C, Geerlings P (eds) AIP conference proceedings: density functional theory and its application to materials, American Institute of Physics, vol 577
40. Purvis GD, Bartlett RJ (1982) *J Chem Phys* 76:1910
41. Raghavachari K, Trucks GW, Pople JA, Headgordon M (1989) *Chem Phys Letts* 157:479
42. Scalapack specifications can be found at [http://www.netlib.org/scalapack/scalapack\\_home.html](http://www.netlib.org/scalapack/scalapack_home.html)
43. Straatsma TP, McCammon JA (2001) *IBM Syst J* 40:328
44. Straatsma TP, Philippopoulos M, McCammon JA (2000) *Comp Phys Comm* 128:377
45. Szabo A, Ostlund NS (1996) Modern quantum chemistry. McGraw-Hill, New York
46. Valiev M, Bylaska EJ, Govind N, Kowalski K, Straatsma TP, van Dam HJJ, Wang D, Nieplocha J, Aprà E, Windus TL, de Jong WA (2010) *Comp Phys Comm* 181:1477
47. Valiev M, Garrett BC, Tsai MK, Kowalski K, Kathmann SM, Schenter GK, Dupuis M (2007) *J Chem Phys* 127:51102
48. Windus TL, Bylaska EJ, Andzelm J, Govind N (2009) *J Comp Theor Nano* 6:1297
49. Zhang YK, Liu HJ, Yang WT (2000) *J Chem Phys* 112:3483

# O

## Omega Calculator

► [Omega Test](#)

## Omega Library

► [Omega Test](#)

## Omega Project

► [Omega Test](#)

## Omega Test

DAVID WONNACOTT  
Haverford College, Haverford, PA, USA

### Synonyms

Omega calculator; Omega library; Omega project

### Definition

The Omega Test is an algorithm for dependence analysis that was developed by Bill Pugh's Omega Project in the 1990s to enable work on advanced loop optimizations. It is notable for its manipulation of constraints on integer variables to produce a representation of dependences that is more precise than distance/direction vectors, thereby enabling advanced analyses and transformations.

### Exact Instance-Based Array Dependence Information

Optimization of loop nests requires information about dependences that is both detailed and accurate. Even

for the restricted case in which array subscripts and loop bounds must be affine functions of loop indices, testing for the existence of a dependence is equivalent to testing satisfiability of a conjunction of affine constraints on integer variables, which is NP-complete [4]. Fortunately, in practice most array subscripts and loop bounds are even simpler, and low-complexity tests such as the GCD test and Banerjee's Inequalities produce accurate results in many cases.

In the late 1980s and 1990s, Paul Feautrier and Bill Pugh led independent efforts to improve array dependence analysis with techniques that were exact for the entire affine domain and fast for the common cases [2, 9, 11]. While some researchers felt exponential algorithms had no place in compilers, others saw value in the more precise information produced by these techniques. This approach, known variously as “constraint-based analysis,” the “polyhedral model,” or “instance-wise analysis,” remains an important tool for advanced restructuring compilers, some of which make use of the original code of Pugh or Feautrier. The remainder of this article reviews the work of Pugh's Omega Project; for information about other work on this area see the web sites [21, 22].

### Traditional Dependence Abstractions

To illustrate some of the issues involved in dependence analysis, consider the codes shown in Fig. 1 – each of these loops might write an element of array A that is read in a later iteration, so each is said to exhibit an inter-iteration flow dependence based on A, which inhibits parallel execution of the loop iterations. Earlier work on dependence analysis not only approximated some affine cases, but also reported only simple dependence abstractions, e.g., identifying loops that carry a dependence or classifying dependences with direction vectors or distance vectors. Regardless of its accuracy or domain, any test that produces this information must

report that both loops carry a dependence with a distance that is not known at compile time (assuming we lack information about  $n$  and  $k$ ).

A more detailed examination of the dependences of Fig. 1 reveals that the loops can be parallelized with different strategies, as illustrated in Fig. 2 (circles correspond to iterations, with lower values of  $i$  to the left, and each arrow indicates a flow dependence between iterations). For the illustrated case ( $n = 8$  and  $k = 3$ ), Fig. 1a has dependences from iteration 0 (which writes a value into  $A[0]$ ) to iteration 7 (which reads a value from  $A[0]$ ), from 1 to 6, from 2 to 5, and from 3 to 4; Fig. 1b has dependences from 0 to 3, 1 to 4, 2 to 5, 3 to 6, and 4 to 7. These patterns suggest that there may be some cases in which we can run some of the iterations in parallel, e.g., when  $n = 8$  and  $k = 3$  we could execute iterations 0–3 of Fig. 1a simultaneously, and iterations 0–2 of Fig. 1b simultaneously. However, conclusions about program transformation must be made about the general case.

## Dependence Relations

The Omega Test describes the dependences in terms of relations among tuples of integers. To retain correspondence with the program being analyzed, these integers may be identified as inputs (e.g., the source of a dependence), outputs (the sink), or symbolic constants. A finite set of dependences can be given as a list (union) of such tuple relations, e.g., for Fig. 2a as  $\{[0] \rightarrow [7]\} \cup \{[1] \rightarrow [6]\} \cup \{[2] \rightarrow [5]\} \cup \{[3] \rightarrow [4]\}$ . Integer variables and constraints allow a representation that is both concise (allowing  $\{[i] \rightarrow [7-i] \mid 0 \leq i \leq 3\}$

```
// Example 1a
for (i=0; i<n; i++)
 a A[i] = A[n-1-i]*B[i];

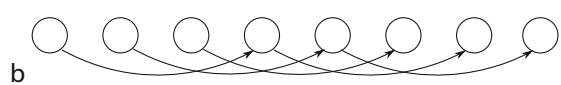
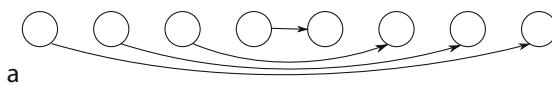
// Example 1b
for (i=0; i<n; i++)
 b A[i+k] = A[i]*B[i];
```

**Omega Test. Fig. 1** Simple loops exhibiting data dependences

for Fig. 2a) and general  $([i] \rightarrow [n-1-i] \mid 0 \leq i < \frac{n}{2})$  when  $n$  is not known, as in Fig. 1a).

These dependence relations follow directly from the application of the definition of data dependence to the program to be analyzed. A flow dependence exists between a write and a read (e.g., from iteration  $[i]$  to  $[i']$ ) when the subscript expressions are equal (e.g.,  $i = n-1-i'$  in Fig. 1a), the loop index variables are in bounds ( $0 \leq i < n \wedge 0 \leq i' < n$ ), and the dependence source precedes the sink ( $i < i'$ ). Combining these constraints produces  $\{[i] \rightarrow [i'] \mid i = n-1-i' \wedge 0 \leq i < n \wedge 0 \leq i' < n \wedge i < i'\}$  for 1a and  $\{[i] \rightarrow [i'] \mid i' = i+k \wedge 0 \leq i < n \wedge 0 \leq i' < n \wedge i < i'\}$  for 1b. These dependence relations are *exact* in the following sense: for any given values of the symbolic constants and loop indices, the constraints evaluate to true if and only if a dependence would actually exist. The constraint-manipulation algorithms of the Omega Library (see ▶Sect. Representation and Manipulation of Sets and Relations) can confirm that these relations are satisfiable and produce a simpler form, in these cases  $\{[i] \rightarrow [n-1-i] \mid 0 \leq i \wedge 2i \leq n-2\}$  ( $2i \leq n-2$  is equivalent to  $i < \frac{n}{2}$  but does not introduce division) and  $\{[i] \rightarrow [i+k] \mid 0 \leq i < n-k\}$ .

This representation of dependences extends naturally to nested loops with the introduction of additional variables in the input and output tuple – dependences among references nested in loops  $i$  and  $j$  would be represented as relations from  $[i,j]$  to  $[i',j']$ . An imperfect loop nest such as that shown in Fig. 3 raises the additional challenge of identifying the lexical position of each statement or inner loop (i.e., is it the first, second, etc. component within the loop?). This challenge can be met by interspersing (among the loop index values) constant values that give the statement position. If the  $i$  loop in Fig. 1 is the first (or only) statement in its function, and the update to  $A$  is the first (or only) statement in this loop, we could represent the dependence above as a relation from  $[1, i, 1] \rightarrow [1, i', 1]$ . For Fig. 3, the dependence from the write of  $A[i]$  to the read  $A[i-1]$  is  $\{[1, t, 2, i, 1] \rightarrow [1, t', 1, i', 1] \mid i'-1 = i \wedge t < t' \wedge 0 \leq t, t' < T \wedge 1 \leq i, i' < N-1\}$ .



**Omega Test. Fig. 2** Iteration spaces and flow dependences for Fig. 1, when  $n = 8$  and  $k = 3$

While it is possible to produce dependence distance/direction vectors from the dependence relations produced by the Omega Test, the results are often no more accurate than those produced by earlier tests such as a combination of Banerjee's inequalities and the GCD test. However, the added precision of the dependence relation can enable other forms of analysis or transformation if it is passed directly to later steps of the compiler.

## Advanced Program Analysis and Transformation

The detailed information present in a dependence relation can be used as a basis for a variety of program analyses, including analysis of the flow of values within a program's arrays, analysis of conditional dependences, and analysis of dependences that exist over only a subset of the iteration space. This information can then be used to drive a number of program transformation/optimization techniques on perfectly or imperfectly nested loops.

### Advanced Analysis Techniques: The Omega Test

Information about the iterations involved in a dependence can reveal opportunities for parallelization after transformations such as index set splitting or loop peeling. Note that any iterations that are not the source of a dependence arc (e.g., iterations 4–7 of Fig. 2a) can be executed concurrently after all other loop iterations have been completed; and those that are not a sink (iterations 0–3 in Fig. 2a) can be executed concurrently before any other has started. The Omega Library's ability to represent sets of integer tuples as well as relations can be used to show that there are never any iterations other than these nonsink and nonsource iterations for Fig. 1a, so this loop (unlike that in 1b) can be run in two sequential phases of parallel iterations.

The symbolic information present in a dependence relation can be used to detect conditional dependences. The flow dependence relation for Fig. 1b clearly implies that the flow dependence exists only when  $0 < k < n$ ; the Omega Library's projection and "gist" operations (see ▶Sect. The Gist Operation) can be used to extract this information. A compiler could introduce conditional parallelism (e.g., run the loop in parallel when  $n \leq k$ )

```

for (t = 0; t < T; t++) {
 for (i = 1; i < N-1; i++) {
 new[i] = (A[i-1] + A[i] + A[i+1]) * (1.0/3);
 }
 for (i = 1; i < N-1; i++) {
 A[i] = new[i];
 }
}

```

**Omega Test. Fig. 3** A set of "imperfectly nested" loops

or use additional analysis or queries to the programmer to attempt to prove no dependence ever exists.

By combining instance-wise information about memory aliasing with information about iterations that overwrite array elements, the Omega Test can produce information about the flow of values (termed "value-based dependence information" by the Omega Project). For example, the "memory-based" flow dependence shown above for Fig. 3 connects writes to reads in all subsequent iterations of the  $t$  loop – i.e., when  $t' > t$ . However, the value written in iteration  $t$  is overwritten in iteration  $t + 1$ , and thus never reaches any subsequent iteration. The value-based dependence relation includes this information, showing dependences only when  $t' = t + 1$ , as shown in ▶Sect. Disjunctive Normal Form. Value-based dependence information can be used for a variety of optimizations, e.g., to identify opportunities for parallelization after array privatization or expansion.

The Omega Project uses the term "Omega Test" for the production of symbolic instance-wise memory- or value-based dependence relations via the steps outlined above. More detail about the specific algorithms involved in dependence testing can be found in [16] as well as earlier publications by these authors.

### Iteration Space Transformation

Iteration space transformations can be viewed as relations between elements of the original and transformed iteration space, and thus represented and manipulated with the same infrastructure as dependence relations. Iteration spaces can be related to execution order via a simple rule such as "iterations are executed in lexicographical order," thereby turning iteration space transformation into a tool for transforming order of execution. A similar approach can be used to describe and transform the association of values to memory cells [19], though this has not been explored as thoroughly.

The “transformation as relation” framework unifies all unimodular loop transformations (e.g.,  $\{[i,j] \rightarrow [j,i]\}$  describes loop interchange) with a number of other transformations. Perhaps more importantly, this framework handles imperfect loop nests, which lie outside the domain of unimodular techniques. For example, consider the problem of fusing the inner loops of Fig. 3. Simple loop fusion can be accomplished by making  $A[i] = \text{new}[i]$  into the *second* statement in the *first*  $i$  loop, i.e.,  $\{[1,t,2,i,1] \rightarrow [1,t,1,i,2]\}$ . However, fusion without first aligning the loops is illegal;  $\{[1,t,2,i,1] \rightarrow [1,t,1,i+1,2]\}$  is a correct alignment and fusion (now element  $A[1]$  is overwritten in iteration  $i = 2$ , just after its last use in the computation of  $\text{new}[2]$ ). This transformation produces a loop nest with much better memory locality, since (barring interference) it moves each array into cache only once per time step rather than twice. Algorithms for finding useful program transformations in this framework, and for generating code from the resulting iteration space sets, are discussed in [5, 6, 10, 18].

### Iteration Space Slicing

Iteration space transformations can also be defined in terms of “iteration-space slicing”: the instance-wise analog of program slicing. This approach can express very general approaches to program transformation in terms that are relatively concise and clear.

To illustrate iteration space slicing, consider the challenge of parallelizing the code in Fig. 1b even when the dependence exists, as in Fig. 2b. Figure 4 shows the iteration space of this loop when  $n = 8$  and  $k = 3$  (as in Fig. 2b), together with the backward iteration space slice for iteration 7 (the set of other iterations that must be executed to produce the value in the target iteration). A similar backward slice for iteration 6 would include iterations 3 and 0.

It is possible to run Fig. 1b as  $k$  concurrent threads, each of which updates every  $k$ th element. A simple test could be used to identify and parallelize this specific

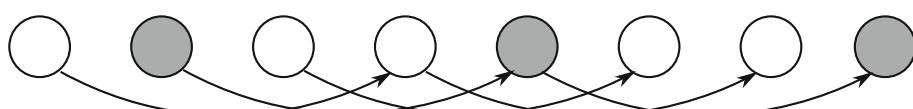
case (a one-dimensional loop with a single constant-distance dependence), but iteration space slicing provides a concise way to generalize this idea: find the backward slices to the iterations that are not the source of any dependence – if these slices do not intersect, they can be run concurrently to produce the result of the loop nest.

In some cases, it may be necessary to find the iteration space slice needed to compute one result given that another slice has already been completed. For example, consider the challenge of increasing the memory locality of Fig. 3 by more than the factor of two achieved in ►Sect. Iteration Space Transformation. Figure 5 shows the iteration space of a sample run of the original form of this code: each time step is shown as a column of circles representing the computations of  $\text{new}[i]$ , followed by a column of squares representing the copy into  $A[i]$ . The slice needed to compute  $A[1]$  is shown as dashed iterations surrounded by a dashed border; the marginal slice needed for  $A[2]$  given that  $A[1]$  has been computed is shown as shaded iterations. Executing this code as a series of sequential slices can greatly improve the memory locality (given a cache large enough to hold the results of a few slices, this can reduce memory traffic from  $O(T \cdot N)$  to  $O(N)$  in the absence of interference, producing dramatic speedups for slow memory systems [19]).

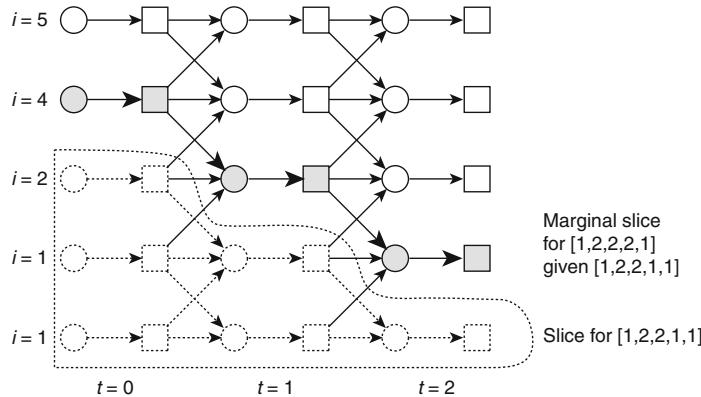
Algorithms for iteration space slicing, for producing transformations with it, and for generating code to execute slices are described in [12, 13, 19].

### Representation and Manipulation of Sets and Relations

The Omega Library (or simply “Omega” when in context) employs a number of techniques to transform the constraints extracted from programs in ►Sect. Exact Instance-Based Array Dependence Information into answers to the questions of ►Sect. Advanced Program



**Omega Test. Fig. 4** The backward iteration space slice for iteration 7 of Fig. 2b



**Omega Test. Fig. 5** Iteration space and two slices for Fig. 3,  $T = 3, N = 7$

**Analysis and Transformation.** The details of representation and manipulation depend on the nature of the constraints and the question being asked.

Most algorithms can be expressed either in terms of individual (in)equations or higher-order operations on relations and/or sets. For example, a compiler could construct each memory-based flow dependence relation by creating appropriate tuples for input and output variables and then examining loop bounds, subscript expressions, etc., to produce the appropriate constraints on the tuples' variables. Alternatively, it could first define a set for the iteration space  $I$  of each statement  $S$  ( $I_s$ ), and a relation  $M$  mapping loop iteration to array index for each array reference  $R$  ( $M_R$ ), and relations  $T$  for forward-in-time order for various loop depths, and then construct a flow dependence by combining these, i.e., the dependence from access  $x$  to  $y$  could be found by taking  $(M_x \bullet (M_y)^{-1}) \cap T_{xy}$  and restricting its domain to  $I_x$  and its range to  $I_y$ .

Some operations on relations or sets simply involve re-labeling of free variables, e.g., swapping the input and output tuples to find  $R^{-1}$  from a relation  $R$ . Others such as intersection ( $\cap$ ) involve matching up variables from input and output tuples, and possibly converting free variables to quantified variables; in the relational join and composition operations (for  $R_1 \bullet R_2$  or the equivalent  $R_2 \circ R_1$ ),  $R_1$ 's input tuple becomes the input tuple,  $R_2$ 's output tuple the output tuple, and  $R_1$ 's outputs and  $R_2$ 's inputs become existentially quantified variables, all constrained to the intersection of the constraints from  $R_1$  and  $R_2$ . Finally, some operations such as transitive closure are defined in Omega only as relational operators.

The remainder of this section gives an overview of the algorithms used in the Omega Library. It progresses from less-expressive to more-expressive constraint languages, roughly following the historical development of the algorithms of the Omega Library, and gives descriptions in terms of collections of (in)equations rather than relations or sets. For more detail on these algorithms or the relationship between high- and low-level operations, see the cited papers, or [7] for transitive closure algorithms.

## Conjunctions of Affine Equations and Inequations

The fundamental data structure of the Omega Library represents a conjunction of affine equations and inequations with integer coefficients (Omega does not currently represent disequations such as  $n \neq m$ , so hereafter the term inequation refers only to  $<$ ,  $\leq$ ,  $\geq$ , and  $>$  constraints, each of which is converted to  $\geq$  form internally). Each individual (in)equation is immediately reduced to lowest terms (e.g., turning  $3x + 9y + 21 = 0$  into  $x + 3y + 7 = 0$ ), and hash keys that ignore constant terms are used to quickly detect (and remove) simple redundancies (e.g.,  $a - 2b \geq 0$  and  $a - 2b + 10 \geq 0$  have the same hash key and the latter can be removed). The hashing system also detects (and replaces) pairs of inequations that are equivalent to an equation (e.g.,  $i \geq 5$  and  $i \leq 5$  are converted to  $i = 5$ ).

High-level operations for these conjunctions rely on lower-level operations on sets of (in)equations, such as variable elimination and redundancy detection. Variable elimination is also referred to as “projection” or

finding the “shadow” of a set or relation, since finding the two-dimensional shadow of a three-dimensional object can be thought of as an example of this operation. Note, however, that the shadow of the integer points in a set/relation is *not* always the same as the integer points in the shadow (as per the discussion of Fig. 6 below).

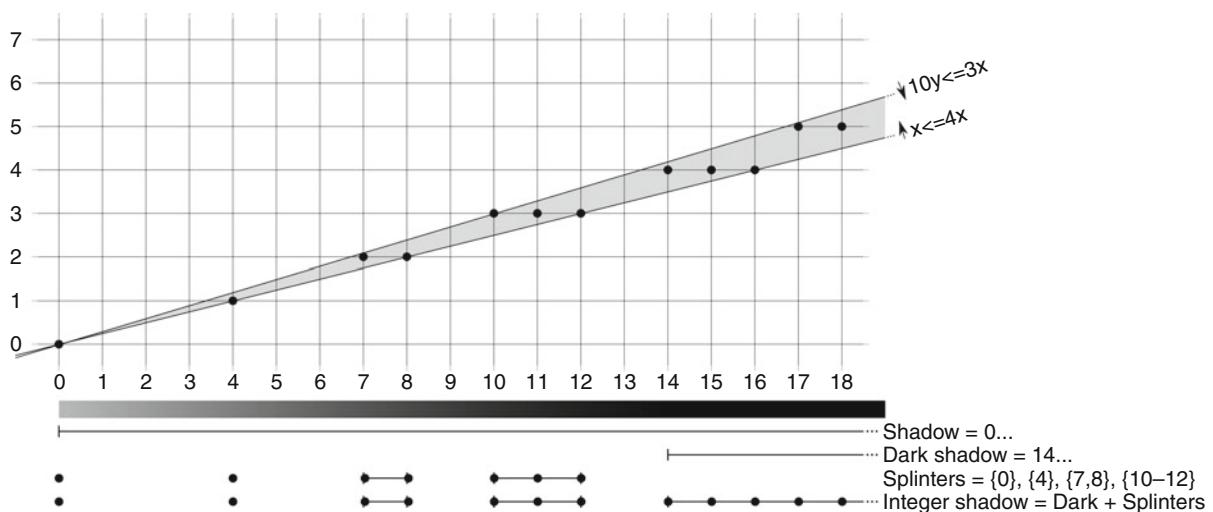
To illustrate Omega’s variable elimination and redundancy detection abilities, recall the dependence from Fig. 1b. Checking for possible dependence corresponds to existentially quantifying a dependence relation’s free variables (inputs, outputs, and symbolic constants), in this example checking the formula  $(\exists i, i', n, k : i' = i + k \wedge 0 \leq i < n \wedge 0 \leq i' < n \wedge i < i')$ . None of the six (in)equations of this formula is redundant with respect to any one other, so all are retained to the start of our query. Omega begins by using equations to eliminate variables: given  $i' = i + k$  it could replace all occurrences of  $i'$  with  $i + k$ , producing the formula  $(\exists i, n, k : 0 \leq i < n \wedge 0 \leq i + k < n \wedge 0 < k)$ . (Additional properties of integer arithmetic are used to eliminate an equation even when no variable has a unit coefficient, as discussed in [9].)

Upon running out of equations, Omega moves on to eliminate variables involved in inequations. Inequations are removed in several passes, with quick passes occurring first. Omega checks for variables that are bounded only on one side (i.e., have no lower bounds or no upper

bounds). Such variables cannot affect the satisfiability of the system, so they can be removed, along with all constraints on them – in this example,  $n$ , then  $k$ , and then  $i$  are removed in turn, producing an empty conjunction, i.e., True.

To illustrate some of the other algorithms used for affine conjunctions, we consider what would happen in this example if we knew  $n \leq 1,000$  (perhaps the loop in Fig. 1 is guarded by the test if  $n \leq 1,000$ ), i.e.,  $(\exists i, n, k : 0 \leq i < n \wedge 0 \leq i + k < n \wedge 0 < k \wedge n \leq 1,000)$ . In the absence of variables bounded on only one side, Omega examines groups of three inequalities to determine if any two contradict a third or make it redundant. In our running example,  $i + k < n$  and  $0 < k$  make  $i < n$  redundant, and  $0 \leq i$  and  $0 < k$  make  $0 \leq i + k$  redundant, reducing the query to  $(\exists i, n, k : 0 \leq i \wedge i + k < n \wedge 0 < k \wedge n \leq 1,000)$ .

Omega then moves on to its most general technique: an adaptation of Fourier’s method of variable elimination. Fourier’s method can be used to eliminate a variable  $v$  in a conjunction of affine inequations, by replacing the set of inequations on  $v$  with the set of all possible combinations of one upper and one lower bound on  $v$  – for example, replacing  $0 \leq i$  and  $i < n - k$  with  $0 < n - k$ . The original conjunction has a rational solution if and only if the new one does; the new system has one variable fewer, but may have many more inequations.



**Omega Test. Fig. 6** Shadow and dark shadow metaphors for integer variable elimination

The Omega Library uses an extension of Fourier’s method that preserves the presence of *integer* solutions. This process eliminates a variable by comparing the result of Fourier’s original technique (which produces the “shadow”) with a variant Pugh calls a “dark shadow” – a lower-dimensional conjunction that has integer solutions *only if* the original did. The set of integer solutions to the lower-dimensional system is the union of this dark shadow and the “splinter” solutions that lie within the shadow but outside the dark shadow [9].

Omega repeatedly eliminates variables to produce a trivially testable system with at most one variable, possibly producing  $(\exists n, k : 0 < n - k \wedge 0 < k \wedge n \leq 1,000)$  and then  $(\exists n : 0 < n \leq 1,000)$  in our running example. Since the query is now clearly satisfiable, the Omega Test concludes that there must be some values of the constants  $n$  and  $k$  for which a dependence exists between some iterations.

The example above, like most occurring during dependence analysis, does not illustrate the use of splintering to project integer solutions. Figure 6 shows an example for the inequations  $10y \leq 3x \wedge x \leq 4y$ , which have integer solutions for  $x = 0, 4, 7, 8, 10, 11, 12$  and  $x \geq 14$ . If Omega were to eliminate  $y$  during satisfiability testing of  $10y \leq 3x \wedge x \leq 4y \wedge x \leq 100$ , it would find that the dark shadow ( $14 \leq x \leq 100$ ) is satisfiable, and report True; if it were to eliminate  $y$  during satisfiability testing of  $10y \leq 3x \wedge x \leq 4y \wedge x \leq 12$ , it would find the rational shadow is satisfiable but the dark shadow is not, and proceed to explore the splinters.

The number of inequations can grow exponentially in the number of variables eliminated by Fourier’s method, and the need to compute splinters can only make matters worse. To control this problem in practice, Omega first eliminates variables that do not introduce splintering (projecting away  $x$  instead of  $y$  in Fig. 6, since the rational and dark shadows on the  $y$  axis are identical), and from these nonsplintering variables selects those that minimize the growth in the number of inequations. These steps appear to control the growth of inequalities during dependence analysis [9, 15].

## The Gist Operation

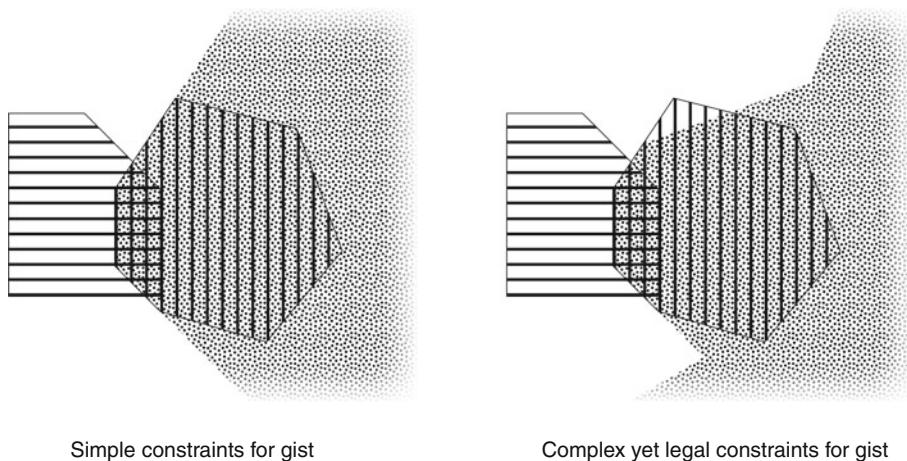
As noted in ►Sect. Advanced Analysis Techniques: The Omega Test, we may wish to classify the flow dependence of Fig. 1b as “conditional,” since there are values

of  $n$  and  $k$  for which no dependence exists and the iterations of the loop can be run in parallel. A simple definition of conditional dependence could be implemented by eliminating the loop index variables and identifying any nontautology as conditional. For this example,  $(\exists i, i' : i' = i + k \wedge 0 \leq i < n \wedge 0 \leq i' < n \wedge i < i')$ , or simply  $(0 < k < n)$ , is not a tautology, and this dependence would thus be marked as conditional.

Unfortunately, this simple definition would categorize as “conditional” almost any dependence inside a loop with a symbolic upper bound, such as that in 1a – projecting away  $i$  and  $i'$  leaves  $(2 \leq n)$ , but of course when  $n < 2$  there is no point in parallelizing the loop. To avoid such “false positives,” the Omega Test defines as conditional any dependence that does not exist under some conditions when we still wish to optimize the code, e.g., when a loop we wish to parallelize runs multiple iterations.

This definition makes use of the Omega Library’s “gist” operation as well as projection. Informally, the “gist” of one set of constraints  $p$  (i.e., those in which a dependence exists) given some other conditions  $q$  (i.e., those in which we care about the dependence) is the “interesting” information about  $p$  given that  $q$  is true. To give precise semantics while allowing flexibility in the processing of “interesting information,” gist is formally defined in terms of the values in the sets/relations being constrained, not the collection of constraints giving the bounds: “(gist  $p$  given  $q$ )” is any set/relation such that  $p \wedge q = (\text{gist } p \text{ given } q) \wedge q$ . Figure 7 gives a graphical illustration of the gist operation. Both examples illustrate, with a speckled region, the gist of a vertically striped region given a horizontally striped region. Both cases illustrate the formal definition of gist, as the speckled region overlaps the horizontally striped region in the same way the vertically striped region does. The option on the left illustrates the “usual” behavior that is produced by the algorithm described below: the gist is a superset of the original vertically striped region, with simple extensions of boundaries that defined that region. The possibility of results like that on the right allows flexibility in the definition of “simple” that is important when working with nonconvex sets.

When  $p$  and  $q$  are conjunctions of affine (in)equations, the Omega Library computes (gist  $p$  given  $q$ ) by first using the redundancy detection steps described above to check the (in)equations of  $p$  for



**Omega Test. Fig. 7** Examples of the gist operation: usual and unusual-but-legal options

redundancy with respect to  $p \wedge q$  (to check an (in)equation  $p_i$ , it uses  $p_i$ 's hash key to determine if any other single (in)equation might make  $p_i$  redundant; checks for variables that are not bounded in one direction that thus might show  $p_i$  cannot be redundant; and (if  $p_i$  is an inequation) compares  $p_i$  with pairs of inequations). If these “quick tests” fail to classify an (in)equation as redundant or not, Omega optionally performs the definitive but potentially expensive satisfiability test of a conjunction of all (in)equations from  $p \wedge q$  but with  $p_i$  negated. The conjunction of (in)equations of  $p$  that are not marked as redundant in  $p \wedge q$  is then returned as (gist  $p$  given  $q$ ).

Returning to the codes of Fig. 1, the flow dependence from Example 1b is marked as conditional, as  $(\text{gist} (0 < k < n) \text{ given } n > 1) = (0 < k < n)$ . Example 1a's dependence is not, since,  $(\text{gist} (2 \leq n) \text{ given } n > 1) = \text{True}$ . More details on the gist operation, such as the algorithms used when  $p$  and  $q$  are not both conjunctions of (in)equations, can be found in [14, 16, 20].

## Disjunctive Normal Form

Value-based dependence analysis, and certain cases of memory-based analysis (e.g., for programs with conditional statements) may produce constraints that are not simple conjunctions of equations and inequations.

In Fig. 3, the memory-based dependence from the write to  $A[i]$  to the read of  $A[i-1]$  is  $\{[1, t, 2, i, 1] \rightarrow [1, t', 1, i', 1] | i' - 1 = i \wedge 0 \leq t, t' < T \wedge 1 \leq i, i' < N - 1 \wedge t < t'\}$ , or more concisely  $\{[1, t, 2, i, 1] \rightarrow [1, t', 1, i+1, 1] | 0 \leq$

$t < t' < T \wedge 1 \leq i \leq N - 3\}$ . To describe the actual flow of values, the Omega Test must rule out all cases in which the value written to  $A[i]$  is overwritten before the read, i.e., any iteration  $\{[1, t'', 2, i'', 1]\}$  that writes to the same location (so  $i'' = i$ ) and occurs between the original write and the read (so  $t < t'' < t'$ ) during a valid execution of the statement ( $0 \leq t'' < T \wedge 1 \leq i'' \leq N - 1$ ); in other words it must represent the dependence relation

$$\begin{aligned} & \{ [1, t, 2, i, 1] \rightarrow [1, t', 1, i+1, 1] \mid 0 \leq t < t' < T \wedge 1 \\ & \quad \leq i \leq N-3 \wedge \\ & \quad \neg (\exists t'', i'' : t'' = i \wedge t < t'' < t' \wedge 0 \leq t'' < \\ & \quad T \wedge 1 \leq i'' < N-1) \} \end{aligned}$$

Omega handles such formulae by converting them into disjunctive normal form, i.e., a list of conjunctions of the form given in ▶Sect. Conjunction of Affine Equations and Inequations whose disjunction is equivalent to the original formula. Conversion to disjunctive normal form can increase the size of the formula exponentially; in practice, the explosion in formula size during dependence analysis is largely due to constraints that, once negated, contradict the positive constraints (on the first line of our example formula). This problem can be controlled by applying the rule  $a \wedge \neg b = a \wedge \neg(\text{gist } b \text{ given } a)$ . (Refer back to Fig. 7 for the intuition behind this rule – since the speckled region ( $\text{gist } b \text{ given } a$ ) must intersect  $a$  in exactly the way  $b$

did, so must the complement of the speckled region  $\neg(\text{gist } b \text{ given } a)$  intersect  $a$  as  $\neg b$  does.) This use of gist turns the formula above into

$$\{[1, t, 2, i, 1] \rightarrow [1, t', 1, i + 1, 1] \mid 0 \leq t < t' < T \wedge 1 \leq i \leq N - 3 \wedge \neg(t \leq t' - 2)\}.$$

The inequation  $t \leq t' - 2$  can then be negated to produce  $t' \leq t + 1$  rather than the disjunction of eight inequations that would have arisen without gist. In this context, Omega applies only the “quick tests” for gist, as there is no point in the optional full computation here. Techniques from ▶Sect. Conjunctions of Affine Equations and Inequations show our example formula is satisfiable and further simplify it to our final description of the value-based flow dependence:

$$\{[1, t, 2, i, 1] \rightarrow [1, t + 1, 1, i + 1, 1] \mid 0 \leq t < T - 2 \wedge 1 \leq i \leq N - 3\}.$$

The Omega Library can apply this conversion to disjunctive normal form recursively, allowing it to operate on the full language of “Presburger Arithmetic”: arbitrary formulae involving conjunction, disjunction, and negation of linear equations and inequations. Note that this logic has super-exponential complexity [3], and queries lacking the properties discussed above may well exhaust all available time or memory or produce integer overflow.

The Omega Test actually performs value-based analysis in a slightly different way from that described here, to “factor out” the effects of an overwrite on many different dependences (see [16, 20] for this and other details). However, the need for, and implementation of, negated constraints follows the principles stated here.

## Non-affine Terms and Uninterpreted Function Symbols

The constraint-manipulation techniques of the previous paragraphs can be used for memory-based, value-based, and conditional dependence analysis as long as the terms that are gathered from the program are affine functions of the loop bounds and symbolic constants. When some program term is not affine, many dependence tests simply report a maximally conservative approximation (“some dependence of some sort might exist here”) without further analysis. The Omega

Test can use two abilities of the Omega Library to produce more precise results, in many cases ruling out dependences despite the presence of nonaffine terms.

The simplest approach to handling nonaffine terms is to approximate the troublesome constraint but retain precise information about other constraints. The Omega Library provides a special constraint *UNKNOWN*, to be used in place of any constraint that cannot be represented exactly. Its presence indicates an approximation of some sort, but does not prevent Omega from manipulating other constraints or producing an exact final result in some cases (note that  $(\text{UNKNOWN} \vee \text{True}) = \text{True}$ ) and  $(\text{UNKNOWN} \wedge \text{False}) = \text{False}$ ); however, since the two unrepresentable constraints may arise from unrelated terms in a program,  $\neg\text{UNKNOWN}$  does not contradict *UNKNOWN*, and  $(\text{UNKNOWN} \wedge \neg\text{UNKNOWN} = \text{UNKNOWN})$ .

For example, suppose Fig. 1b updated  $A[i^*k]$  rather than  $A[i+k]$  — the constraint  $i' = i + k$  must be replaced with  $i' = i \cdot k$ , but of course this is outside the domain of the Omega Library. (Polynomial terms in dependence constraints were explored in [8] and by a number of authors not related to the Omega Project, but never released in Omega.) Thus, the Omega Test instead produces the relation  $\{[i] \rightarrow [i'] \mid \text{UNKNOWN} \wedge 0 \leq i < n \wedge 0 \leq i' < n \wedge i < i'\}$ .

A more detailed description can be created by replacing the unrepresentable *term* rather than the entire unrepresentable *constraint*. Omega records such a term as an *uninterpreted function symbol* — a term that indicates a value that may depend on some parameters. For example,  $i \cdot k$  depends only on  $i$  and  $k$ , and can thus be represented as  $f(k, i)$  for some function  $f$ . While *UNKNOWNs* cannot be combined in any informative way, two occurrences of the same function *can* be combined in any context in which their parameters can be proved to be equal, e.g.,  $(f(k, i) > x \wedge i = i' \wedge f(k', i') < x - 5 \wedge k' = k)$  can be simplified to False. This principle holds even when  $f$  is not a familiar mathematical function, most notably *any* expression  $e$  nested in loops  $i_1, i_2, i_3, \dots, i_n$  can be represented  $f_e(i_1, i_2, i_3, \dots, i_n)$ . The use of function symbols can improve the precision of dependence analysis, and when this does not eliminate a dependence, the Omega Test may be able to disprove it by including user assertions involving function symbols (if the programmer has provided them).

Presburger Arithmetic with uninterpreted function symbols is undecidable in general, and the Omega Library currently restricts function parameters to be a prefix of the input or output tuple of the relation. The Omega Test uses these tuples to represent the values of loop index variables source and sink of a dependence, and thus must approximate in cases in which a function is applied to some other values, in particular the values of loop indices at the time of an overwrite that kills a value-based dependence (i.e.,  $t''$  and  $i''$  in the relation in ►Sect. Disjunctive Normal Form). Further detail about the treatment of nonaffine constraints in the Omega Test, including empirical studies, can be found in [16].

## Current Status and Future Directions

The Omega Project's constraint manipulation algorithms are described in the aforesited references and the dissertations of Bill Pugh's students. Algorithms with relatively stable implementations were generally released as the Omega Library, a freely available code base which can still be found on the Internet (an open-source version containing updates from a number of Omega Library users can be found at <http://github.com/davewathaverford/the-omega-project/>). Notable omissions from the Omega Library code base include the code for iteration-space slicing, implementations of algorithms for polynomial constraints, some "corner cases" of the full Presburger satisfiability-testing algorithm, and any way of handling cases in which the core implementation of integer variable elimination produces extremely large coefficients. The Omega Library is generally distributed with the Omega Calculator, a text-based interface that allows convenient access to the operations of the library.

The constraint-based/polyhedral approach that was pioneered by Bill Pugh's Omega Project and by Paul Feautrier and his colleagues in France remains central to a number of ongoing compiler research projects. It is also a valuable tool in industrial compilers such as that produced by Reservoir Labs, Inc. For a discussion of the state-of-the-art before this work, see Michael Wolfe's "High-Performance Compilers for Parallel Computing" [17]; additional discussion of the instance-wise approach to reasoning about program transformations can be found in Jean-Francois Collard's "Reasoning

About Program Transformations: Imperative Programming and Flow of Data" [1].

A number of constraint-manipulation program analysis/transformation techniques and associated libraries have been developed since the release of the Omega Library. These typically contain more modern algorithms for a number of advanced functions of Omega, such as code generation, or more general implementations of underlying constraint algorithms, for example avoiding Omega's use of limited-range integers. As of the writing of this article, most do not support all of the techniques described in this article, notably:

- Algorithms that are (at least in principle) exact for integer variables
- Algorithms for the full domain of Presburger Arithmetic
- Separation of dependence analysis and transformation from the core constraint system via
  - An API allowing high-level relation/set operations and low-level operations
  - A text-based interface to allow easy exploration of the abilities of the system
- Special terms for communicating with the core constraint system about information not in its primary domain, e.g., function symbols
- Tagging of approximations to distinguish them from exact results
- Iteration space slicing (never released with Omega)

## Bibliography

1. Collard J-F (2002) Reasoning about program transformations. Springer, New York
2. Feautrier P (1988) Array expansion. In: ICS, St. Malo, pp 429–441
3. Fischer MJ, Rabin MO (1974) Super-exponential complexity of Presburger arithmetic. In: Karp RM (ed) Proceedings of the SIAM-AMS Symposium in Applied Mathematics, vol 7, pp 27–41, Providence, AMS
4. Garey MR, Johnson DS (1979) Computers and intractability: a guide to the theory of np-completeness. W.H. Freeman and Company, New York
5. Wayne Kelly and William Pugh. Finding legal reordering transformations using mappings. In *LCPC*, pp 107–124, 1994
6. Kelly W, Pugh W (1996) Minimizing communication while preserving parallelism. In: Proceedings of the 10th international conference on supercomputing, ICS '96, ACM, Philadelphia, New York, pp 52–60
7. Kelly W, Pugh W, Rosser E, Shpeisman T (1995) Transitive closure of infinite graphs and its applications. In: Proceedings of the 8th

- international workshop on languages and compilers for parallel computing. LCPC '95, Springer-Verlag, London, pp 126–140
8. Maslov V, Pugh W (1994) Simplifying polynomial constraints over integers to make dependence analysis more precise. In: Proceedings of the third joint international conference on vector and parallel processing: parallel processing, CONPAR 94-VAPP VI, Springer-Verlag, London, pp 737–748
  9. Pugh W (1991) The omega test: a fast and practical integer programming algorithm for dependence analysis. In: Proceedings of the 1991 ACM/IEEE conference on supercomputing, Supercomputing '91, Albuquerque, ACM, New York, pp 4–13
  10. Pugh W (1991) Uniform techniques for loop optimization. In: Proceedings of the 5th international conference on supercomputing, ICS '91, Cologne, ACM, New York, pp 341–352
  11. Pugh W (1992) A practical algorithm for exact array dependence analysis. Commun ACM, New York, 35(8):102–114
  12. Pugh W, Rosser E (1997) Iteration space slicing and its application to communication optimization. In: Proceedings of the 11th international conference on supercomputing, ICS '97, Vienna, ACM, New York, pp 221–228
  13. Pugh W, Rosser E (1999) Iteration space slicing for locality. In: Proceedings of the 12th international workshop on languages and compilers for parallel computing, LCPC '99, Springer-Verlag, London, pp 164–184
  14. Pugh W, Wonnacott D (1992) Eliminating false data dependences using the Omega test. In: SIG-PLAN Conference on Programming Language Design and Implementation, pp 140–151, San Francisco, California, June 1992
  15. Pugh W, Wonnacott D (1994) Experiences with constraint-based array dependence analysis. In: Principles and Practice of Constraint Programming, Second International Workshop, vol 874, Lecture Notes in Computer Science, pp 312–325. Springer, Berlin, May 1994. Also available as Tech. Report CS-TR-3371, Department of Computer Science, University of Maryland, College Park
  16. Pugh W, Wonnacott D (1998) Constraint-based array dependence analysis. ACM Trans Program Lang Syst 20(3):635–678
  17. Wolfe M (1996) High performance compilers for parallel computing. Addison-Wesley, Boston
  18. Wonnacott D (2000) Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In: Proceedings of the 14th international symposium on parallel and distributed processing, IEEE Computer Society, Washington, pp 171–180
  19. Wonnacott D (2000) Achieving scalable locality with time skewing. Int J Parallel Program 30(3):1–221
  20. Wonnacott DG (1995) Constraint-based array dependence analysis. PhD thesis, Department of Computer Science, The University of Maryland, August 1995. Available as [refftp://ftp.cs.umd.edu/pub/omega/dav-ewThesis/davewThesis.ps](http://ftp.cs.umd.edu/pub/omega/dav-ewThesis/davewThesis.ps)
  21. PolyLib - A library of polyhedral functions, Universite de Strasbourg, [icps.u-strasbg.fr/polylib](http://icps.u-strasbg.fr/polylib)
  22. Frameworks supporting the polyhedral model. Wikipedia, The Free Encyclopedia, [http://en.wikipedia.org/wiki/Frameworks\\_supporting\\_the\\_polyhedral\\_model](http://en.wikipedia.org/wiki/Frameworks_supporting_the_polyhedral_model)

## One-to-All Broadcast

►Broadcast

## Open Distributed Systems

►Peer-to-Peer

## OpenMP

BARBARA CHAPMAN, JAMES LA GRONE  
University of Houston, Houston, TX, USA

O

### Definition

OpenMP is an application programming interface for parallelizing sequential programs written in C, C++, and Fortran on shared-memory platforms. It provides a collection of compiler directives, a runtime library, and environment variables to enable programmers to specify the parallelism they desire to exploit in a program.

### Discussion

#### Introduction

The OpenMP Application Programming Interface (API) is a parallel programming model for shared-memory computer systems intended to provide a straightforward means of exploiting concurrency inherent in many algorithms. With the insertion of compiler directives, the programmer directs the compiler to parallelize portions of the code at a high level.

Originally designed to target loop-centric algorithms, version 3.0 of the API introduced the ability to define explicit tasks that may be executed concurrently. The OpenMP API is an agreement among hardware and software vendors that make up the OpenMP Architecture Review Board (ARB). The origins of the API are found in a set of compiler directives for writing parallel Fortran compiled by the Parallel Computing Forum (PCF) in the late 1980s. In 1997, the newly formed ARB introduced the first OpenMP specification for a set of

directives for use with Fortran and OpenMP compilers soon followed. Bindings for C and C++ have since been defined and the feature set has grown. Version 3.0, the most recent version, was ratified in 2008. The ARB is a non-profit corporation comprised of members from industry and academia, which owns the OpenMP brand and manages the OpenMP specification.

OpenMP does not require the programmer to explicitly decompose data and control flow to produce parallel computation. This fragmented style of programming is characterized by low-level programming to control the details of the concurrency. Rather, OpenMP allows the programmer to take a high-level view of parallelism and leave the details of the concurrency to the compiler. With the insertion of OpenMP directives, the programmer can specify what portions of sequential code should be executed in parallel. To a non-OpenMP compiler, the directives look like comments and are ignored. So by observing a few rules, one application can be both sequential and parallel, a very good quality that is quite useful in development and testing. Another benefit is the ability to incrementally apply OpenMP constructs to create a parallel program from existing sequential codes. Rather than start from scratch, the programmer can insert parallelism into a portion of the code and leave the rest sequential, repeating this process until the desired speedup is realized. This also means that to use OpenMP one need only learn a small set of constructs, not an entirely new language.

For a sample of how to use OpenMP, consider this fragment of C code that multiplies two matrices, *a* and *b*, and storing the result in a third matrix, *c*.

```
initialize_arrays(a,b,c,K,M,N);

#pragma omp parallel private(i,j,k)
shared (a,b,c)
{
 #pragma omp for schedule(auto)
 for (i = 0; i < N; i++)
 for (k = 0; k < K; k++)
 for (j = 0; j < M; j++)
 c[i][j] = c[i][j] + a[i][k]*b
 [k][j];
} /* end omp parallel */
```

As with any sequential program, execution starts with a single thread of control. The `#pragma omp`

identifies a line of code to be an OpenMP parallel directive, which specifies a structured block of code that should be treated as a parallel region. In this case the parallel region is enclosed with `{}`. When the initial thread of control reaches a parallel directive, it creates a team of threads, all of which will then execute the code in the parallel region. The initial thread becomes the team's master thread while the others are the slaves. The entire team will be available to share the work of the parallel region. This parallel construct includes a private clause and a shared clause. These are used to designate how variables are to be treated in the parallel region. In this case, the loop iteration variables will be treated local to individual threads while the matrices will be shared. Improperly characterized data can lead to incorrect results and errors, so this must be carefully considered. These and other clauses will be discussed later. This need to designate private and shared data is essentially what sets shared-memory programming from other methods.

The next directive identified by `#pragma omp` identifies the `for` construct, which designates that the execution of the immediately following loop is to be executed in parallel. Iterations of the loop will among the threads of the team in the enclosing parallel region according to the specified loop schedule. In this example, the `auto` schedule will allow the compiler and runtime to decide on the actually scheduling of the loop iterations.

## Overview of Features

The OpenMP API relies on directives, library routines, and environment variables for expressing the parallelism desired in a given program. Some of these were mentioned previously. A directive and its accompanying structured block form a *construct*. Directives may have various clauses associated to provide further information to the OpenMP implementation. As seen above, directives for C and C++ begin with `#pragma omp`. Directives in Fortran can begin with `!$omp`, but other options are available.

The fundamental `parallel` directive defines a `parallel region`, marking the structured block of code therein should be executed by multiple of threads. Without a parallel construct, the code will be executed sequentially. Each parallel construct designates whether data should be treated as private or shared. Data not

explicitly designated with a clause is shared by default. Since a parallel region is formed with a structured block, it must have only one entry point and one exit point. A program is *non-conforming* if it branches into or out of a parallel region.

Various constructs may be used with the parallel construct to designate worksharing, tasks, synchronization, and more. The worksharing constructs include the loop, sections, single, and workshare constructs. Each worksharing construct must bind to an active parallel region before its effects will be realized. Worksharing directives encountered by a single thread are ignored and the accompanying code will be executed sequentially, i.e., outside of a parallel region or inside a parallel region with a team of one thread. Clauses used with a parallel directive may be used to designate conditional use, the number of threads, data sharing properties, and reductions, some of which will be discussed below. Worksharing constructs end with an implicit barrier causing all threads to wait for the construct to complete. A worksharing construct that appears in a function or subroutine that is invoked from within a parallel region is called an *orphan* directive. Orphan directives may also be called from outside a parallel region allowing sequential or parallel execution of the function.

Worksharing constructs are used to define how the work in a block of code should be distributed across the executing threads. The loop construct (in C/C++ `for`, in Fortran `do`) is used to execute iterations of loops concurrently. The *sections* construct allows multiple blocks of code to be executed concurrently, each by a single thread. The *single* construct associates with the immediately following structured block for execution by a single, arbitrary thread. The *workshare* construct is supported only for directing the parallel execution of Fortran programs written using Fortran 90 array syntax.

Synchronization of threads in a team is achieved through another set of constructs. A barrier is a point of execution where threads must wait for all other threads in the current team before proceeding. While many constructs have implicit barriers, an explicit barrier is accomplished with the *barrier* directive. Barriers are used to avoid data race conditions. Access to shared data can be isolated to a critical region and protected using the *critical* directive or single assignment statements with the *atomic* directive. Explicit locks are also available for more flexible access of shared data.

Data in OpenMP programs is shared by the threads in the team by default. Any modification of shared data is guaranteed to be available to other threads after the next barrier or other synchronization point in the program. However, this is not always appropriate and some data will need to be local, or private, to each thread. A variable designated as *private* is replicated among the threads as a local copy. If the private variable needs to be initialized by the value of the corresponding variable immediately prior to the construct, the variable should be designated *firstprivate*. If the final value of a private variable is needed after the construct completes, the variable can be designated as *lastprivate*, whereby the final value of the variable inside the construct will be saved for the corresponding variable. Variables may also be designated as *reduction* variables with an associated operator. These variables will be used privately by each thread to perform its share of the work and then combined according to the operator as each thread completes its chunk of work.

OpenMP also specifies how a programmer may interact with the runtime environment. The API defines a set of internal control variables (ICVs) for controlling the execution at runtime. These include controlling the number of threads in a thread team, enabling nested parallelism, and specifying the scheduling of iterations in loop constructs. These ICVs may be accessed by setting environment variables or using runtime library routines. In some cases, the ICV may be set with a clause in the proper directive.

## Loop Parallelism

Since a significant amount of work occurs in loops, it is important to exploit any parallelism present. Any work that can be executed without depending on the outcome of other work can be executed concurrently. This kind of concurrency often occurs in the iterations of the loops. As long as an each iteration does not depend in some way on the outcome of a previous iteration, iterations can be executed concurrently. Existing loop-carried dependences can often be removed with some reorganization to reveal concurrency.

The most commonly used worksharing construct is the loop construct, which is marked with

```
#pragma omp for [clause[, ,] clause]...
for-loops
```

in C/C++ and

```
!$omp do [clause[, clause]...]
do-loops
[!$omp end do [nowait]]
```

in Fortran. This directs the implementation to distribute loop iterations among available threads. The number of iterations in the loop must be countable with an integer and use a fixed increment. This means only `for` loops in C/C++ and `do` loops in Fortran. Some rewriting of the loop may be necessary to expose parallelism. Clauses included can be used to prescribe the loop construct's data environment and loop scheduling as well as others.

Loop scheduling refers to how loop iterations are assigned to threads in the team. These are easily specified, using the clause

```
schedule(kind[, chunk_size])
```

The `kind` of clause can be `static`, `dynamic`, `guided`, `auto`, or `runtime`. If no loop schedule is specified, a static schedule will be used. This schedule will "chunk" loop iterations together in contiguous non-empty sets. Each chunk will be assigned to a thread in a round-robin fashion. The size of a chunk is set by the `chunk_size` value. If the chunk size is not specified, all chunks will be approximately the same size and at most one chunk will be assigned to each thread in the team. For varying workloads, the dynamic and guided schedules may be more appropriate. With the dynamic schedule, each thread will continually grab a chunk iterations until all chunks have been executed, using a chunk size of one if none is specified. Similar to this is the guided schedule except that the chunk size decreases as threads subsequent chunks. Then chunk size is set to be a proportion of the remaining iterations. The auto schedule allows the implementation to decide the schedule, which may be any possible distribution of iterations amongst the thread team. The runtime schedule defers the actual the scheduling of threads until execution, at which time the schedule and chunk size are read from environment variables via a library routine.

## Task Parallelism

More flexibility is often needed to exploit parallelism in code, especially where the amount of parallelism is unknown, as with recursive algorithms or processing pointer-based structures. When the task directive is encountered by a thread, the associated block of code will be made ready for execution at sometime in the future. There is no guarantee when the task will be executed. When the task is given to a thread for execution, it will be executed sequentially. The implementation is allowed to suspend the execution and resume at a later time. By default, the thread that begins a task must complete the task in its entirety. Any suspension in the execution of the task must be resumed on the thread on which it began. This is called a *tied* task. A task designated *untied* may resume suspended execution on any thread. There is no guarantee of the ordering of task execution or completion; tasks will be completed with the use of task synchronization constructs. Tasks may be suspended or resumed when a thread reaches a task scheduling point. Task scheduling points occur at the point immediately following explicit task creation, the end of the task construct, any barriers, and in the taskwait region. The `taskwait` directive causes the task to wait on all child tasks it has generated.

## Implementation

Implementations of OpenMP usually consists of an OpenMP-aware compiler and its runtime library (RTL). The compiler is responsible for recognizing and properly translating the program's OpenMP constructs while the RTL is responsible for thread creation and management at execution time. Most mainstream compilers that implement C, C++, and Fortran are capable of translating OpenMP. Any directives and RTL calls are translated along with the base-language program when the proper compiler options are used. Omission of these options will cause OpenMP directives to be ignored and result in a sequential program.

Some constructs are merely replaced with a call to the RTL, like the barrier and taskwait. Implementations typically convert the structured block marked by a parallel directive into a separate procedure in a process called outlining. Any references to shared variables are replaced with pointers to the variables' memory locations and passed to the outlined procedure as arguments. Private variables are local to the outlined procedure. Values for firstprivate variables are also passed as

arguments. A runtime library routine will fork the necessary threads and pass the outlined procedure to each thread with the needed arguments. Threads often sleep when not involved in an active team. The user can direct whether the threads will sleep or busy-wait when idle for performance considerations.

A parallel loop will likely have each thread execute a runtime routine to determine iteration chunks prior to executing the iterations, followed by a call to a barrier routine. A guided or dynamic schedule may have threads carry out more than one set of iterations by invoking a routine to get a remaining set of iterations. These schedules will have slightly more overhead because of this additional coordination. However, these schedules may execute the work fast enough to justify the added overhead.

A task can also be outlined to a procedure and variables passed as arguments similar to the parallel construct. Variables in tasks are firstprivate by default and their values must be saved at the time of task generation. Shared variables may be replaced with pointers to their memory locations as with the parallel construct. Each task is saved for execution at some time in the future, likely in a queue. The actual scheduling of the execution of the tasks is left to implementation dependence and varies greatly. OpenMP allow for work stealing, allowing a thread to take unfinished from other threads. By default, a task must be executed in its entirety by exactly one thread. If task is defined to be *untied*, it may finish execution on a different thread, making it a candidate for work stealing. As such, implementations may actually employ more than one schedule.

## Performance of OpenMP Programs

While OpenMP makes parallel programs easy to write, some extra work is usually necessary to allow complex programs to scale to high numbers of processors. Inherent in parallel programming is the presence of overhead, or work that would not otherwise be done in a sequential execution of the code. Each construct used will require some overhead to set up the parallel execution environment, so limiting their use to the presence of sufficient parallel work is advised. In addition to avoiding unnecessary overhead, the use of the memory hierarchy needs some careful attention.

Good performance of an OpenMP program begins with the optimizing of its sequential counterpart. This often involves reorganizing array accesses in loops and

removal of redundant code which may not be done by the compiler. Once a satisfactory version of the sequential code is obtained, OpenMP constructs may be inserted one at a time, ensuring with each insertion the resulting execution remains valid. This ability to add parallelism incrementally is one of OpenMP's strong points. One guiding principal in this process should be to keep all threads doing meaningful work as much as possible.

Use of the parallel construct is generally quite expensive as it must deal with the management of the thread teams. The number of parallel constructs should be few, enclosing as much code as possible. An *if* clause is available to create a parallel region only on the condition that sufficient parallel work can be achieved. Synchronization of threads is also expensive, so it is important to consider carefully when it is necessary and avoid it otherwise. Since workshare constructs have implied barriers, it is easy to have redundant barriers occur in code without being obvious. The *nowait* clause can be used to remove such implied barriers.

Since OpenMP is a shared-memory programming model some forethought as to how shared variables will be accessed is necessary. It is also prudent to minimize cache misses. Ensuring that array accesses in loops occur according to the base-language specification is one easy method. For instance, Fortran arrays are stored in memory in column-major order while C and C++ arrays are row-major. Nested loops should take this into account. Otherwise loops will likely access non-contiguous data on each subsequent iteration, causing a cache miss each time the array is accessed.

Cache coherent systems have a side effect known as *false sharing*, the interference among threads writing to different locations within the same cache line. A write by one thread will cause the system to notify the other caches of this modification causing them to update their copy of that entire cache line. Other threads accessing different locations in the cache line will have to wait for this update. While the threads in this case are not accessing shared data, the cache line is shared. This is a matter of performance, not correctness, but can prevent a program from scaling to a large number of threads.

In the following example, a `parallel do` construct exhibits false sharing. If *N* is the number of threads, each thread will execute one iteration of the loop given the chunk size of 1. When each thread updates one element of *b* it also invalidates the cache

line containing it. All other threads accessing that cache line will be affected and will need to get a new copy before their respective array elements are even modified.

```
!$OMP PARALLEL DO shared(b,N)
 schedule(static,1)
 DO i = 1,N,1
 b(i) = 4 * SIN(i*1.0)
 END DO
```

While array padding can help prevent false sharing, a better solution is to avoid the conditions in which false sharing will occur. When multiple threads are modifying shared data in the same cache line(s) in rapid succession, false sharing is likely to considerably impact performance.

Ensuring a balanced workload among the threads will also lead to better performance. The time spent in computation between synchronization points should be comparable. Use of the runtime schedule is helpful in experimenting to find the best loop schedule to balance the load in a given program.

## Correctness Considerations

It is actually quite easy to inadvertently introduce bugs into OpenMP programs. Bugs in shared-memory programming are usually very subtle and difficult to find. One such bug, the data race condition, involves the silent corruption of data during execution that is hard to detect as it may not manifest itself but on only a small fraction of the time. A data race condition occurs when multiple threads concurrently access the same shared data with at least one of the threads attempting to modify the data. Since there is no guarantee of the order in which the threads access data, this may lead to an incorrect result. Data race conditions may result from the lack of proper synchronization, like using a nowait incorrectly or neglecting to enclose such data access in a critical region.

It is usually good practice to minimize the sharing of variables since this can be problematic. By privatizing variables that do not need to be shared, data race conditions and other issues can be avoided. And, since variables are shared by default, overlooked variables may be unintentionally shared. For example, in a Fortran loop, the index variables are always treated as private, but they are only private in a parallel-for loop

in C. So in C/C++, the index variable of an inner loop will be shared if not explicitly privatized. Overlooked shared variables are also a source of the data race condition.

Another problem is the misunderstanding of which constructs have an implied barrier and which do not. In many cases, the single and the master constructs may be used for the same purpose. However, the former has an implied barrier and the latter does not. When using the master construct, it is important to know if an explicit barrier is needed afterwards. With the single, a nowait may be warranted to remove an unnecessary barrier.

Barriers may also lead to deadlock during execution if they are not used properly. An explicit barrier must be executed by all threads in the current thread team. If any thread does not reach the barrier, all the other threads will be waiting at the barrier for a thread that will never arrive. Another source of deadlock is the improper nesting of explicit locks. Deadlocked code will appear to be executing work but will never finish. These conditions are easily avoided with careful use of these constructs.

## Future Directions

The ARB is actively considering new strategies and features to add to or enhance the current API, deliberating on changes frequently being proposed by researchers. The ARB seeks to only make changes in the API that are generally acceptable to all vendors on all platforms. The challenge they face is to keep the API small enough to be relatively easy to use yet robust enough to offer sufficient expressivity for existing parallelism. Proposals under consideration include error handling mechanisms, support for performance tools, enhancing the current task interface, providing the mapping of work and data to threads, and the ability to exploit heterogeneous architectures such as GPUs and accelerators.

## Related Entries

- ▶ [Cilk](#)
- ▶ [Code Generation](#)
- ▶ [Loop Nest Parallelization](#)
- ▶ [OpenMP Profiling with ompP](#)
- ▶ [Shared-Memory Multiprocessors](#)

## Bibliographic Notes and Further Reading

The Parallel Computing Forum was an informal group comprised of representatives from industry and academia. Their parallel Fortran extensions were published in 1991 as a set of compiler directives for parallelizing loops. Though an official subcommittee for the American National Standards Institute (ANSI) was formed and a new standard drafted based on PCF in 1994, it was never adopted when interest dwindled and other parallel programming models gained attention. In the latter half of that decade, the OpenMP ARB formed and introduced the first version of OpenMP, based on the PCF work, in 1997. Later, a C/C++ specification was developed. These two specifications eventually were merged into a single document. Version 3.0 of the specification was ratified in 2008. Plans are underway in 2009 for versions 3.1 and 4.0.

In addition to the current specification of the API, the OpenMP website [1] provides information about the current specification, news about OpenMP, tutorials, a discussion forum, and links to more information about OpenMP.

The International Workshop on OpenMP ([www.iwomp.org](http://www.iwomp.org)) is an annual series of workshops for OpenMP research since 2005 and it is an excellent source for ongoing research by OpenMP users and developers and the ARB. Publications are available annually. Recent papers of interests include [task paper].

Books by Chapman [6] and Chandra [4] provide more detailed discussions of OpenMP, its use, and its implementation.

## Bibliography

1. OpenMP Application Program Interface, Ver. 3.0, May 2008. <http://www.openmp.org>
2. Ayguadé E et al (2009) A proposal to extend the OpenMP tasking model for heterogeneous architectures. In: Evolving openMP in an age of extreme parallelism; 5th International Workshop on OpenMP, IWOMP 2009, Dresden, Germany, vol 5568/2009. Lecture Notes in Computer Science, Springer, Berlin/Heidelberg, pp 154–167
3. Ayguadé E et al (2009) The design of OpenMP tasks. Parallel Distributed Syst. IEEE Trans, 20(3):404–418
4. Chandra R, Dagum L, Kohr D, Maydan D, McDonald J, Menon R (2000) Parallel programming in OpenMP. Morgan Kaufmann, San Francisco

5. Chapman B, Huang L, Bisconti E, Stotzer E, Shrivastava A, Gatherer A (2009) Implementing OpenMP on a high performance embedded multicore MPSoC on a high performance embedded multicore mpsoc. In: IPDPS '09: proceedings of the 2009 IEEE international symposium on parallel and distributed processing. IEEE Computer Society, Washington, DC, pp 1–8
6. Chapman B, Gabriele Jost, Ruud van der Pas. (2008) Using OpenMP: Portable Shared Memory Parallel Programming. MIT Press, Cambridge, MA London
7. Community (2009) cOMPunity – the community of OpenMP users. <http://www.compunity.org/>
8. Duran A, Corbalán J, Ayguadé E (2008) Evaluation of OpenMP task scheduling strategies. In: OpenMP in a new era of parallelism, vol 5004/2008. Lecture Notes in Computer Science, Springer, Heidelberg, pp 100–110
9. Itzkowitz M, Mazurov O, Copty N, Lin Y (2007) An OpenMP runtime API for profiling. Technical report, Sun Microsystems <http://developers.sun.com/solaris/articles/omp-api.html>
10. Parallel Computing Forum (1991) PCF parallel Fortran extensions. V5.0.ACML Sigplan Fortran Forum 10(3):1–57
11. Nanjegowda R, Hernandez O, Chapman B, Jin HH (2009) Scalability evaluation of barrier algorithms for OpenMP. In: Evolving OpenMP in an age of extreme parallelism, vol 5568/2009, IWOMP, Lecture Notes in Computer Science, Springer, Heidelberg, pp 42–52
12. Su E, Tian X, Girkar M, Haab G, Shah S, Petersen P (2002) Compiler support of the workqueuing execution model for Intel SMP architectures. In: The Fourth European Workshop on OpenMP. Rome, Italy
13. Terboven C, an Mey D, Sarholz S (2008) OpenMP on multicore architectures. In: IWOMP '07: proceedings of the 3rd international workshop on OpenMP, Springer, Berlin Heidelberg, pp 54–64

## OpenMP Profiling with OmpP

KARL FÜRLINGER

Ludwig-Maximilians-Universität München, Munich, Germany

### Synonyms

Collection of summary statistics; Performance analysis of openMP applications; Profiling

### Definition

Profiling is the process of gathering summary statistics about the execution of programs. Metrics of interest for OpenMP applications are the breakdown of time spent by threads in various sections of the program to

focus optimization efforts and the analysis of inefficiencies resulting from the parallel execution such as load imbalance and contention for locks and critical sections. Profiling is in contrast to tracing where individual time-stamped events are recorded and analyzed.

## Discussion

### Introduction

OpenMP is a successful approach for writing shared-memory thread-parallel programs. It currently enjoys a renewed interest in high-performance computing and other areas due to the shift toward multicore computing and it is fully supported by every major compiler. Extensions of OpenMP for programming GPU accelerators are also currently investigated [19] and under consideration for inclusion into a future version of the OpenMP specification.

As with any programming model, developers are interested in improving the efficiency of their codes and a number of performance tools are available to help in understanding the execution characteristics of applications. However, the lack of a standardized monitoring interface often renders the performance analysis process more involved, complex, and error-prone than it is for other programming models.

OpenMP is a language extension (and not a library, like MPI) and the compiler is responsible for transforming the program into threaded code. As a consequence, even a detailed analysis of the execution of the threads in the underlying low-level threading package is of limited use to understand the behavior on the OpenMP level. For example, compilers frequently transform parallel regions into outlined routines and the mapping of routine names (the machine's model of the execution) to the original source code (the user's model of the execution) is nontrivial. Vendor tools (paired with a compiler) can overcome this problem because they are aware of the transformations a compiler can employ and the naming scheme used for generating functions. Portable tools cannot have this kind of intricate knowledge for every compiler and OpenMP runtime and have to resort to other mechanisms, discussed later in this entry.

The rest of this entry is organized as follows: Section “►Profiling Tools Supplied by Vendors” discusses the vendor-provided solutions for OpenMP profiling, while Sect. “►Platform-Independent Monitoring”

describes how platform-independent monitoring can be achieved using a source-to-source translator called Opari. Section “►OpenMP Profiling with `ompP`” introduces the profiling tool `ompP` which is built on Opari to gather a variety of useful execution metrics for OpenMP applications. Sections “►Overheads Analysis” and “►Scalability Analysis” discuss the analysis of execution overheads and the scalability of applications based on the data delivered by `ompP`, respectively. Section “►Future Directions” concludes with an outlook on the path forward.

### Profiling Tools Supplied by Vendors

OpenMP is today supported by the compilers of every major computer vendor and the GNU compiler collection has OpenMP support since version 4.2. Most vendors also offer performance tools, such as the Intel Thread Profiler [17], Sun Studio [27], or Cray Pat [5], supporting users in analyzing the performance of their OpenMP codes.

The vendor tools mostly rely on statistical sampling and take advantage of the knowledge of the inner workings of the runtime and the transformations applied by the compiler. Typical performance issues that can be successfully detected by these tools are situations of insufficient parallelism, locking overheads, load imbalances, and in some cases memory performance issues such as false sharing.

A proposal for a platform-independent standard interface for profiling was created by SUN [18]. With this proposal, tools could register for notifications from the runtime and ask for it to supply the mapping from the runtime to user execution model. The proposal is the basis of SUN's own tool suite and is mostly used for statistical profiling there. The proposal is not part of the OpenMP specification but was accepted as an official white paper by the OpenMP Architecture Review Board (ARB). So far, acceptance of the proposal outside of SUN has been limited, however, and independent tool developers have to resort to other methods.

### Platform-Independent Monitoring

All approaches for monitoring OpenMP code in a platform-independent way in tools such as `ompP` [7], Vampir [4, 23], TAU [20, 26], KOJAK [28], and Scalasca [16] today use source code instrumentation. This approach has some shortcomings, for example, it

```

1 POMP_Parallel_fork [master]
2 #pragma omp parallel {
3 POMP_Parallel_begin [team]
4
5 /* user code in
6 parallel region */
7
8 POMP_Barrier_enter [team]
9 #pragma omp barrier
10 POMP_Barrier_exit [team]
11 POMP_Parallel_end [team]
12 }
13 POMP_Parallel_join [master]

```

The diagram illustrates the nesting of subregions for the OpenMP parallel construct. The main region is labeled "main". Inside it, there is a "body" region, which contains a "barr" (barrier) region, and finally an "exit" region.

**OpenMP Profiling with OmpP. Fig. 1** Instrumentation added by Opari for the OpenMP parallel construct. The original code is shown in boldface, the square brackets denote the threads that execute a particular POMP call. The right part shows the subregion nesting used by ompP

requires the user to recompile their code for instrumentation. However, it has been found in practice to work well for many applications and the usage of automated preprocessor scripts eases the burden for developers.

The monitoring approach used by all the aforementioned tools is based on a source code preprocessor called Opari [22]. Opari is a part of the KOJAK and Scalasca toolsets and it adds calls inside and around OpenMP pragmas according to the POMP specification [21]. An example source code fragment with Opari instrumentation is shown in Fig. 1: In this example, the user code is a simple parallel region (lines 2, 5, 6, and 12 in bold font represent the original user code), all other lines are added by Opari. POMP\_Parallel\_{fork|join} calls are placed on the outside of the parallel construct and POMP\_Parallel\_{begin|end} calls are placed as the first and last statements inside the parallel region, respectively. An additional explicit barrier and corresponding POMP\_Barrier\_{enter|exit} calls (lines 8–10) are placed toward the end of the parallel region as well. Doing so converts the implicit barrier at the end of the parallel region into an explicit barrier that can subsequently be monitored by tools, for example, to detect load imbalances among threads.

### OpenMP Profiling with ompP

ompP [7] is a profiling tool for OpenMP based on source-code instrumentation using Opari. ompP implements the POMP interface to monitor the execution of the instrumented application. An application is linked

```

#pragma omp parallel
{
 #pragma omp critical
 {
 sleep(1.0);
 }
}

```

**OpenMP Profiling with OmpP. Fig. 2** A simple OpenMP code fragment with a critical section

with ompP's profiling library and on program termination a profiling report is written to a file. Environment variables can be used to influence various settings of the data collection and reporting process.

Figure 2 shows an example OpenMP program fragment. The fragment contains a critical section inside a parallel region and the critical section only contains a call to `sleep()` for 1s. Figure 3 shows the corresponding profile for the critical section: this critical section was unnamed (for named critical sections the name would appear instead of "default") and that it is located at lines 34–37 in a file named "main.c." ompP's internal designation for this region is R00002.

The profile represents an execution with four threads. Each line displays data for a particular thread and the last line sums over all threads. A number of columns depict the actual measured profiling values. Column names that end in a capital "C" represent counts while columns ending with a capital "T" represent times. In this example, each thread executed the critical construct and that the total execution time

| R00002 main.c (34-37) (default) CRITICAL |       |       |       |        |       |
|------------------------------------------|-------|-------|-------|--------|-------|
| TID                                      | execT | execC | bodyT | enterT | exitT |
| 0                                        | 3.00  | 1     | 1.00  | 2.00   | 0.00  |
| 1                                        | 1.00  | 1     | 1.00  | 0.00   | 0.00  |
| 2                                        | 2.00  | 1     | 1.00  | 1.00   | 0.00  |
| 3                                        | 4.00  | 1     | 1.00  | 3.00   | 0.00  |
| SUM                                      | 10.01 | 4     | 4.00  | 6.00   | 0.00  |

**OpenMP Profiling with OmpP.** Fig. 3 An example of ompP’s profiling data for an OpenMP critical section. The body of this critical section contains only a call to `sleep(1.0)`

(`execT`) varies from 1.0 s to 4.0 s. `bodyT` is the time inside the body of the critical construct and it is 1.0 s corresponding to the time in `sleep()`, while `enterT` is the time (if any) threads have to wait before they can enter the critical section. In the example in Fig. 3, thread 1 was the first one to enter, then thread 2 entered after waiting 1.0 s, then thread 0 after waiting 2.0 s, and so on.

ompP reports timings and counts in the style shown in Fig. 3 for all OpenMP constructs. Table 1 shows the full table of constructs supported by ompP and the timing and count categories used in the profiling report. `execT` and `execC` are always reported and represent the entire construct. An additional breakdown is provided for some constructs, such as the `startupT` and `shutdownT` for the time required starting up and tearing down the threads of a parallel region, respectively. This breakdown is based on the subdivision of regions into subregions, as shown on the right-hand side of Fig. 1. If applicable, a region is thought of being composed of a subregion for entering (`enter`), exiting (`exit`), a main body (`body`), and a barrier (`barr`), and `main = enter + body + barr + exit` always holds.

In addition to the basic flat region profiles such as the ones shown in Fig. 3, ompP supports a number of more advanced features.

- ompP records the application’s call graph and the nesting of the OpenMP regions and computes inclusive and exclusive times from it. This allows performance data for the same region involved from different execution paths to be analyzed separately.
- It allows the monitoring of hardware performance counter values using PAPI [3, 24]. If a PAPI installation is available on a machine, ompP will include the necessary functionality to set up, start, and stop the counters for all OpenMP constructs.

- ompP computes the parallel coverage (i.e., the fraction of total execution time spent in parallel regions). According to Amdahl’s law [1] a high parallel coverage is quintessential to achieve satisfactory speedups and high parallel efficiency as the number of threads increases.
- Another feature of ompP is continuous and incremental profiling [6, 10]. Instead of capturing only a single profile at the end of the execution, this technique allows an attribution of profile features on the timeline axis without the overheads typically associated with tracing.
- A further recent technique aimed at providing temporal runtime information without storing full traces is capturing and analyzing the execution control flow of applications. The method described in [12, 13] shows that this can be achieved with low overhead and small modifications to the data collection process. The resulting flow graphs are designed for interactive exploration together with performance data in the form of timings and hardware performance counter statistics to determine phase-based and iterative behavior [11, 14].

## Overheads Analysis

Any parallel algorithm and the execution of a parallel program will usually involve some overheads limiting the scalability and the parallel efficiency. Typical sources of overheads are synchronization (the need to serialize the access to a shared resource, for example) or any time spent in setting up and destroying the threads of parallel execution.

An example of this is also shown in Fig. 3. `enterT` is the time threads have to wait to enter the critical section. From an application’s standpoint threads idle and do not do useful work on behalf of the application and hence `enterT` constitutes a form of synchronization overhead (since it arises due to the fact that the threads have to synchronize their activity). Considering the timings reported by ompP, a total of four different overhead classes can be identified.

**Synchronization:** Overheads that arise because threads need to coordinate their activity. An example is the waiting time to enter a critical section or to acquire a lock.

**OpenMP Profiling with OmpP. Table 1** All timing and count categories reported by `ompP` for various OpenMP constructs

| construct          | main  |       | enter  |          | body  |          |          |         |         | barr     | exit  |          |
|--------------------|-------|-------|--------|----------|-------|----------|----------|---------|---------|----------|-------|----------|
|                    | execT | execC | enterT | startUpT | bodyT | sectionT | sectionC | singleT | singleC | exitBarT | exitT | shutdwnT |
| MASTER             | •     | •     |        |          |       |          |          |         |         |          |       |          |
| ATOMIC             | •     | •     |        |          |       |          |          |         |         |          |       |          |
| BARRIER            | •     | •     |        |          |       |          |          |         |         |          |       |          |
| FLUSH              | •     | •     |        |          |       |          |          |         |         |          |       |          |
| USER REGION        | •     | •     |        |          |       |          |          |         |         |          |       |          |
| CRITICAL           | •     | •     | •      |          | •     |          |          |         |         |          | •     |          |
| LOCK               | •     | •     | •      |          | •     |          |          |         |         |          | •     |          |
| LOOP               | •     | •     |        |          | •     |          |          |         |         | •        |       |          |
| WORKSHARE          | •     | •     |        |          | •     |          |          |         |         | •        |       |          |
| SECTIONS           | •     | •     |        |          |       | •        | •        |         |         |          |       |          |
| SINGLE             | •     | •     |        |          |       |          |          | •       | •       |          | •     |          |
| PARALLEL           | •     | •     |        | •        | •     |          |          |         |         |          | •     |          |
| PARALLEL LOOP      | •     | •     |        | •        | •     |          |          |         |         |          | •     |          |
| PARALLEL SECTIONS  | •     | •     |        | •        | •     | •        | •        |         |         |          | •     |          |
| PARALLEL WORKSHARE | •     | •     |        | •        | •     |          |          |         |         |          | •     |          |

*Imbalance:* Overhead due to different amounts of work performed by threads and subsequent idle waiting time, for example, in work-sharing regions.

*Limited Parallelism:* This category represents overhead resulting from unparallelized or only partly parallelized regions of code. An example is the idle waiting time threads experience while one thread executes a `single` construct.

*Thread Management:* Time spent by the runtime system for managing the application's threads. That is, time for creation and destruction of threads in parallel regions and overhead incurred in critical sections and locks for signaling the lock or critical section as available.

[Table 2](#) shows the classification of the timings reported by `ompP` into no overhead (•) or one of the four overhead classes: S for synchronization overhead, I for imbalance, L for Limited Parallelism, and T for Thread management overhead.

`ompP`'s application profile includes an overhead analysis report which offers various interesting insights into where an application wastefully spends its time. The first part of the overhead analysis report shows the

total runtime (this corresponds to the duration of the overview section of the profiling report). Then the total number of parallel regions is reported (this includes combined worksharing-parallel regions) and then the parallel coverage is listed. The parallel coverage is the amount of execution time spent in parallel regions. A low parallel coverage limits the speedup that can be achieved by parallel execution.

The following two sections list overheads according to the classification scheme described in [Table 2](#) and discussed in detail in [8]. The difference between these two sections is the order in which parallel regions are listed and the way in which percentages are computed. In "Overheads wrt. whole program" the percentages are computed with respect to the total execution time (i.e.,  $\text{duration} \times \text{number of threads}$ ) and this section hence allows the easy identification of regions that cause high overheads for the whole application.

## Scalability Analysis

The scalability of an application or of individual parallel regions within an application can be analyzed in detail

**OpenMP Profiling with OmpP. Table 2** Overhead classification of the times reported by `ompP` into one of the four overhead classes. Synchronization (S), Load Imbalance (I), Thread Management (M), Limited Parallelism (L)

| <i>construct</i>   | <i>main</i>  |              | <i>enter</i>  |                 | <i>body</i>  |                 |                 |                | <i>barr</i>    | <i>exit</i>     |              |                 |
|--------------------|--------------|--------------|---------------|-----------------|--------------|-----------------|-----------------|----------------|----------------|-----------------|--------------|-----------------|
|                    | <i>execT</i> | <i>execC</i> | <i>enterT</i> | <i>startupT</i> | <i>bodyT</i> | <i>sectionT</i> | <i>sectionC</i> | <i>singleT</i> | <i>singleC</i> | <i>exitBarT</i> | <i>exitT</i> | <i>shutdwnT</i> |
| MASTER             | •            | •            |               |                 |              |                 |                 |                |                |                 |              |                 |
| ATOMIC             | S            | •            |               |                 |              |                 |                 |                |                |                 |              |                 |
| BARRIER            | S            | •            |               |                 |              |                 |                 |                |                |                 |              |                 |
| FLUSH              | S            | •            |               |                 |              |                 |                 |                |                |                 |              |                 |
| USER REGION        | •            | •            |               |                 |              |                 |                 |                |                |                 |              |                 |
| CRITICAL           | •            | •            | S             |                 | •            |                 |                 |                |                |                 | M            |                 |
| LOCK               | •            | •            | S             |                 | •            |                 |                 |                |                |                 | M            |                 |
| LOOP               | •            | •            |               |                 | •            |                 |                 |                |                | I               |              |                 |
| WORKSHARE          | •            | •            |               |                 | •            |                 |                 |                |                | I               |              |                 |
| SECTIONS           | •            | •            |               |                 | •            | •               |                 | •              |                | I/L             |              |                 |
| SINGLE             | •            | •            |               |                 | •            |                 |                 | •              | •              | I               |              |                 |
| PARALLEL           | •            | •            |               | M               | •            |                 |                 |                |                | I               |              | M               |
| PARALLEL LOOP      | •            | •            |               | M               | •            |                 |                 |                |                | I               |              | M               |
| PARALLEL SECTIONS  | •            | •            |               | M               | •            | •               |                 | •              |                | I/L             |              | M               |
| PARALLEL WORKSHARE | •            | •            |               | M               | •            |                 |                 |                |                | I               |              | M               |

---

----- `ompP` Overhead Analysis Report -----

---

Total runtime (wallclock) : 16.38 sec [4 threads]

Number of parallel regions : 10

Parallel coverage : 10.93 sec (66.73%)

Parallel regions sorted by wallclock time:

|        | Type     | Location          | Wallclock (%) |
|--------|----------|-------------------|---------------|
| R00004 | PARALLEL | muldoe.F (63-145) | 4.01 (24.45)  |
| R00007 | PARALLEL | muldoe.F (63-145) | 3.99 (24.34)  |

...

Overheads wrt. whole program:

| Total  | Ovhds (%) | = Synch (%)  | + Imbal (%) | + Limpar (%) | + Mgmt (%)  |             |
|--------|-----------|--------------|-------------|--------------|-------------|-------------|
| R00007 | 15.95     | 0.22 ( 0.34) | 0.00 (0.00) | 0.22 (0.34)  | 0.00 (0.00) | 0.00 (0.00) |
| R00004 | 16.02     | 0.16 ( 0.25) | 0.00 (0.00) | 0.16 (0.25)  | 0.00 (0.00) | 0.00 (0.00) |

by computing an overhead breakdown while varying the number of OpenMP threads for the execution.

Assuming a constant workload and increasing number of threads (i.e., a strong scaling study), one can plot scalability graphs like the one shown in Fig. 4. Here the number of threads is plotted on the horizontal axis and the total aggregated execution time (summed over all threads) is plotted on the vertical axis. The topmost line is derived from the overall wallclock execution time of an application. From this total time, one can subtract the four overhead classes (synchronization, load imbalance, thread management, and limited parallelism) for each thread count. A perfectly scaling code (linear decrease in the execution time the number of threads is increased) would result in a horizontal line in this type of scalability graph. A line inclining from the horizontal corresponds to imperfect scaling and a declining from the horizontal points toward superlinear speedup.

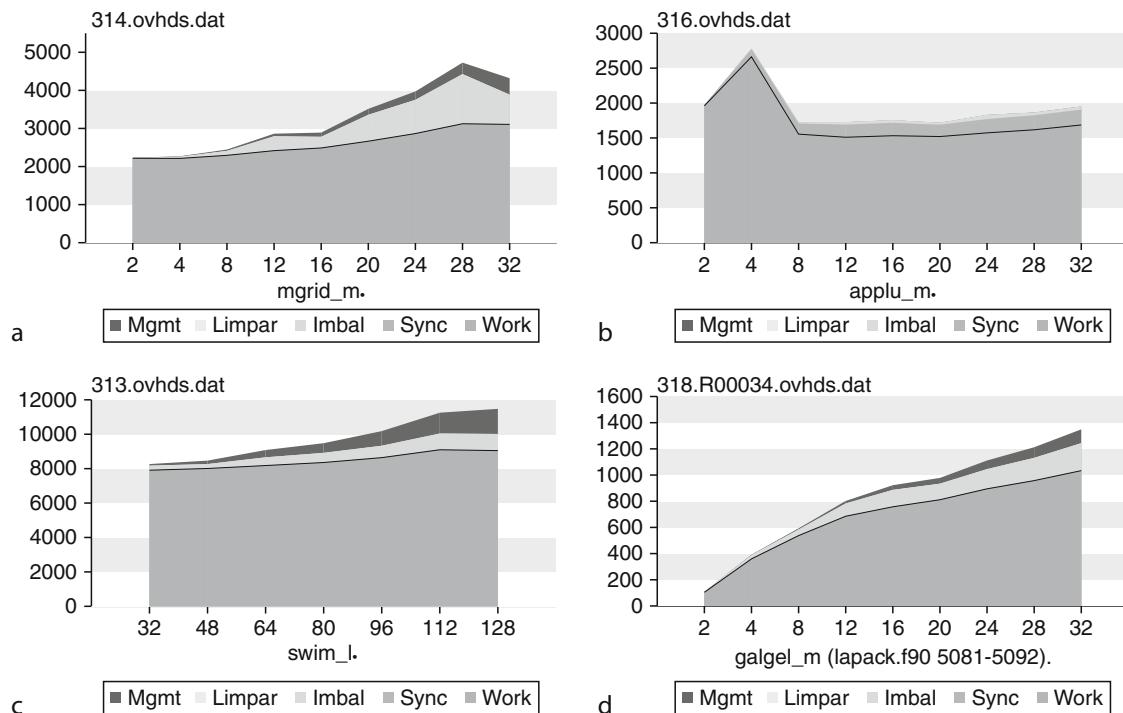
Figures 4a, b, and c are scalability graphs of full applications from the SPEC OpenMP benchmark suite ([2, 25]), corresponding to the “mgrid,” “applu” and

“swim” applications, respectively. Evidently the scaling behavior of these applications is widely different. A detailed discussion of the scaling properties of several of the SPEC OpenMP benchmarks can be found in [9].

In addition to scalability graphs for the whole application, `ompP` offers the option to analyze the scaling of individual parallel regions. As the workload characteristics of parallel regions within a program may vary considerably, so can their scalability characteristics and each region might have a different operating sweet spot in terms of computing throughput or energy efficiency. As an example, Fig. 4d shows an example for a scalability graph of an individual region of the application “galgel.”

## Future Directions

OpenMP is a continually evolving standard and new features pose new requirements and challenges for performance profiling. A recent major feature introduced with version 3.0 of the OpenMP specification is



**OpenMP Profiling with OmpP. Fig. 4** Scalability graphs for some of the applications of the SPEC OpenMP benchmark suite. Suffix \_m refers to the medium size benchmark while suffix \_l denotes large size benchmark. The horizontal axis denotes processor (thread) count and the vertical is the accumulated execution time (over all threads) in seconds

*tasking* – the ability to dynamically specify and enqueue small chunks of work for later execution.

A preliminary study [15] investigated the feasibility of extending the source code-based instrumentation approach to handle the new tasking-related constructs. The resulting modified version of `ompP` could successfully handle many common use cases for tasking. The notion of overheads had to be adapted, however, to reflect the fact that with tasking threads reaching the end of a parallel region are no longer idle but can in fact start fetching and executing tasks from the task pool.

One use case, that of *untied* tasks, proved to be a challenge. With untied tasks, an executing thread can stop and resume task execution at any time and in fact a different thread can pick up where the previous thread left off. These events are impossible to be observed purely based on source code instrumentation without a standardized callback mechanism that would notify a tool of such an occurrence.

To overcome these issues in particular and to simplify usage of `ompP` in general, tying together instrumentation sources from multiple layers, such as source code instrumentation, interposing on the native threading library, and incorporating the SUN whitepaper seems like a promising way for the path forward.

## Related Entries

- [Intel® Thread Profiler](#)
- [Scalasca](#)
- [TAU](#)
- [Vampir](#)

## Bibliography

1. Amdahl GM (2000) Validity of the single processor approach to achieving large scale computing capabilities. In: Readings in computer architecture. Morgan Kaufmann Publishers, San Francisco, pp 79–81 (Reprint of a work originally published in 1967)
2. Aslot V, Eigenmann R (2001) Performance characteristics of the SPEC OMP2001 benchmarks. SIGARCH Comput Archit News 29(5):31–40
3. Browne S, Dongarra J, Garner N, Ho G, Mucci PJ (2000) A portable programming interface for performance evaluation on modern processors. Int J High Perform Comput Appl 14(3): 189–204
4. Brustein H, Mohr B (2005) Performance analysis of large-scale OpenMP and hybrid MPI/OpenMP applications with Vampir NG. In: Proceedings of the first international workshop on OpenMP (IWOMP 2005), Eugene, May 2005
5. Using Cray performance analysis tools. <http://docs.cray.com/books/S-2376-51/S-2376-51.pdf>. Accessed 5 April 2011
6. Fürlinger K, Dongarra J (2007) On using incremental profiling for the performance analysis of shared memory parallel applications. In: Proceedings of the 13th international euro-par conference on parallel processing (Euro-Par '07). Lecture notes in computer science, vol 4641. Springer, August 2007, pp 62–71
7. Fürlinger K, Gerndt M (2005) `ompP`: a profiling tool for OpenMP. In: Proceedings of the first international workshop on OpenMP (IWOMP 2005), Eugene, May 2005
8. Fürlinger K, Gerndt M (2006) Analyzing overheads and scalability characteristics of OpenMP applications. In: Proceedings of the seventh international meeting on high performance computing for computational science (VECPAR'06). Lecture notes in computer science, vol 4395. Rio de Janeiro, pp 39–51
9. Fürlinger K, Gerndt M, Dongarra J (2007) Scalability analysis of the SPEC OpenMP benchmarks on large-scale shared memory multiprocessors. In: Proceedings of the 2007 international conference on computational science (ICCS 2007). Beijing, May 2007, pp 815–822
10. Fürlinger K, Moore S (2007) Continuous runtime profiling of OpenMP applications. In: Proceedings of the 2007 conference on parallel computing (PARCO 2007), volume 15 of Advances in parallel computing. IOS press, Sept 2007, pp 677–686
11. Fürlinger K, Moore S (2008) Detection and analysis of iterative behavior in parallel applications. In: Proceedings of the 2008 international conference on computational science (ICCS 2008). Lecture notes in computer science, vol 5103. Krakow, June 2008, pp 261–267
12. Fürlinger K, Moore S (2008) Visualizing the program execution control flow of OpenMP applications. In: Proceedings of the 4th international workshop on OpenMP (IWOMP 2008). Lecture notes in computer science, vol 5004. Purdue, May 2008, pp 181–190
13. Fürlinger K, Moore S (2009) Capturing and analyzing the execution control flow of OpenMP applications. Int J Parallel Program (IJPP) 37(3):266–276
14. Fürlinger K, Moore S (2010) Recording the control flow of parallel applications to determine iterative and phase-based behavior. J Future Gener Comput Syst 26(1)
15. Fürlinger K, Skinner D (2009) Performance profiling for OpenMP tasks. In: Proceedings of the 5th international workshop on OpenMP (IWOMP 2009), Dresden June 2009
16. Geimer M, Wolf F, Wylie BJN, Mohr B (2006) Scalable parallel trace-based performance analysis. In: Proceedings of the 13th European PVM/MPI Users' group meeting on recent advances in parallel virtual machine and message passing interface (EuroPVM/MPI 2006). Bonn, pp 303–312
17. Intel Corporation. Intel Thread Profiler. <http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/>. Accessed 05 April 2011
18. Itzkowitz M, Mazurov O, Copty N, Lin Y An OpenMP runtime API for profiling. (2007) Accepted by the OpenMP ARB as an official ARB White Paper available online at <http://www.community.org/futures/omp-api.html>. Accessed 05 April 2011

19. Lee S, Eigenmann R (2010) OpenMPC: Extended OpenMP programming and tuning for GPUs. In: Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis, SC'10, IEEE press
20. Malony AD, Shende SS (2000) Performance technology for complex parallel and distributed systems, Kluwer Academic Publishers, pp 37–46
21. Mohr B, Malony AD, Hoppe H-C, Schlimbach F, Haab G, Hoeflinger J, Shah S (2002) A performance monitoring interface for OpenMP. In: Proceedings of the fourth workshop on OpenMP (EWOMP 2002), Rome, Sept 2002
22. Mohr B, Malony AD, Shende SS, Wolf F (2001) Towards a performance tool interface for OpenMP: An approach based on directive rewriting. In: Proceedings of the third workshop on OpenMP (EWOMP'01), Sept 2001
23. Nagel WE, Arnold A, Weber M, Hoppe H-C, Solchenbach K (1996) VAMPIR: visualization and analysis of MPI resources. Supercomput 12(1):69–90
24. Terpstra D et al. PAPI web page: <http://icl.cs.utk.edu/papi/>. Accessed 05 April 2011
25. Saito H, Gaertner G, Jones WB, Eigenmann R, Iwashita H, Lieberman R, van Waveren GM, Whitney B (2002) Large system performance of SPEC OMP2001 benchmarks. In: Proceedings of the 2002 international symposium on high performance computing (ISHPC 2002). Springer, London, pp 370–379
26. Shende SS, Malony AD (2005) The TAU parallel performance system. International Journal of High Performance Computing Applications, ACTS Collection Special Issue, 2005
27. Oracle, Inc. Oracle Solaris Studio web site <http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html>. Accessed 05 April 2011
28. Wolf F, Mohr B (2003) Automatic performance analysis of hybrid MPI/OpenMP applications. In: Proceedings of the 11th Euromicro conference on parallel, distributed and network-based processing (PDP 2003), IEEE Computer Society Press, Feb 2003, pp 13–22

There is a long-standing and successful family of SHMEM APIs but there is no standard and implementations differ from each other in various subtle ways, hindering acceptance, portability, and in some cases, program correctness. We discuss the differences between SHMEM implementations and contrast SHMEM with other extant libraries supporting RMA semantics to provide motivation for a standards-based OpenSHMEM with the requisite breadth of functionality.

The Message Passing Interface (MPI) [1] is currently the most widely used communication model for large-scale simulation-based scientific parallel applications. Of these applications, a large number rely on two-sided communication mechanisms. Two-sided communication mechanisms require both sides of the exchange (source and destination) to actively participate, such as in MPI\_SEND and MPI\_RECV. While many algorithms benefit from the coupling of data transfer and synchronization that two-sided mechanisms provide, there exists a substantial number of algorithms that do not benefit from this coupling, and in fact may be hindered by the induced overhead of the synchronization.

The one-sided communication mechanisms are capable of decoupling the data transfer from the synchronization of the communication source and target. Remote memory access (RMA) is a one-sided communication mechanism that allows data to be transferred from one process memory space to another (remote) process memory space. The RMA operation is described entirely by one process (the active side) without the direct intervention of the other process (the passive side). Irregular communication patterns, in which data source and data target are not known *a priori* (as demonstrated in the GUPs [2] benchmark), often benefit from one-sided communication models such as RMA.

One of the most widely used one-sided communication models is SHMEM which stands for Symmetric Hierarchical MEMory access, so named for providing a view of distributed memory through the use of references to symmetric data storage on each sub-domain of a scalable system. While some sources (incorrectly) identify SHMEM as “SHared MEMory access,” it should be noted that SHMEM is unrelated to AT&T System V Shared Memory as implemented by the shmat, shmctl, shmget, etc. UNIX system calls. SHMEM [3] has a long

## OpenSHMEM - Toward a Unified RMA Model

STEPHEN W. POOLE<sup>1</sup>, OSCAR HERNANDEZ<sup>1</sup>, JEFFERY A. KUEHN<sup>1</sup>, GALEN M. SHIPMAN<sup>1</sup>, ANTHONY CURTIS<sup>2</sup>, KARL FEIND<sup>3</sup>

<sup>1</sup>Oak Ridge National Laboratory, Oak Ridge, TN, USA

<sup>2</sup>University of Houston, Houston, TX

<sup>3</sup>SGI, Eagan, MN, USA

### Definition

OpenSHMEM is a standards-based partitioned global address space (PGAS) one-sided communications library.

history as a parallel programming model, having been used extensively on a number of products since 1993, including Cray T3D, Cray X1E, the Cray XT3/4, SGI Origin, SGI Altix, clusters based on the Quadrics interconnect, and to a very limited extent, Infiniband-based clusters.

- History of SHMEM
  - Cray SHMEM
    - \* SHMEM first introduced by Cray Research Inc. in 1993 for Cray T3D
    - \* Cray is acquired by SGI in 1996
    - \* Cray is acquired by Tera in 2000 (MTA)
    - \* Platforms: Cray T3D, T3E, C90, J90, SV1, SV2, X1, X2, XE, XMT, XT
  - SGI SHMEM
    - \* SGI purchases Cray Research Inc. and SHMEM was integrated into SGI's Message Passing Toolkit (MPT)
    - \* SGI currently owns the rights to SHMEM and OpenSHMEM
    - \* Platforms: Origin, Altix 4700, Altix XE, Altix ICE, Altix UV
    - \* SGI was purchased by Rackable Systems in 2009
    - \* SGI and Open Source Software Solutions, Inc. (OSSS) signed a SHMEM trademark licensing agreement, in 2010
  - Other Implementations
    - \* Quadrics (Vega UK, Ltd.), Hewlett Packard, GPSHMEM, IBM, QLogic, Mellanox
    - \* University of Houston, University of Florida

Despite being supported by a variety of vendors there is no standard defining the SHMEM memory model or programming interface. Consistencies (where they exist) and extensions across the various implementations have been driven by the needs of an enthusiastic user community. The lack of a SHMEM standard has allowed each implementation to differ in both interface and semantics from vendor to vendor and even product line to product line, which has to this point limited broader acceptance. For an introduction to the SHMEM programming model, please see any of the following: [4, 5] or [6].

The SHMEM API and SHMEM in general encompass the following ideas:

- Single Program/Multiple Data (SPMD) style
- Characteristics: One-sided, data passing, RDMA, RMA, PGAS
- Put, Get, Atomic Updates (AMO) to reference remote memory or memories
- Remote memory/arrays are symmetric
- SHMEM decouples data transfer from synchronization
- Barriers and polling synchronization are used
- Collectives
- Explicit control over data transfer
- Low latency communications
- Library programming interface to PGAS
- SHMEM is a suitable under-layer for some PGAS implementations
- SHMEM is a viable alternative to MPI

In addition to SHMEM, a number of other parallel communication libraries currently support the one-sided communication model via RMA. These RMA APIs exhibit much commonality but also several core differences. In this entry, we will compare the most widely used RMA APIs for high performance computing to the SHMEM model.

The following RMA models will be considered in this entry:

*SGI SHMEM:* SGI SHMEM [7] (SGI-SHMEM) has one of the richest sets of RMA operations and is currently supported on the NUMAlink-based and Infiniband-based Altix product lines.

*Cray SHMEM - Unicos MP:* Cray SHMEM on the Unicos MP [8] (MP-SHMEM) is very similar to SGI-SHMEM and is currently supported on the XIE supercomputer.

*Cray SHMEM - Unicos LC:* Cray SHMEM on the Unicos LC [9] (LC-SHMEM) is a subset of SHMEM on the Unicos MP but lacks a number of key items such as full data-type support. Unicos LC is currently available on the Cray XT 3/4/5 supercomputers.

*Quadrics SHMEM:* Quadrics SHMEM [10] (Q-SHMEM) supports most of the communication mechanisms available in Unicos MP but lacks a number of key items that enhance usability.

**Cyclops-64 SHMEM:** Cyclops-64 SHMEM [11] (C64-SHMEM) is a SHMEM API which supports the Cyclops-64 architecture. Most of the core features of Cray SHMEM are available with some additional interfaces specific to the Cyclops-64 architecture.

**GASNet:** GASNet [12] is a lower level interface with minimal RMA support intended to be used as a communication mechanism for other parallel programming models and environments. As such, it is missing many of the usability features of the SHMEM family of RMA APIs. GASNet also differs from the other communication models due to its unique support for active messages.

**ARMCI:** ARMCI [13] is another lower level interface designed to support higher level protocols and programming models such as Global Arrays.

**MPI-2:** MPI-2 [14] provides a one-sided interface that has not been widely adopted for a number of reasons [15] but was intended to provide a portable alternative to SHMEM and other one-sided communication models.

In order to effectively compare these RMA APIs, we will define the following areas of functionality:

**Symmetric Objects:** Symmetric objects provide a mechanism to address remote variables using the address of the corresponding local variable.

**Remote Write:** Put operations across a number of primitive data-types as well as contiguous, strided, or indexed memory locations

**Remote Read:** Get operations across a number of primitive data-types as well as contiguous, strided, or indexed memory locations

**Synchronization:** Two basic types:

- Task Synchronization – synchronization operation between tasks, such as barriers and locks
- Memory Synchronization – memory polling, and ordering of remote and local memory reads and writes

**Atomic Memory Operations:** Compare/Mask Swap, fetch-and-add, increment

**Reductions:** Reductions across a number of primitive data-types and reduction operations such as and, or, min, and max

**Collective Communication:** Broadcast, collect, and fcollect operations

## Symmetric Objects

In SGI-SHMEM, MP-SHMEM, LC-SHMEM, and Q-SHMEM some data objects are said to be “symmetric” across multiple PEs. For a single-executable program, launched on multiple PEs, all statically allocated objects are symmetric across all PEs. Additionally, objects allocated dynamically via symmetric allocation (by calling SHMEM’s `shmalloc()` collectively) are also symmetric across all PEs. These symmetric data objects greatly simplify RMA operations by allowing the active side to initiate a remote operation on the passive side by specifying the address of the local symmetric data object. The RMA operation proceeds as if the address specified were the address of the passive side’s symmetric data object. In the absence of symmetric data objects, the address of the passive side’s data object must be communicated to the active side for use in the RMA (Remote) operation. In SHMEM, such objects are known as asymmetric objects. Asymmetric objects include stack variables as well as those objects dynamically allocated by nonsymmetric means (e.g., `malloc()`). Care must be taken in referencing asymmetric objects as the local (active side) address calculation will likely not result in the correct address on the remote (passive) side, though some implementations will not fail if asymmetric addresses strictly refer to local (active side) objects, e.g., the source for a PUT or the target for a GET. *Remote access to asymmetric objects is an extension that is not supported by SGI’s SHMEM or OpenSHMEM V1.0*

Symmetric data objects are not supported on the other RMA (Remote) APIs. GASNet requires that the active side specify the address of the data object on the remote side. ARMCI provides a collective memory allocation mechanism that allows each process to allocate memory locally and share the memory location with its peers. While convenient, this is no different from allocating memory locally and gathering the addresses from other PEs. SHMEM provides similar functionality via `shmalloc` with one difference, the memory allocated in `shmalloc` is symmetric (the Symmetric Heap). This allows RMA (Remote) operations to the remote memory by simply specifying addresses within the local symmetric memory. All SHMEM implementations discussed in this entry provide symmetric memory allocation. MPI-2 uses a hybrid approach by requiring processes to participate in the creation of a “window” for

RMA (Remote) operations. The active side can then specify the target's location for the RMA (Remote) operation via a displacement in the window.

It is important to note that symmetric data objects require either special hardware, compiler, or library support. The use of the address of a local data object in an RMA (Remote) operation to a remote PE indicates that the data object is symmetric. A (pre-)compiler can detect this usage and take appropriate action to ensure that the remote data object is properly addressed. SGI-SHMEM, MP-SHMEM, LC-SHMEM, and Q-SHMEM support symmetric data objects, asymmetric data objects, and symmetric memory allocation. C64-SHMEM supports only asymmetric data objects and symmetric memory allocation.

Code listing 1 (a slightly modified example from [8]) illustrates the use of symmetric data objects. Note the declaration of the “static short target” array and its use as the remote destination in `shmem_short_put`. The use of the “static” keyword results in the target array being symmetric on all PEs in the program. Each PE is able to transfer data to the target array by simply specifying the local address of the symmetric data object which is to receive the data. This aids programmability as the address of the target all PEs other than 0 need not be exchanged with the active side (PE 0) prior to the RMA (Remote memory operation) – an important reduction network data-flow, particularly for large systems with irregular data flows. Conversely, the declaration of the “short source” array is asymmetric. Because the put handles the references to the source array only on the active (local) side, the asymmetric source object is handled correctly. Note that C64-SHMEM does not support this mechanism and relies on the use of symmetric memory allocation. Other code examples are included in the examples section.

## Remote Write/Read

Remote Read/Write operations are the basic building blocks of any RMA API. Generally, these are referred to as put/get operations, and in the most basic form allow one processing element (PE) to transfer data from a local memory location to a remote memory location belonging to another PE, or vice versa.

*Remote Write:* In a Remote Write operation (PUT), the initiating (active side) PE is the *source* and the

remote (passive side) PE is the *target*. The active side PE specifies the local (active side) PE's (source) memory from which the data will be sent, and the remote (passive side) PE's (target) memory to which the data will be written. MP-SHMEM, LC-SHMEM, Q-SHMEM, and C64-SHMEM all provide the semantic that after `shmem_put` returns, the data specified for transfer has been buffered; this local completion on the active side does not equate to remote completion of the put on the passive side. LC-SHMEM, Q-SHMEM, and C64-SHMEM also provide a non-blocking PUT operation `shmem_put_nb` which may return before the data is buffered for transfer. Local completion of non-blocking remote (RMA) writes are guaranteed either when `shmem_test_nb` returns success or `shmem_wait_nb` returns. Unlike MPI requests, it is invalid to wait on a non-blocking operation after test returns success.

*Remote Read:* In a Remote Read operation (GET), the initiating (active side) PE is the *target* and the remote (passive side) PE is the *source*. The active side PE specifies the remote (passive side) PE's (source) memory from which the data will be retrieved, and the local (active side) PE's (target) memory to which the data will be written. In MP-SHMEM, LC-SHMEM, Q-SHMEM, and C64-SHMEM `shmem_get` returns after data has been delivered to the initiator PE's memory. LC-SHMEM, Q-SHMEM, and C64-SHMEM provide a non-blocking GET operation `shmem_get_nb` which does not block until the data specified for transfer has been delivered to the active side. Local completion of non-blocking RMA (Remote) Read operations is guaranteed either when `shmem_test_nb` (or `shmem_poll_nb` on C64-SHMEM) returns success or `shmem_wait_nb` returns.

**Table 2** details the primitive data-types supported by Remote Read/Write. Not surprisingly, ARMCI and GASNet only support byte level transfers as they are lower level APIs meant to provide a basis for higher level languages or APIs. The 16 bit, 4 byte, etc., “data-types” are simply syntactic sugar for the generalized byte level data-type supported by all APIs. None of the SHMEM libraries directly support unsigned integers or higher precision types beyond long long and double. Unsigned

```

#include <mpp/shmem.h>
#define SIZE 16
int
main(int argc, char* argv[])
{
 short source[SIZE]; /* local, not symmetric */
 static short target[SIZE]; /* static makes it symmetric */
 int i;
 int num_pe = atoi(argv[1]); /* no. of PEs taken from command-line */

 start_pes(num_pe);

 if (_my_pe(0) == 0) {
 /* initialize array */
 for(i = 0; i < SIZE; i++)
 source[i] = i;

 /* put "size" words into target on each PE */
 for(i = 1; i < num_pe; i++)
 shmem_short_put(target, source, SIZE, i);
 }
 shmem_barrier_all(); /* sync sender and receiver */
 if (_my_pe() != 0) {
 printf("target on PE %d is \t", _my_pe());
 for(i = 0; i < SIZE; i++)
 printf("%hd", target[i]);
 printf("\n");
 }
 shmem_barrier_all(); /* sync before exiting */
 return 0;
}

```

OpenSHMEM - Toward a Unified RMA Model. Fig. 1 RMA (Remote operation) using a symmetric data object

OpenSHMEM - Toward a Unified RMA Model. Table 1 Symmetric data objects and memory allocation

| Type                        | SGI | MP  | LC  | Q   | C64 | GASNet | ARMCI | MPI-2 |
|-----------------------------|-----|-----|-----|-----|-----|--------|-------|-------|
| Symmetric Data Objects      | Yes | Yes | Yes | Yes | No  | No     | No    | No    |
| Symmetric Memory Allocation | Yes | Yes | Yes | Yes | Yes | No     | No    | No    |

types can be used in put/get operations by using the appropriate byte size transfer at the cost of convenience.

In addition to data-type support, these RMA (Remote) APIs offer different access patterns for PUT/GET operations.

*Contiguous Memory Access:* Contiguous memory is specified by the active side as both the source and destination for a PUT or GET operation.

*Strided:* Strided access allows transfers of elements of a memory region with a stride between them. This is a compact representation suitable for regular data layouts such as multi-dimensional arrays.

*Indexed:* This access pattern provides more flexibility than strided access as there is no requirement for a constant stride. This added flexibility requires that

an index array be used to describe the elements within the source and target arrays, and is therefore less compact than the strided data description.

*Generalized I/O Vectors:* Generalized I/O vectors allow transfers to/from discontinuous regions that are made up of contiguous elements of like length. An array of pointers is specified for both the initiator and the target. Each entry in the array specifies the starting address of a contiguous memory region of length K. Both arrays are of length N.

*Arbitrary Discontiguous:* This access mode allows transfers to/from discontinuous regions with minimal restrictions. Contiguous elements within the discontinuous regions may be of any byte length. The total byte length of all contiguous elements must be equal at both the initiator and the target. This access

**OpenSHMEM - Toward a Unified RMA Model. Table 2** RMA (Remote) PUT/GET data-type support

| Type              | SGI | MP  | LC  | Q   | C64 | GASNet | ARMCI | MPI-2 |
|-------------------|-----|-----|-----|-----|-----|--------|-------|-------|
| byte              | Yes | Yes | Yes | Yes | Yes | Yes    | Yes   | Yes   |
| short             | Yes | Yes | Yes | Yes | No  | No     | No    | Yes   |
| int               | Yes | Yes | Yes | Yes | No  | No     | No    | Yes   |
| long              | Yes | Yes | Yes | Yes | Yes | No     | No    | Yes   |
| long long         | Yes | Yes | Yes | Yes | Yes | No     | No    | No    |
| float             | Yes | Yes | Yes | Yes | No  | No     | No    | Yes   |
| double            | Yes | Yes | Yes | Yes | Yes | No     | No    | Yes   |
| long double       | Yes | Yes | No  | Yes | No  | No     | No    | Yes   |
| integer (Fortran) | Yes | Yes | Yes | No  | No  | No     | No    | Yes   |
| logical (Fortran) | Yes | Yes | Yes | No  | No  | No     | No    | Yes   |
| real (Fortran)    | Yes | Yes | Yes | No  | No  | No     | No    | Yes   |
| complex (Fortran) | Yes | Yes | Yes | No  | No  | No     | No    | Yes   |
| 16 bits           | No  | Yes | Yes | No  | No  | No     | No    | No    |
| 4 bytes           | Yes | Yes | Yes | Yes | No  | Yes    | Yes   | No    |
| 32 bytes          | Yes | Yes | Yes | Yes | No  | No     | No    | No    |
| 8 bytes           | Yes | Yes | Yes | Yes | No  | No     | No    | No    |
| 64 bytes          | Yes | Yes | Yes | Yes | No  | No     | No    | No    |
| 128 bytes         | Yes | Yes | Yes | Yes | No  | No     | No    | No    |

**OpenSHMEM - Toward a Unified RMA Model. Table 3** RMA (Remote) PUT/GET access methods

| Type                    | SGI | MP  | LC  | Q   | C64 | GASNet | ARMCI | MPI-2   |
|-------------------------|-----|-----|-----|-----|-----|--------|-------|---------|
| Contiguous              | Yes | Yes | Yes | Yes | Yes | Yes    | Yes   | Yes     |
| Indexed                 | No  | Yes | Yes | No  | No  | No     | No    | Yes     |
| Strided                 | Yes | Yes | Yes | Yes | No  | No     | Yes   | Yes     |
| Generalized I/O Vectors | No  | No  | No  | No  | No  | No     | Yes   | Yes     |
| Arbitrary Discontinuous | No  | No  | No  | No  | No  | No     | No    | Unknown |

mode uses a pair of arrays to describe the memory locations and the length of each element at the memory location. Four arrays are therefore necessary to describe the memory layout for both the initiator and the target.

Note that MPI-2 supports Indexed, Strided, and Generalized I/O Vectors through the use of derived data-types.

## Synchronization

While SHMEM provides a barrier semantic which intertwines synchronization with remote completion, it also provides three somewhat more subtle tools for

enforcing ordering semantics on the sequence of memory updates arriving at a PE: fence, quiet, and wait.

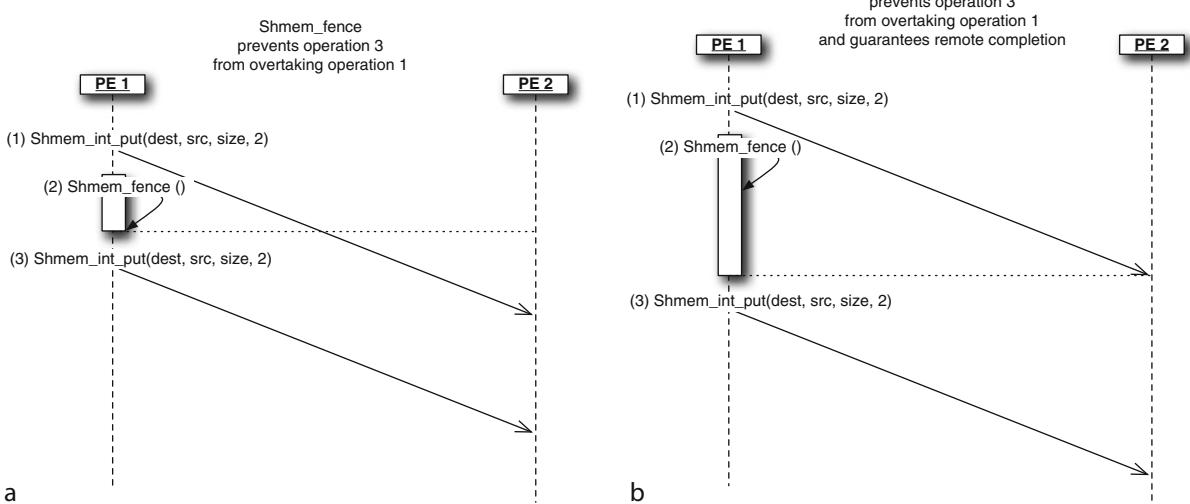
**Fence:** SHMEM Fence ensures ordering of PUT operations to a specific PE. In SGI-SHMEM, MP-SHMEM, and LC-SHMEM, the fence operation forces ordering of the completion of PUT operations on a remote PE. Specifically, while the order of posting of PUT operations on the remote PE is not generally guaranteed, the fence does provide a guarantee that on the remote (passive) side the PUTs issued from a particular source PE before the fence will be completed before the PUTs issued by that source PE after the fence are completed.

Note that this does not guarantee that the remote (passive) side will have completed the PUTs when the fence operation returns on the active side. Q-SHMEM and C64-SHMEM support `shmem_fence` via `shmem_quiet` and add the additional semantic that all outstanding PUT operations will be delivered to the target PE prior to return from the function. [Figure 2](#) illustrates the semantics of `shmem_fence`.

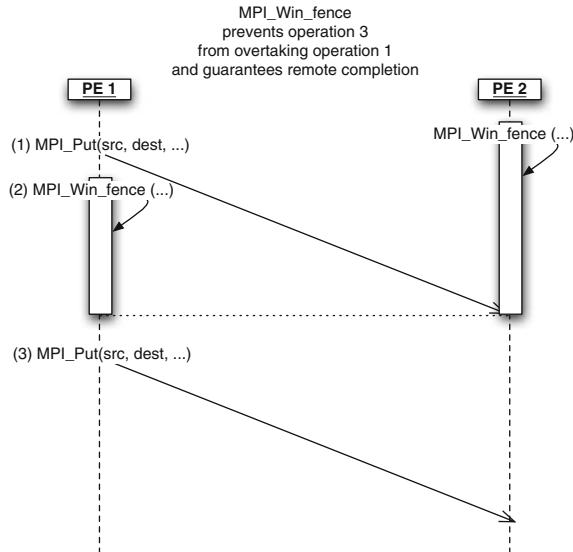
ARMCI provides a fence operation similar to that of Q-SHMEM in that it blocks until all PUTS are delivered to the remote PE. GASNet provides blocking and non-blocking versions of PUT. A blocking PUT will not return until data is delivered to the remote PE. A wait on a non-blocking PUT will not return until data is delivered to the remote PE. Fence (and Quiet) are therefore not necessary in GASNet as both are implicit in the semantics of blocking and non-blocking PUT operations. The GASNet blocking interface combines the semantics of RMA (Remote) initializations, ordering, and remote completion. Applications that do not need the additional semantics may incur additional overhead. GASNet's non-blocking PUT operations separate RMA (Remote operations) initialization from ordering and remote completion, but fail to separate ordering from remote completion.

MPI-2 specifies an MPI\_WIN\_FENCE operation, but differs dramatically from Fence in SHMEM. MPI\_WIN\_FENCE is a collective operation on the group of WIN requiring both the active and passive sides to participate, whereas SHMEM fence is called only on the active side. In addition, MPI\_WIN\_FENCE ensures that all outstanding RMA calls on a window (regardless of origin) are synchronized. Fence in SHMEM may be thought of strictly as an ordering semantic and not as a PE synchronization semantic as in MPI. [Figure 3](#) illustrates the semantics of MPI\_Win\_fence.

*Quiet:* SHMEM quiet ensures ordering of PUT operations to all PEs. In SGI-SHMEM, MP-SHMEM, and LC-SHMEM, the quiet operation ensures the order of PUT operations. All incoming PUT operations issued by any PE and local load and store operations started prior to the quiet call are guaranteed to be complete at their targets and visible to all other PEs prior to any remote or local access to the corresponding target's memory or any synchronization operations that follow the call to `shmem_quiet`. Q-SHMEM and C64-SHMEM extend the semantic in that data is delivered at the remote PE at the return of `shmem_quiet`. This additional semantic intertwines remote completion and ordering which may impose unnecessary overhead to applications



OpenSHMEM - Toward a Unified RMA Model. **Fig. 2** Example of fence **(a)** SGI-, MP-, and LC-SHMEM **(b)** Q- and C64-SHMEM



**OpenSHMEM - Toward a Unified RMA Model. Fig. 3**  
Example of MPI\_Win\_fence

that only require the ordering semantic and not remote completion. Figure 4 illustrates the semantics of shmem\_quiet.

ARMCI provides Quiet through a fence\_all operation similar to Q-SHMEM and C64-SHMEM in that data is guaranteed to be delivered at the remote PEs when fence\_all returns. Again, the ordering semantic is intertwined with remote completion.

**Barrier:** The SHMEM barrier is a collective synchronization routine in which no PE may leave the barrier prior to all PEs entering the barrier. SGI-SHMEM, MP-SHMEM, LC-SHMEM, and Q-SHMEM additionally require the semantic that all outstanding PUT operations are remotely visible prior to the return of the barrier operation. This additional semantic intertwines PE synchronization and remote completion of outstanding write operations. C64-SHMEM does not add this additional semantic.

**Wait:** SHMEM wait waits for a memory location or variable to change on the local PE. This allows synchronization between two PEs. SGI-SHMEM, MP-SHMEM, LC-SHMEM, Q-SHMEM, and C64-SHMEM all support shmem\_wait the other RMA implementations do not provide a similar semantic.

A possible improvement in these semantics would involve separating RMA initialization, RMA ordering, remote completion, and PE synchronization.

## Atomic Memory Operations

An atomic memory operation (AMO) provides the capability to perform simple operations on the passive side of the RMA operation. The SHMEM model provides five common AMO capabilities.

**Swap:** Writes the value specified by the initiator PE to the target PE location and returns the target value prior to the update.

**Compare and Swap:** Compares the conditional value specified by the initiator PE to the target PE location, and if they are equal, it writes the value specified by the initiator PE to the target PE location. The return value is the target PE location's value prior to update.

**Mask and Swap:** Writes the value specified by the initiator PE to the target PE location updating only the bits in the target location as specified in the mask value. The return value is the target PE location's value prior to update.

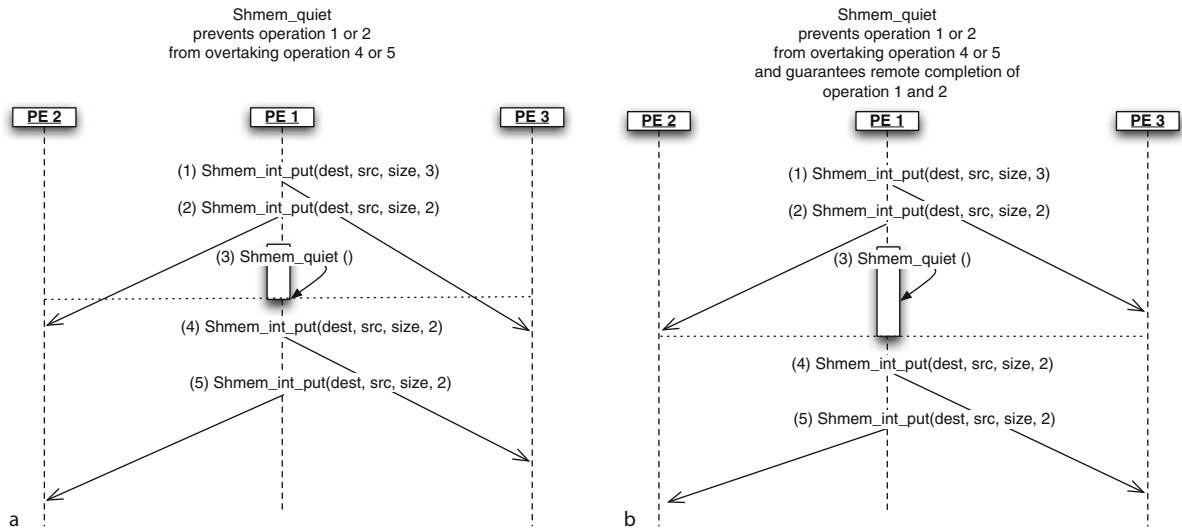
**Fetch and Add:** Atomically adds the value specified by the initiator PE to the target PE location's value. The return value is the target PE location's value prior to the update.

**Fetch and Increment:** Special case of Fetch and Add with implicit add value of 1.

**Extended Atomic Operation Support:** Cyclops-64 includes a general purpose atomic operations: **long** shmem\_atomic\_op(**long** \*target, **long** value, **long** pe,atomic\_t op), which performs the atomic operation specified by atomic\_t op on the specified target. The number of atomic operations supported is quite large and corresponds to Cyclops-64 intrinsic operations.

A more complete list of AMOs along with their definitions which comes from [10] is shown in the OpenSHMEM section of this entry.

Rather than providing atomics as defined above, MPI-2 provides MPI\_ACCUMULATE. While MPI\_ACCUMULATE provides atomic updates of local and remote variables the value of the variable prior to update is not returned to the caller. Atomic swap (and variants) are not supported by MPI-2. The following operations are available for MPI\_ACCUMULATE:



OpenSHMEM - Toward a Unified RMA Model. Fig. 4 Example of quiet (a) MP-SHMEM, LC-SHMEM (b) Q-SHMEM

OpenSHMEM - Toward a Unified RMA Model. Table 4 Synchronization methods

| Synchronization | SGI | MP  | LC  | Q   | C64 | GASNet | ARMCI | MPI-2 |
|-----------------|-----|-----|-----|-----|-----|--------|-------|-------|
| Fence           | Yes | Yes | Yes | Yes | Yes | No     | Yes   | No    |
| Quiet           | Yes | Yes | Yes | Yes | Yes | No     | Yes   | No    |
| Barrier         | Yes | Yes | Yes | Yes | Yes | Yes    | Yes   | Yes   |
| Wait            | Yes | Yes | Yes | Yes | Yes | No     | No    | No    |

OpenSHMEM - Toward a Unified RMA Model. Table 5 Atomic operations

| Atomic Op           | SGI | MP  | LC  | Q   | C64 | GASNet | ARMCI | MPI-2 |
|---------------------|-----|-----|-----|-----|-----|--------|-------|-------|
| Swap                | Yes | Yes | Yes | Yes | Yes | No     | Yes   | No    |
| Compare and Swap    | Yes | Yes | Yes | Yes | Yes | No     | No    | No    |
| Mask and Swap       | No  | Yes | No  | Yes | Yes | No     | No    | No    |
| Fetch and Add       | Yes | Yes | Yes | Yes | Yes | No     | Yes   | No    |
| Fetch and Increment | Yes | Yes | Yes | No  | Yes | No     | No    | No    |

OpenSHMEM - Toward a Unified RMA Model. Table 6 Reductions

| Reduction | SGI | MP  | LC  | Q   | C64 | GASNET | ARMCI | MPI-2 |
|-----------|-----|-----|-----|-----|-----|--------|-------|-------|
| And       | Yes | Yes | Yes | Yes | Yes | No     | No    | Yes   |
| Max       | Yes | Yes | Yes | Yes | Yes | No     | Yes   | Yes   |
| Min       | Yes | Yes | Yes | Yes | Yes | No     | Yes   | Yes   |
| Or        | Yes | Yes | Yes | Yes | Yes | No     | No    | Yes   |
| Product   | Yes | Yes | Yes | Yes | Yes | No     | Yes   | Yes   |
| Sum       | Yes | Yes | Yes | Yes | Yes | No     | Yes   | Yes   |
| Xor       | Yes | Yes | Yes | Yes | Yes | No     | No    | Yes   |

**OpenSHMEM - Toward a Unified RMA Model. Table 7** Collectives

| Collective | SGI | MP  | LC  | Q   | C64 | GASNet | ARMCI | MPI-2 |
|------------|-----|-----|-----|-----|-----|--------|-------|-------|
| Broadcast  | Yes | Yes | Yes | Yes | Yes | No     | Yes   | Yes   |
| Collect    | Yes | Yes | Yes | Yes | No  | No     | No    | Yes   |
| fcollect   | Yes | Yes | Yes | Yes | Yes | No     | No    | Yes   |

```
#include <mpp/shmem.h>

int main(int argc, char* argv[])
{
 int i, me, my_num_pes;
/*
** Starts/Initializes SHMEM/OpenSHMEM
*/
 start_pes(0);
/*
** Fetch the number of processes
** Some implementations use num_pes();
*/
 my_num_pes = _num_pes();
/*
** Assign my process ID to me
*/
 me = _my_pe();
 printf("Hello World from %d of %d\n", me, my_num_pes);
 return 0;
}
```

**OpenSHMEM - Toward a Unified RMA Model. Fig. 5**

Trivial hello world in SHMEM

```
#include <mpi.h>

int main(int argc, char* argv[])
{
 int rank, size;
/*
** Starts/Initializes MPI
*/
 MPI_Init(&argc, &argv);
/*
** Get the current process id
*/
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
/*
** Get the current number of processes
*/
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 printf("Hello World from process %d of %d\n", rank, size);
/*
** Clean up shop and exit
*/
 MPI_Finalize();
 return 0;
}
```

**OpenSHMEM - Toward a Unified RMA Model. Fig. 7**

Trivial hello world in Message Passing Interface (MPI)

```
#include <upc.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
 int i;
 for (i = 0; i < THREADS; ++i)
 {
 upc_barrier;
 if (i == MYTHREAD)
 printf ("Hello world from thread: %d\n", MYTHREAD);
 }
 return 0;
}
```

**OpenSHMEM - Toward a Unified RMA Model. Fig. 6**

Trivial hello world in UPC

```
/* circular shift bbb into aaa */
#include <mpp/shmem.h>
int aaa, bbb;
int main (int argc, char * argv[])
{
 start_pes(0);
 shmem_int_get(&aaa, &bbb, 1, (_my_pe() + 1) % _num_pes());
 shmem_barrier_all();
}
```

**OpenSHMEM - Toward a Unified RMA Model. Fig. 8**

Circular shift in SHMEM

- Maximum
- Minimum
- Sum
- Product
- Logical and
- Bit-wise and

- Logical or
- Bit-wise or
- Logical xor
- Bit-wise xor

## Reductions

Reductions perform an associative binary operation across a set of values on multiple PEs. MP-SHMEM,

```

/* circular shift bbb into aaa */
int aaa, bbb;
int main(int argc, char * argv[])
{
 int numprocs, me;
 MPI_Comm_rank(MPI_COMM_WORLD, &me);
 MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
 send_to = (me - 1 + numprocs) % numprocs;
 recv_from = (me + 1) % numprocs;
 MPI_Bsend(&bbb, 1, MPI_INT, send_to, tag,
 MPI_COMM_WORLD);
 MPI_Recv(&aaa, 1, MPI_INT, recv_from, tag,
 MPI_COMM_WORLD, &status);
}

```

**OpenSHMEM - Toward a Unified RMA Model. Fig. 9**

Circular shift in MPI

```

/* circular shift bbb into aaa */
shared int aaa[THREADS], bbb[THREADS];
int main (int argc, char * argv[])
{
 aaa[MYTHREAD] = bbb[(MYTHREAD + 1) % THREADS];
 upc_barrier;
}

```

**OpenSHMEM - Toward a Unified RMA Model. Fig. 10**

Circular shift in UPC

LC-SHMEM, and Q-SHMEM all provide the same reduction operations differing only in the supported data-types as detailed previously. GASNet provides no reduction operations. ARMCI supports some of the reduction operations of SHMEM. MPI-2 supports all the SHMEM reduction operations as well as reductions based on logical operations.

## Collective Communication

Collective communication is uniformly supported across SGI-SHMEM, MP-SHMEM, LC-SHMEM, and Q-SHMEM. C64-SHMEM and other communication libraries differ in their support for these (or similarly defined) operations.

*Broadcast:* Transfers data from the root PE to the active set of PEs. The active set of PEs is specified using a starting PE and a  $\log_2$  stride between consecutive PEs.

*Collect:* Gathers data from the source array into the target array across all PEs in the active set. Each

PE contributes their source array which is concatenated into the target array on all the PEs in the active set in ascending PE order. Source arrays can differ in size from PE to PE. MPI-2 provides MPI\_ALL\_GATHERV which provides similar semantics to Collect.

*fcollect:* A special case of Collect in which all source arrays are of the same size. This allows for an optimized implementation. MPI-2 provides MPI\_ALL\_GATHER which provides similar semantics to fcollect.

## Tool Support

Few performance tools have been developed specifically for SHMEM and Partitioned Global Address languages. The challenge comes as a result that traditional two-sided communication analyses do not apply to sided data accesses to “symmetric,” or shared variables as the library calls are decoupled from point-to-point synchronization. An event-based performance analysis tool [16] was developed to support both SHMEM and UPC which uses a wrapper function approach to gather SHMEM events. Tools such as CrayPat, TAU, and SCALASCA use similar approaches to profile and trace SHMEM calls. However, the wrapper function approach might not work well for future SHMEM-aware compilers since some might inline and replace SHMEM calls with direct shared memory accesses on certain architectures. GASP [17] has been used as an alternative mechanism to gather performance events for GASNet which is implicit from the communication library API. None of these approaches have been sanctioned for standardization or have been officially adopted for SHMEM. However, work is on the way.

Commercial available debuggers [18, 19] and compile time static checkers [20, 21] have limited support for SHMEM and UPC not to say tools that optimize them. In the case of OpenSHMEM, the semantics of the library can allow compilers or static checkers to report the incorrect use of SHMEM library calls. For example, compiler-based tools can easily check if arguments to the data access calls meet the “symmetric” storage for some variables, check if variables are accessed within bounds, and detect incorrect type usage in `shmemb_put32/64` calls. Such semantic checker tools can be invoked as preprocessors to help the

user reduce the amount of errors detected at runtime. Research needs to be done to find proper ways to optimize SHMEM in compilers. Traditional dataflow frameworks such as SSA (single-static assignment) have been extended in tools for MPI to handle two-sided communication and few for one-sided communications but not specifically for OpenSHMEM programs; and these technologies still have to be introduced in production compilers. The OpenSHMEM community needs to promote the development of tools that facilitate the usage of SHMEM programs as few tools exist for this as for now.

## Toward an Open SHMEM standard

When comparing the various implementations of SHMEM and other RMA APIs, we see that semantics vary dramatically. Not only do SHMEM RMA semantics differ from other RMA implementations (as expected) but different SHMEM implementations differ from each other. Support for primitive data types varies. Discontiguous RMA operations and Atomic RMA operations are not uniformly supported. Synchronization and completion semantics differ substantially which can cause valid programs on one architecture to be completely invalid on another. Symmetric data objects that dramatically aid the programmer are unique to the SHMEM model but are not uniformly supported. There are a number of capabilities that are available from implementations that differ from the SGI SHMEM baseline version. Below is a brief list of potential future additions to OpenSHMEM:

- Support for Multiple Host Channel Adapters (Rails)
- Support for non-Blocking Transfers
- Support for Events
- Additional Atomic Memory Operations and Collectives
- Potential options for NUMA/Hybrid architectures
- Additional communications transport mechanisms
- OpenSHMEM I/O library enhancements
- OpenSHMEM tools and Compiler enhancements
- Enabling exa-scale applications

An OpenSHMEM standard can address the lack of uniformity across the available SHMEM implementations. Standardization levels can be established to provide a base level that all SHMEM implementations must support in order to meet the standard, with higher levels

available for additional functionality and platform specific features.

In 2008, an initial dialogue was started between SGI and a small not-for-profit company called Open Source Software Solutions, Inc.(OSSS). The purpose of the dialogue was to determine if an agreement could be reached for SGI's approval of an open source version of SHMEM that would serve as an umbrella standard under which to unify all of the already existing implementations into one cohesive API. As part of this discussion, OSSS held a BOF on OpenSHMEM at SC08 to discuss this plan. The BOF was well attended by all of the vendors interested in supporting OpenSHMEM/SHMEM, and many of the interested parties in the SHMEM user community. The unanimous opinion of the attendees favored continuing the process toward developing OpenSHMEM as a community standard. The final agreement between SGI and OSSS was signed in 2010. The agreement allows OSSS to use the name OpenSHMEM and directly reference SHMEM going forward. The base version of OpenSHMEM V1.0 is based on the SGI man pages. The "look and feel" of OpenSHMEM needs to preserve the original "look and feel" of the SGI SHMEM. OpenSHMEM version 1.0 has been released and input for version 2.0 is being actively solicited.

There are a number of enhancements under discussion for version 2.0 and future offerings. Some of the features specific to one implementation or another as listed in the preceding tables and elements will be incorporated into later versions of OpenSHMEM. OpenSHMEM will be supported on a variety of commodity networks (including Infiniband from both Mellanox and QLogic) as well as several proprietary networks (Cray, SGI, HP, and IBM). The OpenSHMEM mail reflector is hosted at ORNL and can be joined by sending a request via the OpenSHMEM list server at <https://email.ornl.gov/mailman/listinfo/openshmem>. Future enhancements and RFIs will be sent to developers and other interested parties via this mechanism. Source code examples, Validation and Verification suites, performance analysis, and OpenSHMEM compliance will be hosted at the OpenSHMEM Web site at <http://www.openshmem.org>, and the OpenSHMEM standard will be owned and maintained by OSSS.

With the inexorable march toward exa-scale, programming methodologies such as OpenSHMEM will certainly find their place in enabling extreme scale architectures. By virtue of decoupling data motion from synchronization, OpenSHMEM exposes the potential for synergistic application improvements by scaling more readily than two-sided models, and by minimizing data motion, thus affording the possibility of concomitant savings in power consumption by applications. These gains, along with portability, programmability, productivity, and adoption, will secure OpenSHMEM a place for future extreme scale systems.

## Acknowledgments

This work was supported by the United States Department of Defense and used resources of the Extreme Scale Systems Center located at Oak Ridge National Laboratory.

## Bibliography

1. Message Passing Interface Forum (1994) MPI: a message passing interface standard. *Int J Supercomputer Appl* 8(3/4): 159–416
2. Luszczek PR, Bailey DH, Dongarra JJ, Kepner J, Lucas RF, Rabenseifner R, Takahashi D (2006) The HPC challenge (HPCC) benchmark suite. In: SC'06: Proceedings of the 2006 ACM/IEEE conference on supercomputing, ACM, New York, p 213
3. Feind K (2010) SHMEM: an overview and brief history (2010 SC10 OpenSHMEM BOF)
4. Curtis T, Pophale S (2010) SHMEM tutorial at PGAS'10
5. Su HH (2005) SHMEM tutorial
6. Su HH (2010) Programming in SHMEM
7. SGI Inc.: SGI SHMEM API Man Pages
8. Cray Inc.: Man page collection (Unicos MP): shared memory access HMEM
9. Cray Inc.: Man page collection (Unicos LC): shared memory access SHMEM
10. Quadrics Ltd.: The SHMEM programming manual
11. ET International Inc.: Cyclops-64 programming manual
12. Bonachea D (2002) GASNet specification, v1.1. Technical report, University of California, Berkeley
13. PNNL: ARMCI – Programming interfaces. <http://www.emsl.pnl.gov/docs/parsoft/armci/documentation.htm>
14. Message Passing Interface Forum (1998) MPI2: a message passing interface standard. *High Perform Comput Appl* 12(1–2):1–299
15. Bonachea D, Duell J (2003) Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. In: 2nd workshop on hardware/software support for high performance scientific and engineering computing (SHPSEC-03), New Orleans
16. Su H, Billingsley M, George A (2008) Parallel performance wizard: a performance analysis tool for partitioned global address

space programming. In: 9th IEEE international workshop on parallel and distributed scientific and engineering computing (PDSEC), Miami

17. Leko A, Bonachea D, hsun Su H, Golden B, Sherburne H, George AD (2005) GASP: a performance tool interface for global address space languages. Technical report. Lawrence Berkeley National Lab, Berkeley
18. Krammer B, Himmeler V, Lecomber D (2007) Coupling DDT and Marmot for debugging of MPI applications. In: PARCO, Jülich/Aachen, pp 653–660
19. Gottbrath C, Thompson P (2006) TotalView tips and tricks. In: Proceedings of the 2006 ACM/IEEE conference on supercomputing (SC'06), ACM, New York
20. Luecke G, Coyle J, Hoekstra J, Kraeva M, Xu Y, Kleiman E, Weiss O (2009) Evaluating error detection capabilities of UPC run-time systems. In: The 3rd conference on partitioned global address space programming models, Ashburn
21. Liao C, Quinlan D, Panas T, de Supinski B (2010) A ROSE-based OpenMP 3.0 research compiler supporting multiple run-time libraries. In: Sato M, Hanawa T, Müller M, Chapman B, de Supinski B (eds) Beyond loop level parallelism in OpenMP: accelerators, tasking and more. Lecture notes in computer science, vol 6132. Springer, Berlin/Heidelberg, pp 15–28

## Operating System Strategies

ROLF RIESEN<sup>1</sup>, ARTHUR B. MACCABE<sup>2</sup>

<sup>1</sup>IBM Research, Dublin, Ireland

<sup>2</sup>Oak Ridge National Laboratory, Oak Ridge, TN, USA

## Synonyms

Parallel operating system; Resource management for parallel computers; Runtime system

## Definition

An operating system manages the processors and other resources of a parallel computing system. Multiple instances of individual operating systems and the runtime system form a parallel operating system, which manages the resources of the entire machine and provides services for users and system administrators to obtain information and control various aspects of the machine and the application jobs that run on it.

## Discussion

### Introduction

A parallel computing system employs two or more processing elements (PE) which can be single or multicore



CPUs plus attached Graphic Processing Units (GPU) or other accelerators. Usually the PEs are general-purpose microprocessors, but specialized CPUs have been used. These PEs and other resources in the system need to be managed so they can run parallel jobs and be used effectively.

Some of the tasks of an OS for a parallel system are the same as the tasks a traditional OS performs for a desktop computer or a server. These include starting a job (or process), memory management, and I/O to disks and other peripherals. There are additional tasks an OS for a parallel system must perform or support. For example, in a distributed memory system where not every CPU has access to all memory in the system, data must be transferred between memories. The data is transmitted inside messages, and the OS must provide functions for sending and receiving these messages, or provide functions for applications to access the network interface directly in a safe manner.

Larger parallel machines often use a batch system to schedule jobs. Users submit their requests for a certain number of nodes, how long they will need them, and what application they want to run. The batch system then schedules job launches based on node availability and job priority. The batch system interacts with the OS to allocate nodes, launch jobs, and terminate jobs that have exceeded their requested execution time.

Yet another task that an OS for a parallel system must perform, that is not needed on personal computers, is process coordination across the machine. Very often the processes that are part of a parallel application need to communicate and synchronize with each other. Some parallel systems provide hardware or special networks for such operations, and the OS provides primitives to access these.

Parallel OSs also face constraints that general-purpose OSs do not have. On some large parallel machines, there is relatively little memory for each PE. The more of that memory is consumed by the OS, the less is available for the applications. Since every megabyte wasted is multiplied by the number of PEs in the system, this can be a significant cost.

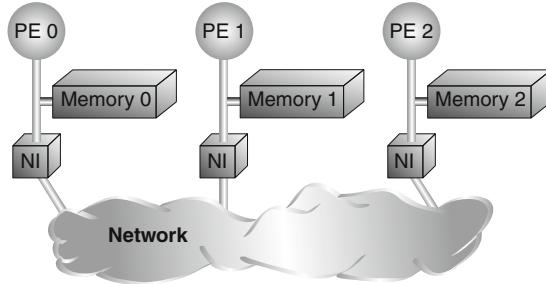
General-purpose OSs do a lot of housekeeping tasks, such as checking for email or the status of an attached printer, while waiting for a key stroke or mouse click from the user. While these administrative tasks do not take a lot of time, they can slow down a parallel

application in the following way. When the individual copies of an OS running on multiple PEs, or the application itself, exchange messages or synchronize activities, they need to interact with each other. When one of them is busy with administrative tasks, it delays all other processes that are part of the synchronization. When the busy OS finds time to service a request and then needs an acknowledgement or more information from another PE, it itself may be delayed because the other OS is now doing administrative tasks instead of responding to requests. This cascading effect is called *OS noise* and can slow down parallel applications. The main strategy to avoid this problem in a parallel OS is to disable services such as email and printing, which are not needed on the PEs of a parallel system. Other strategies include synchronizing timer interrupts and OS services in general and dedicating specific processor cores to OS work.

In addition to the tasks a parallel OS must perform and the constraints it faces, there are also a few things that make a parallel OS simpler. Usually, the PEs in a system are all of the same type and have very few peripherals. That means that a, possibly customized, parallel OS needs to support only a few hardware devices and a lot of functionality, for example, driving a display, is not needed within the OS. Often it is enough to manage the CPU, memory, and the network interface, since no other peripherals are attached or used. Many large-scale systems have no local disks, and the necessary device drivers and file systems need not be present in the OS. In such a case, I/O is done using messages to an external parallel file system running on a disk farm separate from the parallel compute system. This leads to opportunities for simplifications and helps reducing the memory footprint of the OS and the noise it introduces into the system.

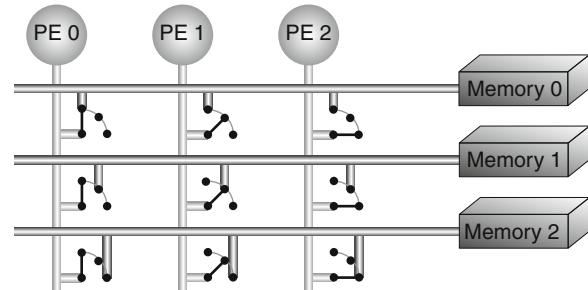
## Parallel Computing Systems

There are two main categories of parallel computing systems: message-passing and shared-memory systems. In a message-passing system, each PE has local memory that it can access directly, using load and store operations. Each PE is also connected to a network that allows data to move from the memory of one PE to the memory of another PE. In order to access data stored in memory attached to a remote PE, the data has to be moved from



**Operating System Strategies. Fig. 1**

A multicomputer consists of PEs that each have their own local memory. Data is moved from one PE to another using explicit message passing. The Network Interface (NI) connects the system bus of a PE to the network



**Operating System Strategies. Fig. 2**

In a shared-memory system, all PEs have access to all memory. A bus or, as in this picture, a crossbar switch connects the PEs to the available memory modules

one memory to the other using *explicit message passing*. Large-scale supercomputers by companies like Cray and IBM are of this type. Clusters, which are built out of commodity computers (desktops or servers), are also message-passing systems. The individual computers in a cluster are connected with a commodity network such as Ethernet or Infiniband, while supercomputers often employ vendor specific networks. Systems where the PEs do not share memory are called multicomputers. Large enough clusters and message-passing systems are sometimes called Massively Parallel Processors (MPP). Figure 1 shows an example of a multicomputer.

In a shared-memory system, all PEs have access to all of the memory in the system. Figure 2 shows an example. The diagram shows a crossbar switch where hardware sets the switches and enables access to memory based on the address the PE uses in its load or store instruction. Smaller and less expensive systems use a bus instead of a crossbar switch. Systems that use a crossbar switch or a shared memory bus have Uniform Memory Access (UMA) where the cost of accessing any memory location is roughly the same. UMA systems are often called Symmetric Multi Processors (SMP) and sometimes multiprocessors.

Larger systems, with many PEs, are Non-Uniform Memory Access (NUMA) machines. Accessing local memory is much faster than accessing the memory that is attached to a remote PE. Although memory access in a shared-memory system may not be uniform, the process of moving data is done by hardware and largely hidden from the program doing the access. A program simply issues load and store instructions to

global addresses which the system resolves, if necessary, by moving data from a remote memory to the PE doing the request. This is in stark contrast to explicit message passing and changes how these systems are programmed and the services an OS must provide.

The OS in a shared-memory system is responsible for enforcing memory access restrictions by configuring the underlying hardware accordingly and providing abstractions for sharing data among processes when needed. In a message-passing system, the OS must provide mechanisms for explicitly passing messages from one PE to another and enforce access restrictions to local memory and the network interface to protect other parts of the system from malicious messages.

Coherency, when a data change in remote memory becomes visible to a local PE, and how write access conflicts are handled depends on the hardware available in a shared-memory system. The OS may provide synchronization and access coordination primitives. The OS is also responsible to initialize the hardware, for example, to set caches to write-through or write-back.

The networks connecting the PEs in a message-passing system range from commodity networks used in clusters to custom hardware used in some supercomputers. A network consists of routers that are connected to each other and form a topology such as a 2-D or 3-D mesh, a hypercube, or a tree. The PEs connect to the routers via a network interface. Some network interfaces are simple, and the OS is responsible to trigger data movement in or out of the network. High-end network interfaces have processors and memory of their own and can be programmed to process incoming data

and make decisions where in memory the data needs to be deposited. For systems like that, the OS is responsible to restrict unprivileged access and provide abstractions to control and program the network interface.

Parallel computing systems are often partitioned so that each node in the system serves a specific purpose. [Figure 3](#) shows an example. Each node is an individual computer of the many that make up a multicomputer. A node in a multicomputer consists of memory, one or more PEs, and a connection to the network. Some nodes have additional network connections that are used by users to log into the system. When they do, they land on one of the service nodes from where they launch and control parallel applications. Often, development and compiling of parallel applications is done on the service nodes as well. The service partition in [Fig. 3](#) consists of the light-colored spheres at the top left.

The majority of nodes in a parallel system are reserved for computation. The cubes in [Fig. 3](#) represent those nodes. Users or a batch processing system launch jobs from the service partition. The runtime system allocates nodes from the compute partition and instructs the OSs on these nodes to start the processes of that application.

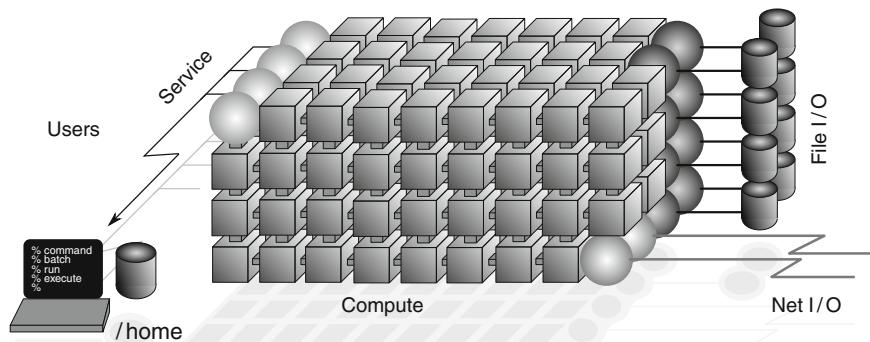
Many larger systems have a parallel file system that is external to the parallel machine itself. In the partitioning model, dedicated I/O nodes, the dark spheres on the right of [Fig. 3](#), are used to coordinate traffic to these external disks. The compute nodes send messages containing file data to the I/O nodes. The file system running on the I/O nodes then deposits the data on the disks. The compute nodes themselves do not

need to run a file system themselves. Some systems may be connected to other, wide-area, networks. Additional I/O nodes are sometimes used to handle the traffic to and from these networks (bottom right in [Fig. 3](#)).

Some clusters have local disks on each node since they consist of commercial, off-the-shelf computers that often come with built-in disks. The usage of these local disks is somewhat problematic on compute nodes for anything other than temporary data during the run of a job. Once the job finishes and the compute nodes are reallocated to another job, it becomes difficult to locate and retrieve the data from the earlier job run. Sometimes service nodes in clusters also serve as compute nodes, but better performance can be achieved using a strict separation of compute and service nodes.

### Overview of OS Approaches

Most clusters and supercomputers today run an OS that is a version of UNIX, such as various versions of Linux, IBM's AIX, or Sun Microsystem's Solaris. These are usually slightly modified or tuned versions of the corresponding desktop and server OSs. An alternative approach is to run a lightweight kernel on the compute nodes and a full-featured OS on the service and I/O nodes. Systems employing this approach include IBM's Blue Gene which runs the Compute Node Kernel (CNK) [7], Cray's XT-3 which runs Catamount [10], Intel's ASCI Red at Sandia National Laboratories which ran Cougar (a port of Puma [14]), and the Intel Paragon which had the option of running the Sandia/University of New Mexico OS (SUNMOS) [11].



**Operating System Strategies. Fig. 3** Partitioning of a system by dedicating nodes to usage functions such as compute, service, and I/O [10]

In the late 1980s and early 1990s, lightweight kernels were required because the supercomputers of that time had very limited amounts of memory available and every last bit of performance needed to be available to the parallel applications. That meant the OS had to have a very small memory footprint and very low processing overhead.

With the advent of clusters in the late 1990s, the situation began to change. Users began to demand features such as multithreading and shared libraries on parallel systems. Because the processors had become more powerful, users were often willing to sacrifice some performance in return for the additional features and convenience of programming. Today most parallel computers run full-featured OSs, while only a few high-end systems use lightweight kernels to ensure performance and scalability.

Desktop and server OSs are designed to allow multiple users or multiple tasks to share the available resources. Memory is partitioned and allocated to the currently running processes. The OS is responsible for memory allocation and protection, making sure that a process has access to only its portion of the memory. In addition, the OS provides control mechanisms to share memory among processes when that is needed for data exchanges. CPU cycles are bundled into time slices and allocated to processes which are currently ready to run. This happens many times per second and provides the illusion to a user that multiple processes are making progress at the same time, even when there is only a single CPU present in the computer.

For the most part, these traditional OS services are not needed in a parallel computer. Instead of time-sharing individual CPUs, whole sets of PEs (CPUs) are allocated to a single application that uses the PEs in parallel to solve a computationally intensive problem. This method of allocating resources is called *space-sharing*. Multiple applications may be running simultaneously on a parallel system, but each application has been allocated memory and PEs for its exclusive use.

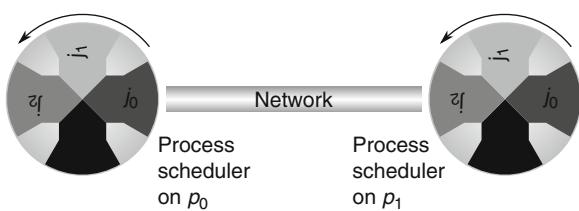
The main reason for space-sharing rather than time-sharing a parallel machine is throughput: The number of jobs a parallel system can complete in a given amount of time. This number depends on how long each job takes to complete. If the PEs of a parallel system were to run multiple jobs concurrently, completion time of a job would increase with the number of PEs it uses.

Using Fig. 4 as an example, let us assume there are multiple jobs,  $j_0 \dots j_3$ , running concurrently on PEs  $p_0$  and  $p_1$ . Since the activities on the individual PEs are not synchronized, it will happen that in a given time slice  $j_0$  is active on  $p_0$ , while  $j_2$  is active on  $p_1$ . If  $j_0$  needs to frequently transfer data between  $p_0$  and  $p_1$ , it will have to wait for its process to become active on  $p_1$  every time it requests an answer. Only active processes can send and reply to messages. When the OS on  $p_1$  makes  $j_0$  active there through a context switch,  $j_0$  will be able to send a reply from  $p_1$ . By the time the reply arrives at  $p_0$ ,  $j_0$  may not be active on that PE anymore, now causing wasted time for  $j_0$  on  $p_1$  waiting for an answer.

On some systems, the process schedulers can be synchronized, leading to a mode of operation called *gang scheduling*. Gang scheduling is not very common and difficult to achieve on large machines due to signal delays from one end of the machine to the other.

## Parallel OS Functionality

In most cases, each node of a parallel system runs one copy of the OS which manages all the resources local to that node. The collection of individual OSs on all the nodes of a parallel system forms the parallel OS and runtime system that controls the machine. In addition to the local OS services, this collection of system software must provide the ability to allocate nodes, launch and control parallel applications, for example, stopping and terminating all processes of a parallel job. In addition, services for debugging, determining system status, and parallel file I/O are usually available.



**Operating System Strategies. Fig. 4**

The two wheels represent the process schedules on  $p_0$  and  $p_1$ . Each process on these two PEs gets its turn to run in a round-robin fashion. Only a running process can send and reply to messages. If the rotation of the wheels (process scheduling) is not synchronized, delays occur because communicating processes have to wait for each other

Parallel jobs can be launched by users interactively from the service partition or through a batch scheduling system. In the latter case, users submit scripts to the batch system that contains information about what application the user wishes to run, how much time it needs to complete, and how many nodes it uses. The batch system queues these requests and when compute nodes become available selects the highest priority jobs and those that will best utilize the system and launches them. This is done in conjunction with the local OSs, but is usually a separate utility.

Whether a job is launched interactively by a user or by the batch system, the job launcher must interact with a node allocator to select the compute nodes that will be used to run the job. A node allocator can be simple and select from the available node list in a sequential fashion. More sophisticated allocators attempt to group the individual processes of an application physically close together on nodes that have one or only a few network hops between them. When the processes of a parallel application are scattered throughout a parallel system, their network traffic can interfere with the traffic from other applications, or traffic to and from the I/O nodes. Message latency also increases when application processes are further apart from each other.

Once nodes are allocated to a job, the launcher must interact with the OS instances on the selected nodes and initiate the start of the processes on each node. This involves coordination and sometimes synchronization among the selected nodes. Once the application is running, the job launcher serves as a collection point for standard input and output for each process of the application and as a control center. The job launcher can terminate the application and often serves as a conduit for debugging and signaling the application processes. Although not strictly part of the OS, the job launcher interacts with the OS on the nodes to control the processes. It also interacts with the node allocator, which is often a separate process, to allocate and release compute nodes. The node allocator often has an interface that allows users and system administrators to obtain status information, such as how many nodes are available for allocation, down for servicing, or currently running interactive or batch jobs.

Memory in early parallel systems was a scarce commodity and the OS had to be designed to consume

as little as possible of that resource. For example, an early version of OSF/1-AD, the OS for the Intel Paragon, consumed more than half of the 16 MB of memory installed on each node for its own code, message buffers, and other data structures. SUNMOS, another OS available for that machine, consumed only a few hundred kilo bytes. Cost of memory for that machine was a factor and wasting half of it on administrative services was not cost effective and limited the scope of scientific simulations that could be run.

Memory has become a lot cheaper since then and is now available in abundance, but conserving it may still be important. For example, some members of IBM's Blue Gene family of computers have a large number of PEs but a modest 512 MB of memory per PE. The future will increase the number of compute cores per processor, which in many systems may not be followed with an equal increase in memory size per core. Some systems may employ new types of memory, for example, FLASH memory which retains its content after a loss of power. Such new memory technologies may be, initially, more expensive and smaller in capacity leading to a need to conserve memory.

While a desktop OS should not waste memory either, it is not quite as critical there. Desktop systems and servers employ a technique called demand paging to provide the illusion that a system has much more memory than is physically available. When the physical memory becomes full, the OS will migrate less frequently used data to the swap area on a local disk and bring it back into memory later when it is needed again. In parallel systems, even on clusters with local disks, demand paging is usually disabled. The long delays of swapping and the unpredictable timing of it harm a parallel system much more because all processes of a single application must make progress together. Each time one of them is delayed, the whole application is delayed.

The individual instances of an OS on each PE run mostly independent of each other. This makes it more difficult to gather information about the system as a whole and exert control over it. For example, it might be interesting to see how busy the individual CPUs in a system are and how many processes are currently running on each. Another interesting statistic is the amount of available memory on each PE. Information like that can

be used to diagnose the health of a system and pinpoint problems. On a UNIX desktop system or server, there are many different utilities to gather such information.

The goal of a Single System Image (SSI) is to provide similar functionality, but for all PEs in a system. Instead of issuing commands to all PEs in a system, which can deliver a daunting amount of information, SSI usually allows commands to be restricted to a subset of PEs, for example, the ones currently running a given job. Examples of SSI systems include BProc, MOSIX, and OpenSSI. In addition to process control, some SSI provide functionality such as process migration, process check-pointing, a shared file system with a single root, and inter-process communications (IPC) between processes independent of their location. Implementing an SSI is easier on a shared-memory system where processes have access to all of memory and sharing data is automated by the hardware. An SSI for a multicomputer using message passing is much more difficult. Typically, full-featured SSI scale to a few tens of nodes, while those with a more restrictive set of features have been shown to run on up to a few hundred nodes. An SSI is integrated into the OS of a parallel computer, or, if SSI is a separate module, needs to make use of services the OS provides.

File I/O, and I/O in general, is a quandary on a parallel system. One problem in large systems is that not enough pathways are available to reach a shared storage device, which leads to network congestion. A second problem is the sharing of files and metadata, which contains information about the status of individual files and directories. For example, if many thousand processes start creating files in a single directory, the information for that directory must be continuously updated which creates an access bottleneck due to locking, unlocking, and synchronizing of these data structures.

Parallel file systems, not usually found on desktop systems or servers, attempt to alleviate the problem. A parallel file system often runs on the dedicated nodes of an I/O partition. These nodes, not burdened with compute tasks, can aggregate the data flow in and out of a parallel storage system and consolidate many of the updates and synchronizations that would otherwise have to happen individually. Parallel file systems in use on clusters and other large parallel computers include PVFS and Lustre.

## Lightweight Versus Full-Featured Operating Systems

Lightweight kernels are used on the compute nodes of some very large parallel systems. Often they are criticized for not providing enough functionality for today's highly complex and multi-layered parallel applications. Some argue that lightweight kernels are difficult to maintain since few people are familiar with them. This section briefly explores why lightweight kernels are a good fit for large message-passing systems.

In a partitioned systems like the one in Fig. 3, instances of a lightweight kernel run on the compute nodes, while the service and I/O nodes run a full-featured OS. This allows a lightweight kernel to be tailored to the hardware of a compute node, which is often much simpler than the hardware found in service and I/O nodes. For example, sometimes specialized or proprietary drivers are needed on I/O nodes to access the attached storage devices. Compute nodes on the other hand are often kept simple to reduce cost and power consumption. Their main components are often limited to one or more CPUs, local memory, and a network interface. OS functionality and device drivers for anything beyond that are not needed and often consume resources, such as memory and CPU cycles, if present.

Large parallel systems usually do not have local disks attached to compute nodes. That means the OSs on these nodes do not need to provide any file systems or device drivers for disks. File I/O requests by the application are converted into messages and sent off to the I/O nodes and external storage devices that handle them.

A large functional block in a modern OS deals with multiprocess management and scheduling. Algorithms to prioritize processes and to ensure fair scheduling are complex and must be able to deal with hundreds of concurrent processes. Since the nodes of a message-passing system are space shared, only one or a handful of processes run on each CPU. Some early lightweight kernels went so far as to allow only a single active process at a time and used cooperative scheduling to switch from one process to another. While this may seem like a throwback to an earlier time in OS design, it has several advantages: Expensive context switches are few, frequent and disrupting timer interrupts are not needed, and the code section for process control inside the kernel becomes much simpler. A desktop OS interrupts

running applications from 100 to 1,000 times per second to check whether there are any administrative tasks for it to do and to potentially context switch to another process. A typical lightweight kernel receives ten interrupts per second and is much quicker in determining whether a context switch is needed. Other than possibly sending a heartbeat signal to a Reliability, Availability, and Serviceability (RAS) system, a lightweight kernel has no other administrative tasks while it is running an application. This greatly reduces the noise signature of a lightweight kernel and gives more CPU cycles to the application.

Another large functional block in a full-featured OS deals with demand paging. Demand paging, even if local disks are available, is bad for the performance of a parallel system. While this feature can be turned off, a lot of the underlying data structures and assumptions remain, even when not in use. For example, page sizes are small to allow the OS more flexibility in allocating and releasing memory, and demand paging. Furthermore, pages that are adjacent to each other in the virtual address space are usually scattered all across physical memory. This increases the number of Translation Look-aside Buffer (TLB) entries needed and limits data transfers to and from most network interfaces to a single page at a time.

A lightweight kernel does not support demand paging and can manage memory as a set of physically contiguous large pages. This leads to performance improvements because the TLB entries are never exhausted, since the few page table entries that are needed usually fit in the TLB. Further performance improvements come from larger data transfers between the network interface and the memory. Large transfers are more efficient and lead to higher bandwidth.

A somewhat related problem has to do with shared and dynamically loaded libraries. Many lightweight kernels require statically linked applications. This is counter to modern software development strategies and is one of the main complaints about lightweight kernels. The processes of a parallel application are fairly well synchronized and reach a given section of code at about the same time. If at that point a shared library needs to be loaded, thousands or hundred thousands of PEs request that particular library. This leads to a massive bottleneck when all these requests converge at a single point in the file system. Because of that, and because of

the simplified way of dealing with memory, lightweight kernels are usually prevented from offering shared and dynamically loaded libraries.

In a message-passing system, moving data from the network into memory has to be done with as little overhead as possible. Traditionally, network interfaces deposit incoming data into buffers managed by the OS. When the application requests the data, the OS copies it from the buffer into the memory location the application requested. This technique works well when the network is much slower than the memory system, and when data packets are small. Modern parallel computers have networks that are often as fast or faster than the memory can copy data, and many parallel applications transmit large messages. Moving data from the network interface to memory and then again from one memory location to another causes delays in message delivery and consumes valuable memory bandwidth.

It is therefore advantageous for the network interface to move incoming messages directly into the memory location where the application expects the data. There are several requirements to make this possible. The Application Programming Interface (API) must allow pre-posting of receives. This allows an application to inform the OS or the network interface where future messages can be deposited. The traditional socket API used for TCP/IP programming does not provide this functionality; data is copied after it has arrived and the application has issued a receive request. The Message Passing Interface (MPI) API was designed for parallel computing and enables preposting of receive requests.

Sophisticated network interfaces used in message-passing systems have the ability to deliver data anywhere in local memory and can filter and direct incoming messages according to criteria such as which process the message is destined for, which process sent the message, its length, and a set of tag bits that can be used by the application to route messages to specific memory locations. To make optimal use of such network interface capabilities, the OS must allow incoming messages to be handled by the hardware without expensive OS interrupts. This is called *OS bypass*. The goal is to avoid invoking the OS on standard message deliveries. However, the OS is still responsible for protection and must be invoked during initial setup, perhaps while the application is initializing. Even with OS bypass it must not be

possible for unrelated processes to harm each other or access each others' private data.

Because a lightweight kernel can arrange physical pages in sequential order, the hardware in a network interface that is required to deliver messages larger than a page size can be much simpler; the network interface does not need its own virtual memory controller. In such a system, there is a one-to-one mapping of virtual to physical addresses which makes it easier to designate a memory region for delivery of messages: Only a starting address and a length are required. This further reduces the hardware complexity of the network interface and makes interaction of an application directly with the network interface more efficient and simpler.

The goal of a lightweight kernel is to turn over most hardware resources of a node to the application, while maintaining protection barriers between unrelated processes and protecting the rest of the system from errant or malicious processes. It is possible to design, implement, and maintain a lightweight kernel with a small group of programmers in a short amount of time. However, the talent pool to perform this kind of work is limited, and lightweight kernels do not offer some features that modern parallel applications demand. Whether these demands make sense on extremely large-scale systems is debatable.

### The Next Big Challenge: Resilience

Future high-end systems are expected to grow in size and part count. Because of that it is also expected that failures in software and hardware will become more common. For large applications that use a significant number of nodes in such a machine, that means they will almost certainly be interrupted by a failing component before the computation has a chance to complete. To nonetheless make progress, applications save their state and intermediate results frequently to disk. During a restart after an interruption, that data is read back from the disk and the application continues where it has left off. This is called *checkpointing* or *checkpoint/restart*.

Writing a checkpoint is time consuming, especially when many nodes do it at the same time. Delaying the writing of checkpoints risks that too much work is lost and has to be redone after an interruption. Current models predict that, even with optimal checkpoint intervals, when job sizes approach 100,000 nodes, applications will spend more than 50% of their

total execution time writing checkpoints, restarting, and redoing lost computations. Clearly, that is not an efficient use of such large and expensive machines. Research into making applications more resilient is under way.

Checkpoint/restart can be done automatically by the OS since the OS knows what portions of memory a process is using. There might be data structures the application uses which it can reconstruct easily without having to save and restore them. Without hints from the application, the OS will have to checkpoint these data structures, making automatic checkpointing somewhat less efficient than application checkpointing or application-directed checkpointing. Before a checkpoint can be written, the application must quiesce to ensure data does not change while checkpointing is in progress. That means no messages must be in transit during that time. Again, this can be done at the user level or with the help of the OS.

In the future, OSs may provide additional features to aggregate checkpoint data and to trigger checkpoints when a failure is imminent. Aggregating checkpoint data from multiple nodes on intermediate nodes before it is sent off to disk storage may allow an OS to schedule massive writes to storage and coordinate them with I/O from other applications currently running. Additional functions, such as data consolidation and compression, may also be feasible. Sometimes the onset of a failure can be predicted. For example, a stark increase in single-bit errors in an Error Correction Code (ECC) protected memory module indicates that an uncorrectable double-bit error is not far off. It makes sense to create a checkpoint at that time and restart the process on the failing node somewhere else in the system.

### Future Directions

The largest computing systems in the world have reached the petascale, and exascale systems are on the horizon. These are systems that will deliver performance in the thousands of petaflops. Early arrivals will look similar to today's large-scale systems. They will consist of thousands of nodes connected by a message-passing network. Each node will have multiple CPUs, each with tens or hundreds of cores. No one knows how these extreme scale systems will evolve after that.

Furthermore, application developers are hoping for a new programming model beyond the traditional MPI

and C/C++/Fortran interface. But, again, there is no consensus yet what that will be. Depending on the architecture of these future systems and how they will be used, OS technology needs to evolve as well. It is possible that some of the protection mechanisms of today's OSs will be implemented in hardware. Whole sections of a machine can then be turned over to an application, to be used in the fashion that provides the most benefit. Commonly used functions will be in libraries, and system control utilities interact with the hardware directly. Since the future is so uncertain, it is more likely that OSs and runtime systems take on an even bigger role to support the new hardware and provide new usage models to applications. For that, it may be possible that whole cores or CPUs will be dedicated to running portions of the OS and to accelerate message passing.

Some applications can make use of the compute capabilities a GPU provides. In the future, more systems will be delivered with GPUs and other specialized processors to accelerate computing. Again, programming and usage models for such systems are still emerging and the role OSs will play to make these accelerators accessible is unclear at the moment.

## Related Entries

- [Checkpointing](#)
- [Clusters](#)
- [Distributed-Memory Multiprocessor](#)
- [Flynn's Taxonomy](#)
- [Linda](#)
- [MIMD \(Multiple Instruction, Multiple Data\) Machines](#)
- [Metrics](#)
- [Numa Caches](#)
- [Nonuniform Memory Access \(NUMA\) Machines](#)
- [File Systems](#)
- [Race Conditions](#)
- [Scheduling Algorithms](#)
- [Shared-Memory Multiprocessors](#)
- [Single System Image](#)

## Bibliographic Notes and Further Reading

Many of the functions and services an OS for a parallel system provides are the same as the ones in most modern OSs. Standard textbooks, such as [8, 12, 13],

provide good introductions to the various general OS topics mentioned in this article. Understanding some of the hardware and the architecture of computers in general and parallel systems in particular is also necessary to understand OSs. The best book for that is [5]. A good introduction to the issues involved in designing and evaluating a parallel file system is [6].

The evolution of a family of lightweight kernels and the experience from developing and using them is described in [10]. The Compute Node Kernel (CNK) in use on IBM's Blue Gene machines is discussed in [7]. Puma [14] and its predecessor, the Sandia/University of New Mexico OS (SUNMOS) [11], were lightweight kernels for early massively parallel machines such as the nCUBE 2, the Intel Paragon, and Intel's ASCI Red at Sandia National Laboratories.

Performance comparisons between a full-featured OS and a lightweight kernel can be found in [2, 3]. Another comparison of an earlier system is in [11]. The experiences designing and deploying one of the earliest large clusters, including the system software required to do it, are chronicled in [1]. An explanation of the partitioning of a system into service, compute, and I/O partition is also in [1]. A brief overview of SSI is presented in [4] and a more thorough treatment is in Chap. 11 of [9].

## Bibliography

1. Brightwell R, Fisk LA, Greenberg DS, Hudson T, Levenhagen M, Maccabe AB, Riesen R (2000) Massively parallel computing using commodity components. *Parallel Comput* 26(2–3):243–266
2. Brightwell R, Maccabe AB, Riesen R (2003) On the appropriateness of commodity operating systems for large-scale, balanced computing systems. In: International parallel and distributed processing symposium (IPDPS '03), Nice. IEEE Computer Society, Washington, DC
3. Brightwell R, Riesen R, Underwood K, Bridges PG, Maccabe AB, Hudson T (2003) A performance comparison of Linux and a lightweight kernel. In: IEEE international conference on cluster computing, Hong Kong, pp 251–258
4. Buyya R, Cortes T, Jin H (2001) Single system image. *Int J High Perform Comput Appl* 15(2):124–135
5. Hennessy JL, Patterson DA (2006) Computer architecture, 4th edn. Morgan Kaufmann, Boston
6. May JM (2001) Parallel I/O for high performance computing. Morgan Kaufmann, San Francisco
7. Moreira JE, Almási G, Archer C, Bellofatto R, Bergner P, Brunheroto JR, Brutman M, Castaños JG, Crumley PG, Gupta M, Inglett T, Lieber D, Limpert D, McCarthy P, Megerian M, Mendell M, Mundy M, Reed D, Sahoo RK, Sanomiya A, Shok R, Smith B,

- Stewart GG (2005) Blue Gene/L programming and operating environment. *IBM J Res Dev* 49:367–376
8. Nutt G (2003) Operating systems, 3rd edn. Addison Wesley
  9. Pfister GF (1998) In search of clusters: the ongoing battle in lowly parallel computing, 2nd edn. Prentice-Hall, Upper Saddle River
  10. Riesen R, Brightwell R, Bridges PG, Hudson T, Maccabe AB, Widener PM, Ferreira K (2009) Designing and implementing lightweight kernels for capability computing. *Concurr Comput Pract Exp* 21(6):793–817
  11. Saini S, Simon HD (1994) Applications performance under OSF/1 AD and SUNMOS on Intel Paragon XP/S-15. In: Proceedings of the 1994 ACM/IEEE conference on supercomputing, Supercomputing '94, Washington, DC. ACM, New York, pp 580–589
  12. Silberschatz A, Galvin PB, Gagne G (2004) Operating system concepts, 7th edn. Wiley
  13. Tanenbaum AS (2007) Modern operating systems, 3rd edn. Prentice Hall
  14. Wheat SR, Maccabe AB, Riesen R, van Dresser DW, Stallcup TM (1994) PUMA: an operating system for massively parallel systems. *Sci Program* 3:275–288

## Overdetermined Systems

- [Linear Least Squares and Orthogonal Factorization](#)

## Overlay Network

- [Peer-to-Peer](#)

## Owicki-Gries Method of Axiomatic Verification

CHRISTIAN LENGAUER  
University of Passau, Passau, Germany

## Optimistic Loop Parallelization

- [Speculative Parallelization of Loops](#)

## Orthogonal Factorization

- [Linear Least Squares and Orthogonal Factorization](#)

## OS Jitter

- [Cray T3E](#)

## OS, Light-Weight

- [Operating System Strategies](#)

## Out-of-Order Execution Processors

- [Superscalar Processors](#)

## Definition

Axiomatic verification, although a general term, refers usually to the method, introduced by Bob Floyd [4] and Tony Hoare [7] for sequential programs, of proving that a program satisfies a specification formulated as a predicate pair of a pre- and a postcondition. Precondition  $P$  states the assumptions made of the program variables before program  $S$  executes. Postcondition  $R$  characterizes the state of the program variables after program  $S$  terminates, if it does (partial correctness). Sometimes, termination is part of the proof obligation (total correctness). The proof obligation is usually denoted by the so-called *Hoare triple*  $\{P\} S \{R\}$ .

The verification method proceeds by inserting intermediate assertions between the program's statements and proving the resulting Hoare triples, based on a number of axiomatic inference rules. Susan Owicki and David Gries extended the axiom system to parallel programs with shared memory [13, 14].

## Discussion

### Overview

Hoare's axiom system, also called *Hoare logic*, provides one axiomatic inference rule for each of the following language constructs: empty statement, assignment ( $:=$ ), sequential composition ( $;$ ), binary deterministic choice (**if**), and iteration (**while**). Two further rules are

needed to make propositional logic work properly. An axiomatic inference rule is written as follows:

$$\frac{\text{Obligations to be proved}}{\text{Conclusion that can be inferred without proof}}$$

The reader of this entry is expected to have a basic knowledge of Hoare logic.

The extension to parallel programs comes in two flavors: a general setting [13] and a restricted setting [14]. The general setting allows for more concurrency but burdens the programmer with, possibly, complex proof obligations. The restricted setting takes some of the burden of proof away from the programmer and relegates it to the compiler, at the expense of a possible loss of concurrency.

In each of the two settings, there is one language construct that introduces parallelism (parallel composition) and one that curtails it (synchronization and exclusion).

## The General Setting

### The Axioms

- *Parallel composition:*

$$\frac{\{P_1\} S_1 \{R_1\}, \dots, \{P_n\} S_n \{R_n\} \text{ are interference-free}}{\{P_1 \wedge \dots \wedge P_n\} \text{ cobegin } S_1 // \dots // S_n \text{ coend } \{R_1 \wedge \dots \wedge R_n\}}$$

The  $S_i$  are called *processes* and are executed in parallel. The  $n$  proof obligations concern the respective processes in isolation. Interference freedom is explained below.

- *Synchronization and exclusion:*

$$\frac{\{P \wedge B\} S \{R\}}{\{P\} \text{ await } B \text{ then } S \text{ end } \{R\}}$$

This rule rests on the assumption that the **await** statement is implemented as a test-and-set operation, i.e., the successful test of  $B$  is followed directly by the execution of  $S$  and all of this must proceed without interference. A comparison with the rule for the **if** statement [7] reveals that **await** does not require a poststate satisfying  $R$  if  $B$  is false, while **if** does. Instead, **await** suspends execution until some parallel statement makes  $B$  true.

The **await** statement has recently received support via the concept of transactional memory (see the corresponding encyclopedia entry).

### Interference Freedom

Shared variables are protected only inside an **await** statement but can also appear elsewhere in the program. To ensure that unprotected shared variables are being treated properly, one must prove that an assignment to a shared variable in one process does not affect the assertions made in the proof of any other parallel process. Thus, if execution of  $S_i$  falsifies neither any precondition of a statement in  $S_j$  nor the postcondition of  $S_j$ , then execution of  $S_i$  cannot interfere with the *proof* of  $S_j$ , and the proof of  $S_j$  is valid even in the face of parallel execution of  $S_i$  with  $S_j$ .

More formally, the proof obligation is as follows:

- Consider every process  $S_i$  of the parallel statement and every statement  $S$  of  $S_i$  that is not inside the body of an **await**.
- For every assertion  $Q$  in the proof of some process other than  $S_i$ , but not in the body of an **await**, prove  $\{Q \wedge \text{pre}(S)\} S \{Q\}$ .

This property is called *interference freedom*. The concept of interference freedom brought the complexity of correctness proofs of parallel programs down to a manageable size. For a program with  $n$  processes, each with  $m$  statements, the computational complexity of the proof obligation is in  $O(n^2 m^2)$ . Further, in many cases, the proof of each process can be organized such that many of the assertions can be written in terms of an invariant, which reduces the proof obligation further to showing that this invariant is not interfered with.

The validity of the noninterference argument rests on two further assumptions:

- *At-most-once property:* Every expression  $e$  in the program accesses at most one nonlocal, mutable variable and does so at most once. In every assignment  $x := e$ , expression  $e$  is at-most-once and  $x$  is local, or  $x$  is a simple variable (i.e., not a data aggregation like an array) and  $e$  accesses no nonlocal, mutable variable.

For example, if  $x$  is nonlocal,  $x := x + 1$  is not at-most-once, since  $x$  is once read and once written. However, with local variable  $t$ , in the equivalent program  $x := t; t := x + 1$ , both assignments are at-most-once. In this way, any program violating the at-most-once property can be refined to one satisfying it.

- *Memory interlock:* Parallel accesses to a fixed memory cell are serialized by hardware.

The **await** statement is a device to exclude portions of the program from the obligation of a noninterference proof. Assertions placed inside the body of an **await** are presumed to be shielded from interference.

The shorthand  $[S]$  stands for the atomic statement: **await true then S end**.

## Auxiliary Variables

The above rules do not guarantee the relative completeness of the method, i.e., that every correct program is provable (modulo the incompleteness of integer arithmetic due to Gödel). For example, one cannot prove that

**cobegin**  $[x := x+1] // [x := x+1]$  **coend**

increases variable  $x$  by 2, while one can prove that

**cobegin**  $[x := x+1] // [x := x+2]$  **coend**

increases  $x$  by 3 (see the example proofs below). The reason is that the two processes in the former parallel statement are indistinguishable, so the individual progress made by each process cannot be accounted for. In order to make it recognizable, one must introduce auxiliary variables (also called history variables) that log the individual progress but do not influence the flow of data or control of the program that is subject to proof. Then, one must assume by axiom that the proof, which holds for the program with auxiliary variables, also holds for the program without auxiliary variables.

- *Auxiliary variable:*

A set  $AV$  of auxiliary variables of a program contains only variables  $x$  that appear in assignments  $x := e$ , where  $e$  may contain any kind of variable (auxiliary or not).

- *Auxiliary variable axiom:*

Let  $S$  be a program with auxiliary variables and  $S'$  the corresponding program with all assignments to auxiliary variables removed.

$$\frac{\{P\} S \{R\}}{\{P\} S' \{R\}}$$

There is an algorithm that introduces into a program the necessary auxiliary variables to conduct the

proof [12]. It floods the program with auxiliary variables. Typically, the programmer takes up the intellectual challenge instead and inserts a few, well-chosen variables.

The auxiliary variable axiom is very powerful. It also supports the removal of **await** statements, should they become unnecessary after the removal of auxiliary variables.

## Total Correctness

For total correctness, i.e., termination (or at least progress for a nonterminating program), three extra steps are necessary. First, termination functions must be provided for loops in order to prove that the individual processes terminate. Second, it must be shown that the parallel composition does not cause any interference with these termination functions. Third, it must be proved that the **await** statements do not cause deadlock in the parallel composition. Owicky and Gries provide a sufficient, static proof condition for the absence of deadlocks [13].

## Examples

*Claim:*

$$\{x = 0\}$$

**cobegin**

$$[x := x+1] // [x := x+2]$$

**coend**

$$\{x = 3\}$$

*Proof outline:*

$$\begin{aligned} & \{x=0\} \\ & \{(x=0 \vee x=2) \wedge (x=0 \vee x=1)\} \\ & \text{cobegin} \\ & \quad \{x=0 \vee x=2\} \qquad \{x=0 \vee x=1\} \\ & \quad [x := x+1] \qquad // \qquad [x := x+2] \\ & \quad \{x=1 \vee x=3\} \qquad \{x=2 \vee x=3\} \\ & \text{coend} \\ & \quad \{(x=1 \vee x=3) \wedge (x=2 \vee x=3)\} \\ & \quad \{x=3\} \end{aligned}$$

Noninterference argument:

$$\begin{aligned} & \{(x=0 \vee x=1) \wedge (x=0 \vee x=2)\} [x := x+1] \{x = 0 \vee x = 1\} \\ & \{(x=2 \vee x=3) \wedge (x=0 \vee x=2)\} [x := x+1] \{x = 2 \vee x = 3\} \\ & \{(x=0 \vee x=2) \wedge (x=0 \vee x=1)\} [x := x+2] \{x = 0 \vee x = 2\} \\ & \{(x=1 \vee x=3) \wedge (x=0 \vee x=1)\} [x := x+2] \{x = 1 \vee x = 3\} \end{aligned}$$

Note that some noninterference must be proved, even though the two processes are fully exclusive: their respective pre- and postconditions must still be checked.

To prove the following program, auxiliary variables and an invariant to relate them are necessary.

*Claim:*

$$\{x = 0\}$$

**cobegin**

$$[x := x+1] // [x := x+1]$$

**coend**

$$\{x = 2\}$$

*Proof outline:* The invariant is  $I: x=y+z$

$$\begin{array}{c} \{x=0\} \\ y := 0; z := 0 \\ \{y=0 \wedge z=0 \wedge I\} \\ \text{cobegin} \\ \{y=0 \wedge I\} \quad \{z=0 \wedge I\} \\ [x := x+1; y := 1] \quad // \quad [x := x+1; z := 1] \\ \{y=1 \wedge I\} \quad \{z=1 \wedge I\} \\ \text{coend} \\ \{y=1 \wedge z=1 \wedge I\} \\ \{x=2\} \end{array}$$

Noninterference argument:

$$\begin{aligned} &\{(z=0 \wedge I) \wedge (y=0 \wedge I)\}[x := x + 1; y := 1] \{z=0 \wedge I\} \\ &\{(z=1 \wedge I) \wedge (y=0 \wedge I)\}[x := x + 1; y := 1] \{z=1 \wedge I\} \\ &\{(y=0 \wedge I) \wedge (z=0 \wedge I)\}[x := x + 1; z := 1] \{y=0 \wedge I\} \\ &\{(y=1 \wedge I) \wedge (z=0 \wedge I)\}[x := x + 1; z := 1] \{y=1 \wedge I\} \end{aligned}$$

Our final example is again the two processes  $x := x+1$  and  $x := x+2$  but with the exclusion lifted. Thus, they must be made at-most-once. The proof of partial correctness succeeds, except that the proof obligation of noninterference cannot be met.

*Claim:*

$$\{x = 0\}$$

**cobegin**

$$t := x+1; x := t // u := x+2; x := u$$

**coend**

$$\{x = 3\}$$

*Proof outline:*

$$\begin{array}{c} \{x=0\} \\ \{(x=0 \vee x=2) \wedge (x=0 \vee x=1)\} \end{array}$$

**cobegin**

$$\{x=0 \vee x=2\} \quad \{x=0 \vee x=1\}$$

$$t := x+1 \quad u := x+2$$

$$\{t=1 \vee t=3\} \quad // \quad \{u=2 \vee u=3\}$$

$$x := t \quad x := u$$

$$\{x=1 \vee x=3\} \quad \{x=2 \vee x=3\}$$

**coend**

$$\{(x=1 \vee x=3) \wedge (x=2 \vee x=3)\}$$

$$\{x=3\}$$

Noninterference argument:

$x := u$  affects the postcondition of  $x := t$ , i.e., the following proof obligation is not a valid Hoare triple:

$$\{(x=1 \vee x=3) \wedge (u=2 \vee u=3)\} x := u \{x=1 \vee x=3\}$$

The same holds vice versa. There is interference: the program is not correct.

## The Restricted Setting

### The Axioms

- *Parallel composition:* Let  $r$  be a set of program variables.

$$\frac{\{P_1\} S_1 \{R_1\} \wedge \dots \wedge \{P_n\} S_n \{R_n\} \wedge \text{FREE} \wedge \text{RES}}{\{P_1 \wedge \dots \wedge P_n \wedge I(r)\}}$$

$$\text{resource } r : \text{cobegin } S_1 // \dots // S_n \text{ coend} \\ \{R_1 \wedge \dots \wedge R_n \wedge I(r)\}$$

*FREE:* no variable free in  $P_i$  or  $R_i$  is being changed by  $S_j$  ( $j \neq i$ )

*RES:* invariant  $I(r)$  refers only to variables in  $r$

- *Synchronization and exclusion:*

$$\frac{\{P \wedge B \wedge I(r)\} S \{R \wedge I(r)\}}{\{P\} \text{with } r \text{ when } B \text{ do } S \text{ end } \{R\}}$$

The **with-when** statement is a conditional critical region, i.e., it is executed under mutual exclusion.

The rules in this setting work only if the following syntactic restrictions are obeyed:

- Any process may access a resource variable only inside a critical region protecting this variable.
- Any variable changed by process  $S_i$  may be accessed by process  $S_j$  ( $j \neq i$ ) only if it is a resource variable.

Thus, while shared variables must be in the resource, local variables need be in the resource only if the invariant refers to them.

These restrictions can be checked by a compiler. This check takes the place of the proof of interference freedom necessary in the general setting.

### Example

In the following claim, the shorthand  $[S]$  stands for **with  $r$  when true do  $S$  end**.

*Claim:*

$\{x = 0\}$

**resource  $r(x)$ :**

**cobegin**  $[x := x+1] // [x := x+2]$  **coend**

$\{x = 3\}$

*Proof outline:* The invariant is  $I(r) : x = y + z$

```

 $\{x=0\}$
 $y := 0; z := 0$
 $\{y=0 \wedge z=0 \wedge I(r)\}$
resource $r(x, y, z)$:
cobegin
 with r when true do $\{y=0\}$ with r when true do $\{z=0\}$
 $\{y=0 \wedge I(r)\}$ $\{z=0 \wedge I(r)\}$
 $x := x+1; y := 1$ $//$ $x := x+2; z := 2$
 $\{y=1 \wedge I(r)\}$ $\{z=2 \wedge I(r)\}$
 end end
 $\{y=1\}$ $\{z=2\}$
coend
 $\{y=1 \wedge z=2 \wedge I(r)\}$
 $\{x=3\}$

```

Variables  $y$  and  $z$  are accessed locally but, because the invariant refers to them, they must be part of the resource.

### Related Entries

- [Monitors, Axiomatic Verification of](#)
- [Formal Methods-Based Tools for Race, Deadlock, and Other Errors](#)

### Bibliographic Notes and Further Reading

The Owicki–Gries verification method received much attention in the late 1970s and remains useful to this day. It provided the first rigorous treatment of the semantics of imperative shared-memory programs, and its notion of interference freedom is at the core of later

methods. The seminal idea was that the execution of one process should not interfere with the *proof of correctness of another process*, rather than with its execution.

The necessity of the auxiliary variable axiom demonstrated the price that one has to pay for the ability to reason first about processes individually and then about their parallel composition. If the parallel program is considered as a whole, there is no need for auxiliary variables [9, 10].

The Owicki–Gries method demonstrated that proofs of parallel programs are feasible but require an effort that is, in the worst case, quadratic in the number of statements across all processes that are being composed in parallel. This cannot be avoided if shared updates are allowed to happen without synchronization. Further, proofs can be engineered to use invariants in order to reduce the proof effort.

Soundararajan [15] was able to do without auxiliary variables and without the noninterference proof by introducing instead history variables into the assertions (so-called hidden variables [9]). Inherently, their manipulation is of similar complexity but involves no intuition, similarly to Owicki’s algorithm for “flooding” the program with auxiliary variables [12].

The technical complexity of the general setting revealed the intricacies of unbridled parallelism. Still, it played a significant role in helping to prove implementations of intricate algorithms like Dekker’s algorithm for mutual exclusion or the concurrent garbage collector for LISP data structures [5].

The restricted setting was laid out before Owicki and Gries by Hoare [8]. It is relevant in practice today for the verification of monitors. A monitor is an abstract data structure whose operations are conditional critical regions. See the corresponding encyclopedia entry.

The Owicki–Gries method was extended in at least two ways to address distributed-memory parallelism [1, 11]. The lack of shared updates in distributed programs raised the hope for simpler proofs. Unfortunately, the axiom system turned out to be more complex rather than simpler. First, an additional proof step is needed (the so-called cooperation proof [1] or satisfaction proof [11]): the communication operations **send** and **receive** must be matched to mutual pairs whose semantics is essentially the same as that of an assignment. Second, interference freedom may still have to be proved as well. All program variables are local but, in general, there is still the need for shared auxiliary variables.

A methodology for the development of parallel programs, which is similar to the one proposed for sequential programs by Dijkstra [2] and Gries [6], has been proposed by Feijen and van Gasteren [3]. It is based on the Owicki-Gries verification method.

## Bibliography

1. Apt KR, Francez N, de Roever WP (1980) A proof system for communicating sequential processes. *ACM Trans Program Lang Syst* 2(3):359–385
2. Dijkstra EW (1976) A discipline of programming. Series in automatic computation. Prentice-Hall, Englewood Cliffs
3. Feijen WHJ, van Gasteren AJM (1999) On a method of multiprogramming. Springer, New York
4. Floyd R (1967) Assigning meanings to programs. In: Schwartz JT (ed) Proceedings of symposium on applied mathematics, vol 19, American Mathematical Society, Providence, pp 19–32
5. Gries D (1977) An exercise in proving parallel programs correct. *Commun ACM* 20(12):921–930
6. Gries D (1981) The science of programming. Texts and monographs in computer science. Springer, Berlin
7. Hoare CAR (1969) An axiomatic basis for computer programming. *Commun ACM*, 12(10):576–580, 583
8. Hoare CAR (1972) Towards a theory of parallel programming. In: Hoare CAR, Perrott RH (eds) Operating systems techniques, A.P.I.C. studies in parallel processing 9, Academic, London, pp 61–71
9. Lengauer C (1982) A methodology for programming with concurrency: the formalism. *Sci Comput Program* 2(1):19–52
10. Lengauer C, Hehner ECR (1982) A methodology for programming with concurrency: an informal presentation. *Sci Comput Program* 2(1):1–18
11. Levin GM, Gries D (1981) A proof technique for communicating sequential processes. *Acta Informatica* 15(3):281–302
12. Owicki SS (1976) A consistent and complete deductive system for the verification of parallel programs. In: Proceedings of the Eighth Annual ACM Symposium on Theory of Computing (STOC'76), ACM, 1976, pp 73–86
13. Owicki SS, Gries D (1976) An axiomatic proof technique for parallel programs. *Acta Informatica* 6(4):319–340
14. Owicki SS, Gries D (1976) Verifying properties of parallel programs. *Commun ACM* 19(5):279–285
15. Soundararajan N (1984) A proof technique for parallel programs. *Theor Comput Sci* 31(1–2):13–29

# P

## Parafrase

BRUCE LEASURE  
Saint Paul, MN, USA

### Synonyms

Parallelization

### Discussion

Parafrase was a successful project allowing experimentation in source-to-source translation of FORTRAN programs. Parafrase was the work of David J. Kuck and his students at the University of Illinois at Urbana-Champaign. It was supported largely by NSF. The name of system is due to Stott Parker. A wide variety of optimization strategies were developed by researchers, and the performance achieved was measured using a broad cross section of applications, across a variety of theoretical and actual machines. Parafrase was successful because it reduced the effort required of the researcher to implement an optimization strategy, and to perform experiments.

Parafrase was structured much like a multi-pass compiler of the same era. The FORTRAN program being compiled was processed one routine at a time. A scanner consumed the source code for a FORTRAN routine and built an equivalent representation in the Internal Language (IL). Then a series of passes that transformed the IL in various ways was run. The researcher would select the passes based on the goals of the research. Then lastly, a FORTRAN code regenerator pass was run. This entire process was repeated for each of the routines in the input program. The FORTRAN program output by Parafrase could be compiled by a FORTRAN compiler and run on any machine.

Because the ultimate goal of Parafrase was to regenerate optimized FORTRAN code, the IL used by Parafrase was considerably different than the IL found in a compiler. The IL used by Parafrase was a relatively

straight forward representation of FORTRAN source, with some additional look-aside tables to hold summary information about variables and loop structures. Each pass was required to accept and generate this common IL.

Having an IL that was very close to FORTRAN enabled Parafrase to provide an easily understandable IL dump by simply pretty printing the FORTRAN program that was represented in the data structures. The pretty printer could be invoked by Parafrase at the end of a pass, or called by the pass itself to display intermediate results. This made it easy for the researcher to see what an optimization was doing because the researcher only had to understand FORTRAN, not the details of some unusual IL.

Parafrase also included an IL verification phase that could be enabled to run after any pass to ensure that that pass was following the rules. This was especially useful when identifying which pass was incorrectly processing a FORTRAN routine.

Because all of the passes were required to accept and generate the common IL, the passes could be run in any order. This flexibility allowed the researcher to construct a specialized ordering of the passes to achieve specific goals. Very early in Parafrase's life, the ability to specify the order of the passes in a text file was added. This enabled the researcher to adjust the order of the passes without having to rebuild the executable image of Parafrase.

Parafrase was implemented in PL/I and run on an IBM System/360 before virtual memory. With all of the passes and the limited amount of physical memory available, the executable had to be built using the technique of overlays. By placing each pass in its own overlay, the passes would all share the same memory address space, thus reducing the memory foot print. Of course, it was a little more complicated than that, and the obscure interface to IBM's link editor to build the overlays was understood by only a few people. A build tool was created for Parafrase to automatically

identify an appropriate overlay structure, to create the appropriate link editor commands, and to build the overlay executable, thus relieving the researcher from understanding the messy details of this process.

When researchers started to use Parafrase on more than just toy programs, it was quickly discovered that a scanner handling ANSI FORTRAN was not nearly strong enough. The various extensions to FORTAN promoted by computer manufacturers had changed common use. The only solution was to add these extensions to the Parafrase scanner, and to extend the IL where needed to include them. Eventually, VAX, CDC, HP, IBM, and other manufactures extensions were implemented. Some syntax extensions were just syntactic sugar, and were translated away by the scanner. Others required unique extensions to the IL.

Another unexpected discovery when processing non-toy programs was that most FORTRAN programs contained hand optimized code targeting a particular machine. For most of the research performed with Parafrase, the ideal input program was one that was not hand optimized.

Consider the case of exploring the impact of vectorization on a program. If the loops in the program that could be vectorized were unrolled by hand in the original code, then the vector length and the reference pattern for each vector operand would be different than if the loops were not unrolled in the original program. Vector lengths would be shorter, and operands

would be less likely to be contiguous. On most vector capable machines, these changes would seriously impact performance.

Consequently, a collection of de-optimization passes were written for Parafrase. It was impossible to write de-optimization passes for every hand optimization. There were just too many. Instead, we kept track of how many important loops exhibited a particular hand optimization, and whenever a sufficiently large number of loops exhibited a particular hand optimization, we wrote a de-optimization pass to address the issue.

Parafrase ended up having de-optimization passes and optimization passes that performed inverses of each other for many of the common optimizations. For example: Loop rerolling and loop unrolling, forward substitution and code floating, and loop distribution and loop fusion.

The research efforts supported by Parafrase dealt with optimization techniques for supporting new types of hardware acceleration devices: vector processors, streaming memory systems, cache memory systems, multiple functional units, parallel processors, and memory banks. At the time, the impact of any one of these acceleration devices on the performance of ordinary programs was not well understood.

The optimization techniques to automatically exploit these acceleration devices were not well understood. Parafrase provided a reasonable vehicle to explore optimization techniques.

**Parafrase. Table 1** Theses related to Parafrase

| Year | Author                       | Topics                                                                   |
|------|------------------------------|--------------------------------------------------------------------------|
| 1971 | Yoichi Muraoka               | Arithmetic expressions, loop dependence testing, wave fronts, scheduling |
| 1975 | Steve S.C. Chen              | m-th order linear recurrences                                            |
| 1976 | Ross A. Towle                | Dependence testing, loops with branching, parallel parsing time          |
| 1976 | Bruce Leasure                | Design of Parafrase                                                      |
| 1978 | Walid Abu Sufah              | Virtual memory optimization, name partitioning, loop reindexing          |
| 1979 | Utpal Banerjee               | Data dependence tests for multi-loop programs                            |
| 1979 | David A. Padua               | Clustered system loop transforms, loop pipelining, scheduling            |
| 1980 | Robert H. Kuhn               | Vector optimization, decision tree optimization                          |
| 1982 | Michael J. Wolfe             | Direction-vector based optimization, recurrences, while loops            |
| 1984 | Ron G. Cytron                | Doacross loop optimization and scheduling                                |
| 1985 | Alex Veidenbaum              | Blocks of assignment statements, coarse grain optimization               |
| 1986 | Constantine Polychronopoulos | Loop coalescing, subscript blocking, static and dynamic scheduling (GSS) |

The history of research can be traced from the early 1970s to the mid 1980s through thesis topics, shown in [Table 1](#), and published papers listed in the References. Twelve theses and 11 papers are listed from a larger collection of work on the Parafrase system.

## Bibliography

1. Kuck D, Muraoka Y, Chen SC (1972) On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Trans Comput C-21(12)*:1293–1310
2. Kuck D, Budnik P, Chen SC, Davis E Jr, Han J, Kraska P, Lawrie D, Muraoka Y, Strebendt R, Towle R (1974) Measurements of parallelism in ordinary FORTRAN programs. *Computer* 7(1):37–46
3. Kuck DJ (1976) Parallel processing of ordinary programs. *Adv Comput* 15:119–179, Rubinoff M, Yovits MC (eds). Academic Press, New York
4. Abu-Sufah W, Kuck D, Lawrie D (1979) Automatic program transformations for virtual memory computers. Proceedings of the 1979 national computer conference, AFIPS Press, June 1979, pp 969–974
5. Kuck DJ, Padua DA (1979) High-speed multiprocessors and their compilers. Proceedings of the 1979 international conference on parallel processing, Aug 1979, pp 5–16
6. Padua DA, Kuck DJ, Lawrie DH (1980) High-speed multiprocessors and compilation techniques, special issue on parallel processing. *IEEE Trans Comput C-29(9)*:763–776
7. Kuck DJ, Kuhn RH, Leasure B, Wolfe M (1980) The structure of an advanced vectorizer for pipelined processors. Proceedings of COMPSAC 80, The 4th international computer software and applications Conference, Chicago, IL, Oct 1980, pp 709–715
8. Kuck DJ, Kuhn RH, Padua DA, Leasure B, Wolfe M. Dependence graphs and compiler optimizations. Proceedings of the 8th ACM symposium on principles of programming languages (POPL), Williamsburg, VA, Jan 1981, pp 207–218
9. Cytron R, Kuck DJ, Veidenbaum AV (1985) The effect of restructuring compilers on program performance for high-speed computers. In: Duff IS, Reid JK (eds) Special issue of computer physics communications devoted to the proceedings of the conference on vector and parallel processors in computational science II, vol 37 Elsevier Science Publishers B V (North-Holland Physics Publ.), Oxford, England, pp 39–48
10. Lee G, Kruskal CP, Kuck DJ (1985) An empirical study of automatic restructuring of nonnumerical programs for parallel processors. Special issue on parallel processing. *IEEE Trans Comput C-34(10)*:927–933
- II. Constantine Polychronopoulos, Kuck D, Padua D (1989) Utilizing multidimensional loop parallelism on large-scale parallel processor systems. *IEEE Trans Comput* 38(9):1285–1296

## Parallel Computing

DAVID J. KUCK

Intel Corporation, Champaign, IL, USA

## Definition

Parallel computing covers a broad range of topics, including algorithms and applications, programming languages, operating systems, and computer architecture. Each of these must be specialized to support parallel computing, and all must be designed and implemented coherently to provide highly efficient parallel computations.

## Discussion

### Introduction and History

All computations transform data – logically and arithmetically – following a series of algorithms. The broad goals of parallel computing are to improve the speed or functional ease with which this is done. Speed and functionality goals can usually be met, but in many practical cases, difficulties arise that present far more complexity than does traditional sequential computing. The following gives an overview of background issues and the current state of parallel computing.

The long and diverse history of computing is intertwined with parallelism. There will be no attempt here to formulate a rigorous definition of parallel computing. Instead, an introduction to parallel computing can be provided by sketching the many uses for parallelism over time. Computer system *performance* has always been the key driver of *hardware parallelism*. On the other hand, *software parallelism* has been driven by both performance and application functionality.

The baseline of parallelism is serial computing, and in the beginning, some computers operated in bit-serial fashion. One bit, the minimal information unit in a computer, was processed at a time. To improve speed, multiple bits were fetched from memory and transformed in parallel, a process that continued up to word-level computation (32 or 64 bits). This required making all data paths in a system – memory, processor, and interconnection – one word wide. Fast parallel algorithms for individual arithmetic operations occupied the research agenda in early computing, into the 1960s.

## Parallel Communication Models

- [Bandwidth-Latency Models \(BSP, LogP\)](#)

Some early machines used one hardware (HW) technology to implement all parts of a computer, from processor to memory. Over time, as technology advanced and more performance was demanded, distinct technologies were used in different parts of machines – e.g., transistors for processing and magnetic cores for memory. As processor technology speeds increased more quickly than memory, parallel memory units were introduced to match the data rates (i.e., bandwidths measured in bits per second) of processors. As one part of the processor became a performance bottleneck relative to others, multiple units (e.g., 2 adders) were introduced within the processor.

The idea of performing more than one arithmetic operation in parallel is an obvious way to improve performance, and was mentioned by the first computer designer, Charles Babbage (who designed a mechanical digital computer in the 1840s [1, 2]). Eventually, reasonably balanced systems for various types of computation became well understood, and parallelism as above was employed to produce very efficient uniprocessor systems. *Simultaneous operation* on whole words and *overlap* of various operations are the basic principles that support parallelism.

After sufficient HW has been invested to produce a good architecture using various types of low-level parallelism, the clock speed (which determines the time of a computer's most basic steps) can be increased to speed up a system. From the beginning, new device technology allowed faster clocks; from the 1960s on, this and architectural refinements drove system performance. But at the high end of performance demands, the ability to change the internal structure of a system by simply adding low-level parallelism became increasingly difficult and yielded slower growth. Architectural refinements were added to keep systems balanced (e.g., cache memory hierarchies) as technology advances drove processor performance faster than off-chip memories.

## Parallel Architectures

System clock speed determines the bandwidth (bits/sec) of various computer elements, and the clock speed depends on the basic technology available. Physical issues underlie technology, including:

- Transistor density in an integrated circuit limits clock speed

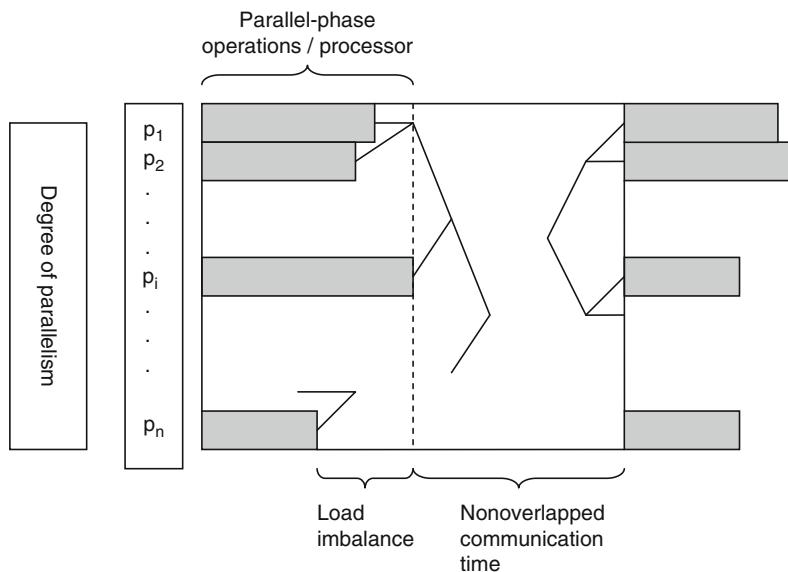
- The speed of light limits data transmission time or latency (measured in seconds)
- Manufacturing costs of HW limit the complexity that can be built into a system

To avoid limits imposed by the first and last of these, parallel processors can be used, each carrying out part of an overall computation. In other words, transistor density increases can be frozen as can the complexity of faster uniprocessor manufacturing, by moving to parallel processing. Although this forces the system size to grow, which in turn exacerbates latency issues, two of the three physical issues are controlled. The number of parallel units being used is often expressed as the *degree of parallelism*,  $n$ , see Fig. 1.

Parallelism also raises a new architectural issue: How does one interconnect the processors and the processors to memory? Ideally one would choose direct connection of each unit to all others (a crossbar switch), but this raises severe cost and design issues as the degree of parallelism grows. At the other extreme, the bus can be extended from within each processor to all others, a simple but low-performance idea. Between these extremes, one can choose a fast parallel ring, torus, or shuffle interconnect network pattern, each with many parallel data transmission paths and various performance and manufacturing trade-offs. For details and more references see [1, 3].

From the 1960s through the 1990s, a host of research and commercially available parallel systems were built, exploring a wide variety of architectural trade-offs. At the same time, beginning in the 1960s, several related approaches to computing emerged. Multiprocessing, the connection of whole computers to increase system throughput, began. The distinction can be captured by viewing *parallel computing* as turnaround computing – whose goal is the fast turnaround of one job at a time. In contrast, *multiprocessing* can be viewed as throughput computing – the goal being to complete a large volume of similar transactions quickly.

The third approach, *distributed computing* arose for a totally different reason [4]. Large companies bought multiple computer systems at different sites, and wanted to exchange results among them – e.g., financial information for the whole enterprise – via telephone line communications. Distributed computing in general puts less demand on communication than parallel computing or multiprocessing. By the late 1960s, this idea



**Parallel Computing. Fig. 1** Key parallel computation parameters

grew into the ARPANET among government contractors and agencies, and was followed by new concepts that led to the Internet.

At the HW technology level, transistor speeds increased continually and casualties developed – by the early 1990s ECL (emitter coupled logic) could not be pushed faster, and was supplanted by traditionally slower CMOS transistors. By the first decade of the twenty-first century, it became clear that silicon transistors, whose size continued to decrease (Moore's Law [5]) and whose power dissipation density concomitantly increased with clock speed, were hitting a “power wall.” The result was that clock speeds could no longer increase, even though densities could continue to grow, allowing more processors on a single semiconductor chip. Thus, parallel computing reached the mainstream, and by 2010 has become ubiquitous.

## Parallel Software Concepts

Over the decades of parallel architecture development, research into algorithms and parallel programming tools enjoyed broad growth. However, software (SW) application development is a very diverse subject, and parallel programming has not matured at the rate that parallel HW has arrived in the mainstream of computing. Roughly speaking, parallel application development suffers from all the problems inherent in sequential programming,

plus a number of additional problems arising from parallelism.

As is obvious from the release delays, security weaknesses, and other flaws in sequential applications, SW development methods are still maturing. Some decades after sequential computing HW reached its eventual complexity (except perhaps for cache hierarchy management), SW development remains labor intensive, with results that are error prone, and hard to test. Improving parallel SW development tools remains an important activity.

P

## Application Software Development

In 2010, various programming languages are able to capture any given algorithm in many forms. Some algorithms and applications are naturally parallel. For example, little dependence exists among tasks in multiprogramming and distributed computing, and any language that suits their problem domain will do. Other applications contain data and control dependences that appear to force sequential execution of program phases. While a continuum of ideas exists to deal with parallel programming, the following breakdown captures some key points, concluding with the unexploited parallelism remaining in useful parallel applications. This section is an introduction; more details appear in the following sections.

## Parallelization Methods

1. For some apparently nonparallel algorithms and programs, effective SW procedures lead to good parallel performance:
  - (a) Compilers can extract some parallelism from sequential constructs by language-level transforms [6].
  - (b) Parallel libraries allow developers to express parts of programs in parallel terms at the language level, without the need to understand the underlying parallel algorithms [7].
  - (c) For other program constructs, compilation is too hard and libraries are not available, so a language or algorithm change is necessary for compiler success, forcing the application developer to think through parallelism detail. Three programming models will be discussed in detail below as found in [8–10].

Thus, there are several ways to adjust some programs to be suitable for efficient parallel execution, but in other cases parallelism is harder to obtain.

## Parallelization Impediments

2. For the most difficult applications, algorithms and their programs, there are inherent, non-parallelizable issues:
  - (a) Data and control dependences arise that cannot practically be removed by a combination of compiler and run-time HW checks. For example, too much branching, or too many pointers or run-time parameter tests exist.
  - (b) Architecture-related clashes arise, e.g., there is too much nonoverlapped interprocessor communication or imbalance between the sizes of parallel tasks that waste parallel computation time, see Fig. 1.

In case 2, the best approach is to rethink the algorithms and program, and restructure the program or use entirely different algorithms. These steps can be very difficult and nonintuitive, unless a developer is specifically driven by the parallel architectural constraints imposed, understands parallel languages, and has a deep understanding of the underlying problem and various algorithms for solving it.

## Unexploited Parallelism in “Well-Parallelized” Applications

3. Consider the amount of “unexploited” parallelism remaining in an application that is already regarded as well parallelized for a given system. In other words, how far do parallel applications in use deviate from the perfect exploitation of parallelism in their algorithms? This question exposes language and compiler issues. Assume that an algorithm and program contain reduction operations in the form of Eq. 1, which sums the elements  $a_i$  of array  $a$  into scalar  $x$ . If the compiler cannot parallelize this,

$$x = 0; \quad \text{for all } i \quad (x = x + a_i) \quad (1)$$

then unexploited parallelism can be recovered by using a language (extension) that contains a parallel reduction operator. Much language research is directed toward filling such language gaps in ways that are easy to use in specific algorithms or application domains, e.g., games, image processing, and transaction processing.

Figure 1 summarizes several details that determine parallel system performance. The shaded areas represent useful work (their unequal lengths show that not all processor steps take the same amount of time), and the trees represent communication or synchronization, e.g., join/fork or message-passing patterns. Performance increases with the degree of parallelism and parallel-phase operations per processor, which increase the use of parallel resources. Performance decreases due to load imbalance and nonoverlapped communication, which waste resources relative to decreasing elapsed time in Eq. 2.

## System Architecture Performance Criteria

Let *computation time* be defined as a measure of the useful processing steps in a source program, i.e., corresponding to necessary steps in the original algorithms and their program counterparts. *Communication time* includes the total HW interconnect time and system SW overhead required in a program and its resulting computation to support the desired computation time.

Our overall objective is to *minimize total elapsed time*, in Eq. 2 (cf. Fig. 1).

$$\begin{aligned} \text{Total elapsed time} &= \text{Computation time} \\ &+ \text{Nonoverlapped communication time} \end{aligned} \quad (2)$$

In general, the diversity of parallel architectures, especially issues arising from major increases in processor count, change some of the inherent needs of algorithms to satisfy the constraints outlined above. Furthermore, each data size to which a given algorithm applies may force algorithm adjustments for machine size. Thus, for a given algorithm type, adaptations to machine parameters may be necessary.

As outlined above, algorithms, languages, run-time libraries, and architecture are closely linked in obtaining top performance. Good parallel performance across all types of problems can usually be obtained by an approach that includes all of these factors.

A universally important performance criterion that follows from Eq. 2 is shown in Eq. 3 (where *communication time* includes overlapped and nonoverlapped time).

$$\frac{\text{Computation time}}{\text{Communication time}} \geq 1 \quad (3)$$

Maintaining this inequality, averaged over time windows throughout a computation, means that the HW design and application SW running may be brought into balance by overlapping communication with meaningful computation.

The ratio can be used as a guideline in system and SW design. If the ratio of Eq. 3 is less than 1, increasing the numerator, while decreasing the denominator can reduce overall time and bring the HW use into balance. For example, if the ratio is less than 1, performance may be improved by using an algorithm with redundant operations that reduces the total data communication time. Such a trade-off causes no problems because the key ratio was too low to begin with.

## Parallel Performance

A successfully performing parallel computation must satisfy four basic requirements:

1. Algorithmic parallelism
2. Program parallelism
3. Data size and parallelism
4. Architecture balance and parallelism

These success requirements have many interpretations and interrelations, which are sketched below. They include mathematical issues at the algorithmic level, programming languages that well-suit important application areas, accommodating the data structures and

sizes that occur naturally, and then synthesizing these requirements into an architectural design that is well suited to efficient solution of the problems defined by the original applications. Many books discuss various aspects of performance for a range of algorithms and applications, ranging from broad coverage [11] to emphases on nonnumerical [12] to numerical [13] computations.

1. *Algorithmic parallelism*: The algorithms used must avoid dependences among operations that force one step to follow another. The essence of parallelism is to be able to do large numbers of operations simultaneously, and this must begin at the algorithm and data structure level. A simple case is an interactive program that has several sequential algorithms, each of which can be run simultaneously – e.g., a game may use three processors working independently on processing user input, readying the next frame, and displaying the current frame.

A more scalable algorithm that may require any number of processors is one that operates on arrays of many numbers, or files of many records. For example, if computing  $x_i$  requires  $a_i$  and  $b_i$ ,  $1 \leq i \leq n$ , as in Eq. 4, the process can be carried out for any number of values at once. If it requires  $x_{i-1}$ , as in Eq. 5, this is a linear recurrence, which

$$\text{for all } i (x_i = a_i + b_i) \quad (4)$$

$$x_0 = 0; \quad \text{for all } i (x_i = x_{i-1} + a_i) \quad (5)$$

appears to have sequential dependences but can be transformed and computed by reasonable parallel algorithms in  $O(\log_2 n)$  steps. On the other hand, nonlinear recurrences, e.g., Eq. 6 ( $1 \leq i \leq n$ ), usually present insurmountable mathematical problems

$$x_{-2} = 2; x_{-1} = 1; x_0 = 1; \quad \text{for all } i \left( x_i = \frac{a_i(x_{i-1})^2 x_{i-2}}{b_i x_{i-3}} \right) \quad (6)$$

for parallelization, so a different algorithm must be found.

2. *Program parallelism*: Given a set of parallel algorithms, expressing them effectively in an appropriate parallel language is relatively straightforward. However, the structure of an algorithm and architecture

will force the choice of language, as the following example illustrates.

*Game Example:* Consider the many algorithms necessary to support an interactive computer game. At the highest level, communication from the players must be coordinated with the internals of the game. These include a changing scene driven by the game logic as well as modifications made by players' actions. High-quality graphics depends on computationally intense solutions of complex physics equations for mechanical structures, fluid flow, cloth and hair motion, etc. An important consideration for game writers is how to use all of the computing power available to create appealing graphics and game experiences. Finally, each new frame must be composed and displayed. All of this must occur in real time.

Three *programming models* form the core of most programming languages and methods (cf. Fig. 1).

- (a) *Message passing* among communicating sequential (or parallel) processes may be the appropriate programming model at the highest level of game SW. Each process runs under independent control of the operating system on one or more processors, with the added feature that processes occasionally may send data to, and receive data from, one another. To keep the program logic valid, the system SW must provide for processes interrupting one another, and later expecting acknowledgements back from other processes. Communication delays are typical of performance problems in message-passing programs. A good exposition of MPI, a popular message-passing language for scalable systems, is found in [9].
  - (b) The *fork-join parallel* model matches well with the execution of high-level tasks internal to a game. Any number of tasks can be dispatched with fork statements to separate processors, in the form of threads, and when their independent work is finished, they combine in join statements. *Threads* share memory and may be controlled by each application without OS intervention in scheduling. The degree of parallelism is limited by the number of parallel tasks. For a description of OpenMP and its use in shared memory programming, see [8].
  - (c) *Data parallel languages*, embody a third type of program parallelism. They normally express a great deal of low-level parallelism in the data, e.g., application-specific array oriented languages can be used to express parallelism directly in the form required by an application algorithm. Equation 4 is an example, and may occur in rendering an image for display, where each pixel is computed independently of all others. Another example is movie making, where each frame can be rendered independently of the others. An introduction to Cuda and OpenCL, two recent data parallel languages, is given in [10].
3. *Data size and parallelism:* The degree of parallelism is also limited by data dependence and data size. The number of independent data structures and the size of each are indicators of the degree of available parallelism in a computation. A successful parallel computation requires data locality in that program references stay relatively confined to the data available in each processor – otherwise too much overhead time can be consumed, ruining parallel performance.
- The number of *parallel-phase operations per processor* must be sufficient to dominate the communication time per phase, see Fig. 1. Equation 3 can be used to explain the reason why. In parallel algorithms, communication among processors (or processor-memory communication) occurs with some frequency; in the easy case, the frequency is low. But as the frequency grows, so does the communication in any time window. Equation 3 says that as the communication time exceeds computation time, performance can suffer. If there is little data per processor, then the computation time window without communication is expected to be small. Thus a small numerator leads to a large denominator, driving the ratio to violate the inequality.
4. *Architecture balance and parallelism:* The architecture must have a sufficient number of processors, sufficiently fast global memory access, and interprocessor communication of data and control information that allows parallel scalability. As systems grow in degree of parallelism, the memory and communication systems define the success of a design.

Of course, these qualitative statements imply many difficult technical design choices (mentioned earlier) that must provide a good match to the programs and data sets to be run. In systems with low degrees of parallelism, including modern multicore chips, all of the processors generate addresses for a single, shared address space. This *shared-memory* parallelism (SMP) allows them to communicate quickly and simply via memory. But as the degree of parallelism increases and parallelism grows, *distributed-memory* parallel (DMP) systems are used.

Whether DMP clusters of smaller SMP systems occupy a single large computer room, or span different cities, large systems have nonuniform memory access time (NUMA), so called because of the very large range of overheads involved in communicating with local (SMP) and global (DMP) memory. These systems present greater programming challenges, in order to minimize and overlap communication and computation time (recall Eq. 2) and to schedule tasks properly (load balancing) to minimize wasted resources (cf. Fig. 1). The great variety in real-world computations has led designers down many successful paths from the 1980s to the present [14, 15].

## Parallel Program Correctness

Correctness is harder to determine for a parallel program than for a sequential program, and fixing bugs is concomitantly much harder. Whole new classes of bugs arise in parallel programs, for various reasons. A major problem that arises at the hardware level is *non-determinism* of execution. Because the system is large and some clocks may be out of sync, slight timing variations may exist from run to run of a given program.

Even if in sync, different interactions with the OS (e.g., on which processor a process is scheduled) or with other applications, or slightly different data-dependent execution paths can cause a program to execute in different ways each time it runs. These nondeterministic issues make error states nearly impossible to capture or recreate. This is obvious for reactive or interactive programs, whose inputs are difficult to recreate exactly (precisely when did a keystroke that triggered a process occur?).

In some parallel programs, synchronization points occur to coordinate the work of distinct processors, e.g.,

at *join* points and at the end of *data parallel* operations. These are necessary to ensure that the computations on all processors are complete for a given parallel phase of the overall program, before continuing. In some programming languages, message passing is used to send data and synchronize control of various processors. This is common in distributed memory machines. Software developers often find it hard to get the logic of such programs exactly right, given the variety of inputs some programs have, and the fact that multiple developers may have worked on a given section of code.

The logic of some programs may require *critical sections*, in which a data structure is accessed by only one computational process at a time. For example, if work tasks are being chosen from a queue, to assure that each task is assigned to only one processor, the task assignment algorithm would include a critical section. Software *locks* may be used to allow just one process at a time to execute a critical section.

When combinations of the above occur, various anomalies may arise, including program *deadlock*, in which, e.g., two processors halt, each waiting for the other. In what is perhaps the worst case, poorly synchronized programs actually do not halt, but spend so much time waiting for one another and occasionally proceeding, that the effect is to lead naïve developers to confuse a complex correctness bug with a performance problem. The result may be fruitless developer time spent using performance tools and techniques to find a correctness problem, which needs other approaches.

## Future

The era of ubiquitous parallel processing hardware has arrived, but the development of parallel application SW is lagging. In 2010, single chips contain eight shared memory processors (or “cores”). Distributed memory clusters of these, containing from 32 to over 100 K processors are the state of the art in server computers. The use of clustered systems ranges from scientific research and engineering design, to Internet search engine support.

While their clock speeds are frozen, the core count in future chips will continue to provide more performance, based on better architecture, parallel applications algorithms, and SW. Computers based on new technologies and principles (e.g., quantum or biological

principles) could change all of this in some application areas, but it is most likely that parallel SW and architectural issues will continue to dominate much of system design for many decades.

There is a seldom mentioned issue in parallelism. After the challenges of getting all current applications well parallelized are met, how does performance continue to grow? There are only two ways that more parallel processors can produce more performance: larger data sets, or more complexity in algorithms for today's applications. Both of these factors have driven much of computing in the past, but clearly there are limitations for some applications in the future. Those that cannot grow, will be frozen at a performance level, which is fine in some areas. However, application developers will be forced to work harder in the future to exploit parallelism than they did in the past, and new parallel applications will continue to arise.

This leads to a host of new and continuing problems to be solved in order to allow parallelism to continue to provide performance benefits. Parallel algorithm research will continue to provide important basic parallelism. SW researchers will design new development tools to aid the above, work on languages that prevent the commission of serious bug errors, and drive toward hardware support for easier ways to write correct programs or tools to aid in debugging. Architecture and HW research will provide faster systems in accordance with Eqs. 1 and 2. All of these efforts tend to increase the degree of parallelism while reducing parallel computation time; increasing the width and shrinking the length of Fig. 1.

## Bibliography

1. Kuck DJ (1978) The structure of computers and computations. Wiley, New York
2. Randell B (1982) The origins of digital computers. Springer, New York
3. Hennessy J, Patterson DA (1996) Computer architecture: a quantitative approach, 2nd edn. Morgan Kaufmann, San Francisco
4. Jia W, Zhou W (2005) Distributed network systems: from concepts to implementations. Springer, New York
5. Noyce RN (1977) Microelectronics. Sci Am 273(3):63–69
6. Kennedy K, Allen R (2002) Optimizing compilers for modern architectures. Morgan-Kaufmann, San Francisco
7. Reinders J (2007) Intel threading building blocks. O'Reilly Media, Sebastopol
8. Chapman B, Jost G, van der Pas R (2008) Using open MP: portable shared memory parallel programming. O'Reilly Media, Sebastopol

9. Gropp W, Lusk E, Skjellum A (1999) Using MPI: portable parallel programming with the message-passing interface, 2nd edn. MIT, Cambridge
10. Kirk D, Hwu W-m (2010) Programming massively parallel processors. Morgan Kaufmann, San Francisco
11. Akl SG (1989) The design and analysis of parallel algorithms. Prentice Hall, Englewood Cliffs
12. Gramma A (2003) Introduction to parallel computing. Addison Wesley, Reading
13. Trobec R, Vajtersic M, Zinterhof P (eds) (2009) Parallel computing: numerics, applications, and trends. Springer, New York
14. Kuck DJ (1996) High performance computing. Oxford University Press, New York
15. Culler DE, Singh JP, Gupta A (1999) Parallel computer architecture. Morgan Kaufmann, San Francisco

---

## Parallel I/O Library (PIO)

- [Community Climate System Model](#)

---

## Parallel Ocean Program (POP)

- [Community Climate System Model](#)

---

## Parallel Operating System

- [Operating System Strategies](#)

---

## Parallel Prefix Algorithms

- [Reduce and Scan](#)
- [Scan for Distributed Memory, Message-Passing Systems](#)

---

## Parallel Prefix Sums

- [Reduce and Scan](#)
- [Scan for Distributed Memory, Message-Passing Systems](#)

## Parallel Random Access Machines (PRAM)

► PRAM (Parallel Random Access Machines)

## Parallel Skeletons

SERGEI GORLATCH<sup>1</sup>, MURRAY COLE<sup>2</sup>

<sup>1</sup>Westfälische Wilhelms-Universität Münster, Münster, Germany

<sup>2</sup>University of Edinburgh, Edinburgh, UK

### Synonyms

Algorithmic skeletons

### Definition

A parallel skeleton is a programming construct (or a function in a library), which abstracts a pattern of parallel computation and interaction. To use a skeleton, the programmer must provide the code and type definitions for various application-specific operations, usually expressed sequentially. The skeleton implementation takes responsibility for composing these operations with control and interaction code in order to effect the specified computation, in parallel, as efficiently as possible. Abstracting from parallelism in this way can greatly simplify and systematize the development of parallel programs and assist in their cost-modeling, transformation, and optimization. Because of their high level of abstraction, skeletons have a natural affinity with the concepts of higher-order function from functional programming and of templates and generics from object-oriented programming, and many concrete skeleton systems exploit these mechanisms.

### Discussion

Traditional tools for parallel programming have adopted an approach in which the programmer is required to *micromanage* the interactions between concurrent activities, using mechanisms appropriate to the underlying model. For example, the core of MPI is based around point-to-point exchange of data between processes, with a variety of synchronization modes. Similarly, threading libraries are based around primitives for locking, condition synchronization, atomicity,

and so on. This approach is highly flexible, allowing the most intricate of interactions to be expressed. However, in reality many parallel algorithms and applications follow well-understood patterns, either monolithically or in composition. For example, image processing algorithms can often be described as *pipelines*. Similarly, *parameter sweep* applications present a parallel scheduling challenge which is orthogonal to the detail of the application and parameter space.

### Programming with Skeletons

The term *algorithmic skeleton* stems from the observation that many parallel applications share common internal interaction patterns. The parallel skeleton approach proposes that such patterns be abstracted as programming language constructs or library operations, in which the implementation is responsible for implicitly providing the control “skeleton,” leaving the programmer to describe the application-specific operations that specialize its behavior to solve a particular problem. For example, a pipeline skeleton would require the programmer to describe the computation performed by each stage, while the skeleton would be responsible for scheduling stages to processors, communication, stage replication, and so on. Similarly, in a parameter sweep skeleton, the programmer would be required to provide code for the individual experiment and the required parameter range. The skeleton would decide upon (and perhaps dynamically adjust) the number of workers to use, the granularity of distributed work packages, communication mechanisms, and fault-tolerance. At a more abstract level, a divide-and-conquer skeleton would require the programmer to specify the operations used to divide a problem, to combine subsolutions, to decide whether a problem is (appropriately) divisible, and to solve indivisible problems directly. The skeleton would take on all other responsibilities, from whether to use parallelism at all, to details of dynamic scheduling, granularity, and interaction.

Thus, in contrast to the micromanagement of traditional approaches, skeletons offer the possibility of *macromanagement* – by selection of skeletons, the programmer conveys macro-properties of the intended computation. This is clearly attractive, provided that the skeletons offered are sufficiently comprehensive collectively, while being sufficiently optimizable individually and in composition. Sometimes a skeleton may

have several implementations, each geared to a particular parallel architecture, for example, distributed- or shared-memory, multithreaded, etc. Such customization has the potential for achieving high performance, portable across various target machines.

Application programmers gain from abstraction, which hides much of the complexity of managing massive parallelism. They are provided with a set of basic abstract skeletons, whose parallel implementations have a well-understood behavior and predictable efficiency. To express an application in terms of skeletons is usually simpler than developing a low-level parallel program for it.

This high-level approach changes the program design process in several ways. First, it liberates the user from the practically unmanageable task of making the right design decisions based on numerous, mutually influencing low-level details of a particular application and a particular machine. Second, by providing standard implementations, it increases confidence in the correctness of the target programs, for which traditional debugging is too hard to be practical on massively parallel machines. Third, it offers predictability instead of an *a posteriori* approach to performance evaluation, in which a laboriously developed parallel program may have to be abandoned because of inadequate efficiency. Fourth, it provides semantically sound methods for program composition and refinement, which open up new perspectives in software engineering (in particular, for reusability). And last but not least, abstraction, that is, going from the specific to the general, gives new insights into the basic principles of parallel programming.

An important feature of the skeleton-based methodology is that the underlying formal framework remains largely invisible to application programmers. The programmers are given a set of methods for instantiating, composing, and implementing diverse skeletons, but the development of these methods is delegated to the community of implementers.

In order to understand the spectrum of skeleton programming research, it is helpful to distinguish between skeletons that are predominantly data-parallel in nature (with an emphasis on transformational approaches), skeletons that are predominantly task-parallel or related to algorithmic classes, and the concrete skeleton-based systems that have embedded these concepts in real frameworks.

## Data-Parallel Skeletons and Transformational Programming

The formal background for data-parallel skeletons can be built in the functional setting of the Bird–Meertens formalism (BMF), in which skeletons are viewed as higher-order functions (functionals) on regular bulk data structures such as lists, arrays, and trees.

The simplest – and at the same time the “most parallel” – functional is *map*, which applies a unary function  $f$  to each element of a list, that is,

$$\text{map } f [x_1, x_2, \dots, x_n] = [f x_1, f x_2, \dots, f x_n] \quad (1)$$

*Map* has the following natural data-parallel interpretation: each processor of a parallel machine computes function  $f$  on the piece of data residing in that processor, in parallel with the computations performed in all other processors.

There are also the functionals *red* (reduction) and *scan* (parallel prefix), each with an associative operator  $\oplus$  as parameter:

$$\text{red}(\oplus)[x_1, x_2, \dots, x_n] = x_1 \oplus x_2 \oplus \dots \oplus x_n \quad (2)$$

$$\text{scan}(\oplus)[x_1, x_2, \dots, x_n] = [x_1, x_1 \oplus x_2, \dots, x_1 \oplus \dots \oplus x_n] \quad (3)$$

Reduction can be computed in parallel in a tree-like manner with logarithmic time complexity, owing to the associativity of the base operation. There are also parallel algorithms for computing the scan functional with logarithmic time complexity, despite an apparently sequential data dependence between the elements of the resulting list.

Individual functions are composed in BMF by means of backward functional composition  $\circ$ , such that  $(f \circ g)x = f(gx)$ , which represents the sequential execution order on (parallel) stages.

Special functions, called *homomorphisms*, possess the common property of being well-parallelizable in a data-parallel manner.

**Definition 1 (List Homomorphism)** A function  $h$  on lists is called a homomorphism with combine operation  $\otimes$ , iff for arbitrary lists  $x, y$ :

$$h(x + y) = (hx) \otimes (hy) \quad (4)$$

Definition 1 describes a class of functions, operation  $\otimes$  being a parameter, which is why it can be viewed as

defining a skeleton. Both map and reduction can obviously be obtained by an appropriate instantiation of this skeleton.

The key property of homomorphisms is given by the following theorem:

**Theorem 1 (Factorization)** *A function  $h$  on lists is a homomorphism with combine operation  $\otimes$ , iff it can be factorised as follows:*

$$h = \text{red}(\otimes) \circ \text{map } \phi \quad (5)$$

where  $\phi a = h[a]$ .

In this theorem, homomorphism has one more parameter beside  $\otimes$ , namely function  $\phi$ . The practical importance of the theorem lies in the fact that the right-hand side of the equation (5) is a good candidate for parallel implementation. This term consists of two stages. In the first stage, function  $\phi$  is applied in parallel on each processor (*map* functional). The second stage constructs the end result from the partial results in the processors by applying the *red* functional. Therefore, if a given problem can be expressed as a homomorphism instance then this problem can be solved in a standard manner as two consecutive parallel stages – map and reduction.

The standard two-stage implementation (5) may be time-optimal, but only under an assumption that makes it impractical: the required number of processors must grow linearly with the size of the data. A more practical approach is to consider a bounded number  $p$  of processors, with a data block assigned to each of them. Let  $[\alpha]_p$  denote the type of lists of length  $p$ , and subscript functions defined on such lists with  $p$ , for example,  $\text{map}_p$ . The partitioning of an arbitrary list into  $p$  sublists, called *blocks*, is done by the *distribution function*,  $\text{dist}(p) : [\alpha] \rightarrow [[\alpha]]_p$ . The following obvious equality relates distribution to its inverse, flattening:  $\text{red}(+) \circ \text{dist}(p) = \text{id}$ .

**Theorem 2 (Promotion)** *If  $h$  is a homomorphism w.r.t.  $\otimes$ , then*

$$h \otimes \text{red}(+) = \text{red}(\otimes) \circ \text{map } h \quad (6)$$

This general result about homomorphisms is useful for parallelisation via data partitioning: from (6), a

standard distributed implementation of a homomorphism  $h$  on  $p$  processors follows:

$$h = \text{red}(\otimes) \circ \text{map}_p h \circ \text{dist}(p) \quad (7)$$

Sometimes, it can be assumed that data is distributed in advance: either the distribution is taken care of by the operating system, or the distributed data are produced and consumed by other stages of a larger application.

The development of programs using skeletons differs fundamentally from the traditional process of parallel programming. Skeletons are amenable to formal transformation, that is, the rewriting of programs in the course of development, while ensuring preservation of the program's semantics. The transformational design process starts by formulating an initial version of the program in terms of the available set of skeletons. This initial version is usually relatively simple and its correctness is obvious, but its performance may be far from optimal. Program transformation rules are then applied to improve performance or other desirable properties of the program. The rules applied are semantics-preserving, guaranteeing the correctness of the improved program with respect to the initial version. Once rules for skeletons have been established (and proved by, say, induction), they can be used in different contexts of the skeletons' use without having to be reproved.

For example, in the SAT programming methodology [10] based on skeletons and collective operations of MPI, a program is a sequence of stages that are either a computation or a collective communication. The developer can estimate the impact of every single transformation on the target program's performance. The approach is based on reasoning about how individual stages can be composed into a complete program, with the ultimate goal of systematically finding the best composition. The following example of a transformation rule from [8] is expressed in a simplified C+MPI notation. It states that, if binary operators  $\otimes$  and  $\oplus$  are associative and  $\otimes$  distributes over  $\oplus$ , then the following transformation of a composition of the collective operations scan and reduction is applicable:

$$\left[ \begin{array}{l} \text{MPI\_Scan } (\otimes) ; \\ \text{MPI\_Reduce } (\oplus) ; \end{array} \right] \implies$$

```

 Make_pair;
 MPI_Reduce (f(\otimes , \oplus));
 if my_pid==ROOT then Take_first;

```

(8)

Here, the functions `Make_pair` and `Take_first` implement simple data arrangements that are executed locally in the processes, that is, without interprocess communication. The binary operator  $f(\otimes, \oplus)$  on the right-hand side is built using the operators from the left-hand side of the transformation.

Rule (8) and other, similar transformation rules have the following important properties: (1) they are formulated and proved formally as mathematical theorems; (2) they are parameterized in one or more operators, for example,  $\oplus$  and  $\otimes$ , and are therefore usable for a wide variety of applications; (3) they are valid for all possible implementations of the collective operations involved; (4) they can be applied independently of the parallel target architecture.

### Task- and Algorithm-Oriented Skeletons

In contrast to the data-parallel style discussed in the preceding section, there are many instances of skeletons in which the abstraction is best understood by reference to an encapsulated parallel control structure and/or algorithmic paradigm. These can be examined along a number of dimensions, including the linguistic framework within which the skeletons are embedded, the degree of flexibility and control provided through the API, the complexity of the underlying implementation framework and the range of intended target architectures.

While *map* is the simplest data-parallel skeleton, *farm* can be viewed as the simplest task-parallel skeleton. Indeed, in its most straightforward form, *farm* is effectively equivalent to *map*, calling for some operation to be applied independently to each component of a bulk data structure. More subtly, the use of *farm* often carries the implication that the execution cost of these applications is unpredictable and variable, and therefore that some form of dynamic scheduling will be appropriate – the programmer is providing a high-level, application-specific hint to assist the implementation. Typical *farm* implementations will employ centralized

or decentralized master–worker approaches, with internal optimizations which try to find an appropriate number of workers and an appropriate granularity of task distribution (trading interaction overhead against load balance). From the programmer’s perspective, all that must be provided is code for the operation to be applied to each task, and a source of tasks, which could be a data structure such as an array or a stream emerging from some other part of the program. The *bag-of-tasks* skeleton extends the simple *farm* with a facility for generating new tasks dynamically.

*Pipeline* skeletons capture the pattern in which a stream of data is processed by a sequence of “stages,” with performance derived from parallelism both between stages, and where applicable, within stages. Even such a simple structure allows considerable flexibility in both API design and internal optimization. For example, the simplest pipeline specification might dictate that each item in the stream is processed by each stage, that each such operation produces exactly one result, and that all stages are stateless. For such a pipeline, the programmer is required to provide a sequential function corresponding to each stage. More flexible APIs may admit the possibility of stateful stages, of stages in which the relationship between inputs and outputs is no longer one-for-one (e.g., filter stages, source, and sink stages), of bypassing stages under certain circumstances, and even of controlled sharing of state between stages. From the implementation perspective, internal decisions include the selection of the number of implementing processes or threads, allocation of operations to processes or threads, whether statically or dynamically, and correct choice of synchronization mechanism.

The *divide-and-conquer* paradigm underpins many algorithms: the initial problem is divided into a number of sub-instances of the same problem to be solved recursively, with subsolutions finally combined to “conquer” the original problem. For the situation in which the sub-instances may be solved independently the opportunities for parallelism are clear. Within this context, there is considerable scope to explore a skeleton design space in which constraints in the API are traded against performance optimizations within the implementation. A very generic API would simply require the programmer to provide operations for “divide” and “conquer,” together with a test determining whether an

instance should be solved recursively, and a direct solution method for those instances failing this test. Less-flexible APIs, could, for example, require the degree of division to be fixed (every call of divide returns the same number of sub problems), the depth of recursion to be uniform across all branches of divide tree, or even for some aspects of the “divide” or “conquer” operations to be structurally constrained. Each such constraint provides useful information, which can be exploited within the implementation.

Wavefront computations are to be found at the core of a diverse set of applications, ranging from numerical solvers in particle physics to dynamic programming approaches in bioinformatics. From the application perspective, a multidimensional table is computed from initial boundary information and a simple stencil that defines each entry as a function of neighboring entries. Constraints on the form of the stencil allow the table to be generated with a “wavefront” of computations, flowing from the known entries to the unknown, with consequent scope for parallelism. A *wavefront* skeleton may require the programmer to describe the stencil and the operation performed within it, leaving the implementation to determine and schedule an appropriate level of parallelism. As with *divide-and-conquer*, specific *wavefront* skeleton APIs may impose further constraints on the form of these components.

Branch-and-bound is an algorithmic technique for searching optimization spaces, with built-in pruning heuristics, and scope for parallelization of the search. Efficient parallelization is difficult, since although results are ultimately deterministic, the success of pruning is highly sensitive to the order in which points in the space are examined. This makes it both promising and challenging to the skeleton designer. With respect to the API, a *branch-and-bound* skeleton can be characterized by small number of parameters, capturing operations for generating new points in the search space, bounding the value of such a point, comparing bounds, and determining whether a point corresponds to a solution.

## Skeleton-Based Systems

Past and ongoing research projects have embedded selections of the above skeletons into a range of programming frameworks, targeting a range of platforms. The P3L language [15] was a notable early example in which skeletons became first-class language constructs,

together with sequential “skeletons” for iteration and composition. Subsequent projects within the Pisa group have taken similar skeletons into object-oriented contexts, with a focus on distributed and Grid implementations. In contrast, Muesli [3] allows data-parallel skeletons to be composed and called from within a layer which itself composes task-parallel skeletons, all within a C++ template-based library. Several systems have taken a functional approach. Hermann’s HDC [12] focuses exclusively on a collection of divide-and-conquer skeletons within Haskell, and implemented on top of MPI, while the Eden skeleton library [14] and related work [11] have implemented skeletons on top of both implicitly and explicitly parallel functional languages. The COPS project [2] presents a layered interface in which expert programmers can also have access to the implementation of the provided “templates,” all embedded in Java with both threaded and distributed RMI-based implementations. The SkeTo project offers a BMF-based collection of data-parallel skeletons for lists, matrices, and trees, implemented in C with MPI. Domain-specific skeletons are represented within the Mallba project [1] (focusing on combinatorial search) and Skipper and QUAFF projects (image processing). Higher-Order Components (HOCs) [5], Enhance [19], and Aspara [6] extend the idea of skeletons toward the area of distributed computing and Grids. HOCs implement generic parallel and distributed processing patterns, together with the required middleware support and are offered to the user via a high-level service interface. Users only have to provide the application-specific pieces of their programs as parameters, while low-level implementation details such as transfer of data across the grid are handled by the HOC implementations. HOCs have become an optional extension of the popular Globus middleware for Grids.

Skeleton principles are also evident in a number of other emerging parallel programming tools. Most notably, the MapReduce paradigm (related to, but distinct from the similarly named BMF skeletons) represents a pattern common to many applications in Google’s infrastructure. Emphasis in the original implementation was placed on load balancing and fault-tolerance within an unreliable massively parallel computational resource, a task strongly facilitated by the structurally constraining nature of the skeleton. Thain’s Cloud Computing Abstractions [18] implement

a range of distributed patterns with applications in Biometrics and Genomics. As with MapReduce, these are trivial to implement sequentially, and relatively straightforward on a reliable, homogeneous parallel architecture. The strength of the approach becomes apparent when ported to less predictable (or reliable) targets, with no additional effort on the part of the programmer. Finally, skeletal approaches can also be discerned in MPI's collective operations [9], where `MPI_Reduce` and `MPI_Scan` are parameterized by operations as well as data, and Intel's Threading Building Blocks [17], which in particular includes a *pipeline* skeleton.

## Related Entries

- [Collective Communication](#)
- [Eden](#)
- [Glasgow Parallel Haskell \(GpH\)](#)
- [NESL](#)
- [Reduce and Scan](#)
- [Scan for Distributed Memory, Message-Passing Systems](#)

## Bibliographic Notes and Further Reading

The term “algorithmic skeleton” was originally introduced by Cole [4]. A considerable body of work now exists. Helpful snapshots can be obtained by consulting the 2003 book edited by Rabhi and Gorlatch [16], the September 2006 special edition of the journal Parallel Computing [13], and the recent survey by Gonzalez-Velez and Leyton [7].

## Bibliography

1. Alba E, Almeida F, Blesa MJ, Cabeza J, Cotta C, Díaz M, Dorta I, Gabarró J, León C, Luna J, Moreno LM, Pablos C, Petit J, Rojas A, Xhafa F (2002) MALLBA: a library of skeletons for combinatorial optimisation. Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing, Springer, London, pp 927–932
2. Anvik J, Schaeffer J, Szafron D, Tan K (2003) Why not use a pattern-based parallel programming system? Euro-Par, Klagenfurt Austria, pp 81–86
3. Ciechanowicz P, Poldner M, Kuchen H (2009) The Münster skeleton library muesli – a comprehensive overview. Technical report, University of Münster
4. Cole M (1989) Algorithmic skeletons: structured management of parallel computation. MIT Press, Cambridge
5. Dünnebeier J, Gorlatch S (2009) Higher-order components for grid programming: making grids more usable. Springer, New York

6. Gonzalez-Velez H, Cole M (2010) Adaptive structured parallelism for distributed heterogeneous architectures: A methodological approach with pipelines and farms. Concurrency Comput Pract Exp 22(15):2073–2094
7. Gonzalez-Velez H, Leyton M (2010) A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Software Pract Exp 40:1135–1160
8. Gorlatch S (2000) Towards formally-based design of message passing programs. IEEE Trans Softw Eng 26(3):276–288
9. Gorlatch S (2004) Send-receive considered harmful: Myths and realities of message passing. ACM TOPLAS 26(1):47–56
10. Gorlatch S, Lengauer C (2000) Abstraction and performance in the design of parallel programs: an overview of the SAT approach. Acta Inf 36(9/10):761–803
11. Hammond K, Berthold J, Loogen R (2003) Automatic skeletons in template Haskell. Parallel Process Lett 13(3):413–424
12. Herrmann CA, Lengauer C (2000) HDC: a higher-order language for divide-and-conquer. Parallel Process Lett 10(2/3):239–250
13. Kuchen H, Cole M (eds) (2006) Algorithmic skeletons. Parallel Comput 32(7/8):604–615
14. Loogen R, Ortega Y, Pena R, Priebe S, Rubio F (2003) Parallelism abstractions in Eden. In: Rabhi F, Gorlatch S (eds) Patterns and skeletons for parallel and distributed computing. Springer, London, pp 95–128
15. Pelagatti S (1997) Structured development of parallel programs. Taylor & Francis, London
16. Rabhi FA, Gorlatch S (eds) Patterns and skeletons for parallel and distributed computing. Springer, London, ISBN 1-85233-506-8
17. Reinders J (2007) Intel threading building blocks. O'Reilly, Sebastopol CA
18. Thain D, Moretti C (2009) Abstractions for cloud computing with condor. In: Ahson S, Ilyas M (eds) Cloud computing and software services. CRC Press, Boca Raton
19. Yaikhom G, Cole M, Gilmore S, Hillston J (2007) A structural approach for modelling performance of systems using skeletons. Electr Notes Theor Comput Sci 190(3):167–183

## Parallel Tools Platform

GREGORY R. WATSON

IBM, Yorktown Heights, NY, USA

## Synonyms

[PTP](#)

## Definition

The Parallel Tools Platform (PTP) is an integrated development environment (IDE) for developing parallel programs using the C, C++, Fortran, and Unified Parallel C (UPC) languages. PTP builds on the Eclipse platform

by adding features such as advanced help, static analysis, parallel debugging, remote projects, and remote launching and monitoring. It also provides a framework for integrating other non-Eclipse tools into the Eclipse platform. The Parallel Tools Platform is not restricted to any particular programming model, but most tools to date are designed to support coarse-grained parallelism.

## Discussion

### Challenges

The challenges facing a developer writing parallel programs largely fall into the following three areas:

**Coding:** The programmer must translate the mathematical model of a problem into an algorithm that is implemented using a particular programming language and parallel programming model. This process of translation involves a large degree of effort, since there is often a significant mismatch between the model and an implementation that fully exploits the available parallelism.

**Testing and debugging:** Once an application program has been created, its correctness and accuracy must be validated. This involves a cycle of comparing the program output to known values using a variety of input data sets in order to verify that the results are correct. Coding errors, logic errors, and nondeterministic behavior all have to be addressed during this process, typically by employing an interactive debugger that allows program state to be inspected at key points during execution.

**Performance optimization:** A correctly functioning program may still not optimally utilize the available hardware resources. A variety of tools are typically employed to examine program behavior in order to identify bottlenecks and other performance related issues. These tools usually collect relevant information about the program execution, either directly, or in an aggregated form, and provide a means of analyzing and interpreting the data in order to identify actions that can be taken to improve performance. This is important for many situations when the speed of execution is time critical, such as in weather applications.

These activities are made significantly more difficult when applications are being developed for architectures that employ concurrent execution (or parallelism)

in order to achieve extremely high levels of performance. Parallel programming introduces many additional complexities that impact on all aspects of the development process. The nature of these complexities depends on the particular programming model employed, but includes issues such as deciding how the algorithm or data is to be partitioned, the need to deal with *communication* in addition to computation, and the inherent nondeterminism introduced by concurrent execution.

### Productivity

The quest to maximize productivity has received a much greater emphasis in recent years, where productivity is defined as both the speed and ease at which a correctly functioning application can be *developed* combined with the level of *performance* achieved by fully exploiting the available parallelism. Improving productivity is therefore dependent on a wide range of factors. One approach for improving the application development process, and that has seen wide success across a range of computing disciplines, is the use of an integrated development environment (IDE).

An IDE combines a core set of tools into a single package that provides a consistent user interface across a variety of systems and architectures. Many IDEs also provide a plug-in capability so that the core set of tools can be extended as new tools are developed or extra functionality is required. IDEs tend to improve developer productivity because the required tools are always available regardless of the environment, and also because they are integrated together, tools can share data and other information in order to streamline the developer workflow. Scientific computing is one of the few areas that have largely shunned the use of IDEs. Recently, however, the benefits of using an IDE for developing scientific programs have become more apparent to the community. This is probably because the scope of the programming task for the new generation of supercomputers is looking increasingly daunting, combined with an influx of new developers who have already had exposure to, and understand the advantages of, IDEs.

### Productive Parallel Programming

The goal of the Parallel Tools Platform is to address the challenges facing developers of parallel programs in a

manner that leads to improved productivity. The strategy employed by PTP to achieve this is to combine the productivity enhancements inherent in an integrated development environment with a number of key tools that specifically target the parallel application developers' capacity to deliver correct applications that are optimized for their target platforms.

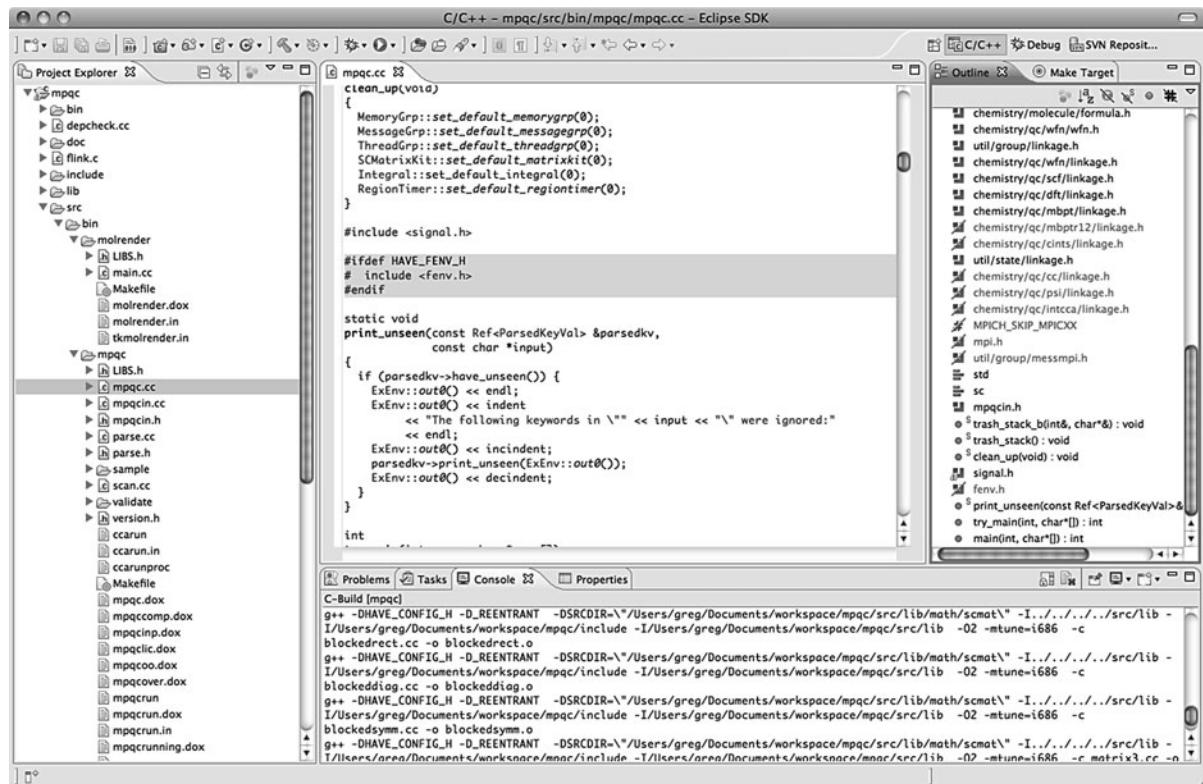
## C, C++, Fortran, and UPC Programming in Eclipse

The Eclipse platform is an open source, vendor neutral, portable, extensible platform for tool integration. It employs a plug-in architecture that allows tools to be integrated directly with the platform, and provides a range of core functionality that is suited to a variety of application development activities. This core functionality includes support for multi-developer projects, an integrated help system, multiple language support, project resource management, advanced editing features, incremental builds, and a range of other features. Eclipse functionality is extensible using the plug-in

mechanism, and there is a rich community of developers and a large number of both commercial and open source plug-ins available. The IDE is also highly portable, so it is available on a wide variety of hardware platforms.

The main interface provided by Eclipse is the *workbench* shown in Fig. 1. The workbench arranges *views* (text editors, project explorers, consoles, and other user interface components) into groupings known as *perspectives*. There is typically a perspective available for particular types of activities (coding, debugging, etc.) Eclipse also permits views to be shared between different perspectives. Along with a traditional menu bar, the workbench provides a toolbar containing a variety of actions that are associated with the currently active perspective. There is also a status bar and a range of other user interface decorations. Individual views may also have toolbars and menus that vary depending on context.

Eclipse provides a number of advanced coding and editing features that are common across any language



Parallel Tools Platform. Fig. 1 The eclipse workbench showing the C/C++ development perspective

that it supports. Many of these features rely on core functionality that is provided as standard in Eclipse, but is typically not available in other development environments. Such features include:

**Context-sensitive help:** In order to provide user assistance that is targeted to a particular context, this feature can deliver help documentation on demand (for example, when the user hits a key) or automatically display information such as prototype definitions or code snippets that pop up when the cursor is placed over a particular piece of text.

**Searching:** In addition to the usual string or pattern matching searches available in many editors, Eclipse allows searching on language elements (such as types, variables, functions etc.), the ability to limit searches to contexts (such as declarations, references, etc.), and to specify the scope of the search (such as a subset of the files in a project).

**Content assistance:** Reducing the amount of typing that a developer is required to do can be a powerful means of improving productivity. By inferring what the developer will type based on context and scope information, Eclipse's *code completion* makes suggestions using the first few letters of the language element currently being entered (e.g., a function name, variable name, etc.), via either manual or automatic activation. Frequently used snippets of code can also be saved as *code templates*. These templates can then be inserted into the text according to the current scope in the same manner as other code completions.

**Refactoring:** Improving code by changing its structure without changing behavior is a common, but error-prone, task in programming. By providing a range of common but sophisticated refactorings that can be automatically performed, Eclipse is able to prevent many of the errors that are introduced by manual changes. Typical refactorings include the ability to rename language elements, such as variables or functions, extract sections of code into a method or function, and automatically declare an interface, as well as many others.

**Wizards:** Another method of reducing the amount of repetitive work is to automate many of the common tasks that a developer must undertake. Eclipse provides a variety of wizards for activities such as creating new projects, classes, or files, importing projects into

Eclipse, exporting projects from Eclipse, and project conversion.

In addition to these powerful features, Eclipse provides support for application building, either using project supplied build scripts (for example Makefiles), or using an internal build system to track file dependencies and manage external tools such as compilers and linkers. The build support is also able to process errors and warnings generated by external tools, and map these back to messages displayed in the user interface, and markers that are inserted into the editor views.

The Eclipse C/C++ Development Tools (CDT) provide C, C++, and UPC language support, while the Parallel Tools Platform *Photran* feature adds Fortran support to Eclipse. Support for a wide range of other languages also available through a variety of Eclipse and third-party plug-ins.

## Code and Static Analysis for Parallel Programs

In addition to the advanced editing features that are a standard part of Eclipse, PTP adds a range of enhanced features that are specifically targeted toward parallel programming.

The first of these provides additional high-level help documentation for language elements used by common programming models, such as the message passing interface (MPI), the OpenMP Application Programming Interface, IBM's Low-level messaging Application Programming Interface (LAPI), and UPC. This documentation can be accessed using Eclipse's standard context sensitive help feature during coding, and will display prototype information along with a detailed description of the element.

Another feature provided by PTP is cataloging and navigation of MPI application programming interfaces and OpenMP pragma statements in a parallel program. This enables the developer to see the elements that have been used in the program at a single glance, and to navigate to the source code line that contains a selected element.

PTP also includes a Barrier Analysis tool that provides a more sophisticated analysis of MPI programs. This tool locates all calls to MPI *barrier* functions (synchronization points) in a program, and computes logical paths through the program to determine if it

is possible for a deadlock to occur. This can happen because all tasks must synchronize on an MPI barrier, but logic errors may enable one or more tasks to miss execution of the barrier. The advantage of this type of analysis is that it can be performed without the need to execute the program.

used to enter all the parameters necessary for a successful program launch. Run configurations are automatically saved so they can be easily used to relaunch an application during testing or debugging activities. PTP provides an extended run configuration for parallel programs that allows system-specific information about the parallel run to be supplied.

## Launching and Monitoring Parallel Programs

One hindrance to developer productivity is the seemingly basic ability to easily launch an application on a target parallel system. Most high-performance computing systems employ complex environments to monitor and control application launching. Many of these systems are also highly centralized assets that are tightly controlled, and they will typically restrict direct user access using a batch scheduler system. A developer's ability to interact with these systems will therefore be limited by their ability to deal with the additional complexity that this introduces. This is particularly the case where programs are being ported from one system to another, or where the developer has access to multiple systems and architectures.

PTP's approach to this issue is to provide a single, uniform interface that allows the developer to launch and monitor applications on a wide range of systems, without needing to understand the intricacies of each particular system. This is achieved through two features:

*Parallel runtime perspective:* This perspective provides a number of views that give the developer a snapshot of system activity at any particular time. Importantly, the views are independent of the type of system that is being targeted, so the developer does not need to be concerned with the underlying system details. The perspective is comprised of the *Resource Manager View*, which lists the systems that are available to the developer for launching and debugging applications, the *Machines View*, which shows the status of the system and its computing elements, and the *Jobs View*, which lists pending, running, and completed jobs on the system. A console is also available to display the program-generated output.  
*Parallel application run configuration:* This utilizes the standard Eclipse *Run Configuration* dialogs, which are

## Parallel Debugging

Developing complex parallel programs is a difficult task, and it is particularly challenging to ensure that they operate correctly. Finding errors in a parallel program is complicated because the many concurrent threads of execution make it difficult to observe the program operation in a deterministic manner. Managing large applications running on many processors may present a problem to the developer as well since the sheer volume of information may be overwhelming. The very act of debugging the program may also disturb its operation enough to make identifying the cause of an error impossible.

The PTP debugger attempts to address some of these issues. In particular, the debugger provides the basic functionality needed to step through the execution of the program examining variables and program state along the way, and an easy to use interface that enables the developer to quickly obtain pertinent information about an executing application. The debugger is invoked in the same manner as any other application launch, by clicking a button. Eclipse will automatically switch to the *Parallel Debug Perspective* when the debugging session is ready.

The Parallel Debugger Perspective comprises a number of views to facilitate the debugging task. The *Parallel Debug View* provides a high level view of the application, showing all the tasks currently executing. Debug operations on groups of processes, such as single stepping, can be performed using this view. The *Debug View* provides information about individual threads of execution, including a stack frame showing the current location. The *Variables View* shows the local variables from the currently selected stack frame in the Debug View, and can be used to inspect variables from a range of tasks. Other views are also available to inspect different aspects of the program state.

## Utilizing External Tools

Although PTP provides a range of tools for developing parallel programs, there are many other tools available that could be used to the developer's advantage if they were accessible from within the Eclipse environment. To facilitate this, PTP provides an external tools framework that allows non-Eclipse tools to be integrated in a manner that preserves the developer's workflow. The framework defines three main integration points during the application development workflow: *compile*, *execute*, and *analyze*. The *compile* integration point specifies how the normal build commands are to be modified during the build process to perform any tool-specific actions. An example of this might be to instrument the application to collect tracing or profiling information. The *execute* integration point specifies how the command used for launching of the application executable can be modified to perform any tool-specific actions. An example of this might be to pass the application executable to a tool that performs data collection during the execution. The *analyze* integration point allows an external tool to be launched once the program execution is complete, such as a tool to analyze performance data generated during execution.

In addition to these integration points, the external tools framework provides a *Feedback View* that enables externally generated information to be mapped back to source files and lines within the Eclipse environment. Eclipse can then display this information in the form of sortable tables, or by annotating the source code directly with markers or other forms of highlighting.

## Remote Development

To be an effective enhancement to conventional development processes, PTP needs to support a broad range of environments and systems. Many of these systems are scarce resources, so access is often tightly controlled from remote locations. Ideally, application developers for these systems need to be able to access these resources for testing, debugging, and performance optimization as if they were local. This enables the development process to be significantly streamlined, since the developer does not need to be concerned with copying executables and/or data files from system to system.

PTP addresses this requirement by adding a *Remote Development Tools (RDT)* feature. This enables a project

to be physically located on a remote machine, but allows access to the project and its source files for editing and building as if they were local. RDT takes care of all the activities necessary to make this process appear as transparent as possible to the developer. This can also be combined with PTP's ability to launch and debug programs on a remote target system, so the developer is able to take advantage of a fully remote enabled environment.

## The Parallel Tools Platform Today

PTP is an evolving project. Started in 2005, it has an active and growing developer community that spans a range of government, academic, and commercial organizations. The number of developers contributing to the project continues to increase, including the renowned National Center for Supercomputing Applications, which has recently begun to actively participate in PTP development.

Another promising development is the contribution of value-added components that enhance the functionality of PTP. Contributions have been made by the University of Oregon to add functionality to support their performance analysis tool, called Tuning and Analysis Utilities (TAU), the University of Utah to support their tool for formal verification of MPI programs, called In-situ Partial Order (ISP), and the University of Florida to support their tool for performance analysis of partitioned global-address-space (PGAS) programming models, called Parallel Performance Wizard (PPW).

PTP's main goal to date has been to demonstrate that Eclipse is a viable alternative for the development of scientific applications for large-scale computer systems. The list of features that are now available, and the growing developer and contributor community, demonstrate that this goal has been met.

## Future Directions

A comprehensive IDE for developing applications for a broad range of parallel platforms is a huge undertaking. PTP has begun by addressing many of the most pressing issues, such as programmer assistance, uniform access to remote systems, parallel debugging, and external tool integration. However there are still a large number of

issues that remain to be dealt with, and these will form the basis of much of the ongoing development over the coming years. Some of these issues include:

**Scalability:** Both application and systems sizes are increasing dramatically. The next generation of petascale systems will have hundreds of thousands of processing elements, and executions that exceed one million threads will be likely. The applications themselves are growing continuously, with millions of lines of code now becoming increasingly common. To continue to be effective, PTP will need to support such scales without unduly impacting developer productivity.

**Debugging paradigms:** Conventional debugging approaches have been successfully applied to parallel programs for some years. However, as program sizes increase, it is likely that these techniques will break down. One reason for this is that the sheer volume of information (for example, a million executing threads) is likely to overwhelm the developer. New debugging paradigms will need to be introduced that address this information overload, yet still allows the developer to accurately pinpoint the location of errors within a program.

**Static analysis tools:** Eclipse has an enormously powerful infrastructure available to the tool developer, and PTP has only really touched the surface of possible tools to exploit this. There is significant scope for much more powerful static analysis and refactoring tools that could have a very beneficial impact on developer productivity, and work is actively underway to enhance this capability.

**Remote development:** To be a truly effective remote development platform, PTP needs to support additional paradigms for developing applications remotely. In particular, the ability to work off-line and support for synchronizing with a remote repository are two areas that would enhance developer productivity significantly.

**Multi-core architectures:** The computing industry is beginning to see a real convergence between multi-core systems, which have already reached hundreds or even thousands of cores, and parallel architectures, which already employ multi-core programming elements. Many of the techniques used for parallel programming may also be beneficial for applications that wish to exploit multi-core technology. PTP is the logical home for exploring this convergence, and work is beginning to take place in this area.

## Related Entries

- ▶ [Compilers](#)
- ▶ [Debugging](#)
- ▶ [Fortran 90 and its Successors](#)
- ▶ [MPI \(Message Passing Interface\)](#)
- ▶ [OpenMP](#)
- ▶ [PGAS \(Partitioned Global Address Space\) Languages](#)
- ▶ [Performance Analysis Tools](#)
- ▶ [Scheduling](#)
- ▶ [TAU](#)
- ▶ [UPC](#)

## Bibliographic Notes and Further Reading

The idea of using computers to aid in the software engineering process extends back to the late 1960s; however, it was not until the 1980s that the market for computer-aided software engineering or CASE-based tools became significant (the term CASE was not coined until 1982). CASE covers a very broad area of software engineering practices, whereas integrated development environments, which are one class of CASE tools, tend to focus on the relatively small set of software engineering practices associated with the edit-build-test-debug development lifecycle.

There have been a number of papers describing the relationship between productivity and the use of IDEs, and the productivity and quality improvements that can be obtained by such environments is well documented [5, 6, 9, 10]. Their success is also reflected in the best practice use of IDEs for most commercial software development today.

Van De Vanter, et al. [11] have asserted that there are many reasons why existing developer tools are not going meet the productivity requirements of high performance computing: they are difficult to learn, they may not scale as machine sizes increase, they are different across different platforms, they are hard to develop or port to new platforms, they are often poorly supported, and they can be expensive.

The use of IDEs for parallel computing has also been explored before [1–4, 8], but all of these appear to have suffered from low levels of use, were academic research projects, or were too expensive to develop and maintain, and so have gradually died out. PTP is the first time that an open-source IDE has been tailored specifically for scientific and parallel computing [12].

The non-for-profit Eclipse Foundation was created in 2004 to oversee the stewardship of the open-source Eclipse platform. Since it was established, the Foundation membership has grown to exceed 150 organizations. Many of these companies use Eclipse as the core of a commercial product offering, after having abandoned their own proprietary IDE technology and switched to the Eclipse platform. Eclipse is widely used across many computing fields, and arguably has the largest number of users of any IDE in history. It is also now one of the largest open-source projects and there have been over two million downloads of the latest version alone [7].

## Bibliography

1. Bemmerl T (1992) TOPSYS for programming distributed multiprocessor computing environments. In: Proceedings of computer systems and software engineering, The Hague, pp 175–180
2. Brode BQ, Warber CR (1998) DEEP: a development environment for parallel programs. In: Proceedings of the international parallel processing symposium, Orlando, pp 588–593
3. Callahan D, Cooper K, Hood R, Kennedy K, Torczon L (1987) ParaScope: a parallel programming environment. In: Proceedings of the first international conference on supercomputing, Athens, Greece, June 1987
4. Cownie J, Dunlop A, Hellberg S, Hey AJG, Pritchard D (1994) Portable parallel programming environments – the ESPRIT PPPE project, massively parallel processing applications and development, The Netherlands, June 1994
5. Frazer A (1993) CASE and its contribution to quality. The Institution of Electrical Engineers, London
6. Granger MJ, Pick RA (1991) Computer-aided software engineering's impact on the software development process: an experiment. In: Proceedings of the 24th Hawaii international conference on system sciences, Jan 1991, pp 28–35
7. <http://eclipse.org/downloads>
8. Kacsuk P, Cunha JC, Dózsa G, Lourenço J et al (1997) A graphical development and debugging environment for parallel programs. Parallel Comput 22(13):1747–1770
9. Luckey PH, Pittman RM (1991) Improving software quality utilizing an integrated CASE environment. In: Proceedings of the IEEE national aerospace and electronics conference, Dayton, May 1991, pp 665–671
10. Norman RJ, Nunamaker JF Jr (1989) Integrated development environments: technological and behavioral productivity perceptions. In: Proceedings of the annual Hawaii international conference on system sciences, Kailua-Kona, Jan 1989, pp 996–1003
11. Van De Vanter ML, Post DE, Zosel ME (2005) HPC needs a tool strategy. In: Proceedings of the second international workshop

on software engineering for high performance computing system applications, ACM, May 2005, pp 55–59

12. Watson GR, DeBardeleben NA (2006) Developing scientific applications using eclipse. Comput Sci Eng 9(4):50–61

## Parallelism Detection in Nested Loops, Optimal

ALAIN DARTE

École Normale Supérieure de Lyon, Lyon, France

### Synonyms

Detection of DOALL loops; Nested loops scheduling; Parallelization

### Definition

Loops are a fundamental control structure in imperative programming languages. Being able to analyze, transform, and optimize loops is a key feature for compilers to handle repetitive schemes with a complexity proportional to the program size and not to the number of operations it describes. This is true for the generation of optimized software as well as for the generation of hardware, for both sequential and parallel execution.

**Exploiting parallelism** is a difficult task that depends, among others, on the target architecture, on the structure of computations in the program, and on the way data are mapped, used, and communicated. In contrast, **detecting parallelism** that can be expressed as loops, i.e., transforming Fortran-like DO loops into DOALL loops (loops whose iterations are all independent) depends only on the dependences between computations in the program being analyzed. Intuitively, an algorithm is **optimal** for parallelism detection in loops, if it transforms the loops of a program so that each statement is surrounded, after transformation, by a maximal number of parallel (DOALL) loops. However, such a notion of optimality needs to be defined with care (see the discussion hereafter) to avoid inconsistent or inaccurate claims.

### Discussion

#### Optimal Parallelism Detection in Loops

How to define optimal parallelism detection in loops? An optimality criterion solely based on the *number of*

parallel loops – thus not on the number of iterations – has no meaning for loops with constant bounds as they can always be unrolled or strip-mined. For example, it is true that hyperplane scheduling can always be applied on  $n$  perfectly nested loops with constant bounds, resulting in a code with one outer sequential loop and  $(n - 1)$  parallel loops. However, it does not mean that it exhibits any parallelism as, for a purely sequential code, each parallel loop has then a single iteration. Similarly, an optimality criterion should not lead to inconsistencies due to the way loops are generated, e.g.,  $n$  nested loops with constant loop bounds can be fully unrolled leading to a code with no loop while applying loop tiling leads to  $2n$  loops.

One way to define **optimality** is **with respect to the execution time** of the parallelized code **on an ideal PRAM-like machine**, with an unlimited number of computation units, where any instruction takes a single unit of time. An algorithm for parallelism detection is then optimal if the corresponding code with DOALL loops has an optimal execution time for this ideal machine, which is the maximal length of a path in the dependence DAG (directed acyclic graph) obtained by fully unrolling the program. However, a full unroll of the code is too costly, often undesirable as it loses the loop structure, and sometimes impossible, e.g., when the loop bounds are parameterized. A more practical definition of optimality is to assume that each loop has a parameterized number of iterations, of order  $N$ , and to say that an algorithm is optimal if the execution time of the parallelized program on the ideal machine is  $O(N^d)$  where  $d$  is minimal. Optimality can then be proved by exhibiting a dependence path in the unroll program whose length is not  $O(N^{d-1})$ . A more accurate definition of optimality can be given by applying the same reasoning for each statement, considering that any other statement takes no time on the ideal machine, i.e., does not count in the length of a dependence path.

In general, algorithms for parallelism detection transform the code so that each statement is surrounded by the same number of loops before and after transformation. This is true, in particular, for algorithms based on unimodular transformations. In this case, one retrieves the intuitive notion of optimality which states that parallelism detection is optimal if each statement is surrounded, after transformation, by a maximal number of parallel loops. The only constraint that a

parallelism detection algorithm must respect is that the partial order of operations defined by the dependences in the program are preserved. How these dependences are computed and abstracted is not part of the algorithm, it is its input. In other words, the **optimality** of an algorithm can only be defined **with respect to the dependence abstraction** it uses. To show that it is optimal, one needs to exhibit a dependence path of the required length, not in the original program, but in its abstraction, i.e., a dependence path in an over-approximation of the actual dependence graph.

A more complete discussion on the definition of optimality for parallelism detection algorithms is provided in [10, 12, 13]. The rest of this essay, with large parts borrowed from [7], shows that it is possible to give, for each classic dependence abstraction (dependence level, uniform dependence vector, direction vector, dependence cone, dependence polyhedron), an algorithm which is optimal with respect to this abstraction. Moreover, this algorithm is a specialization of a more generic algorithm [13], inspired by the decomposition of Karp, Miller, and Winograd for checking the computability of a system of uniform recurrence equations [18]. Notice that the previous optimality criterion makes no distinction between an outer parallel loop and an inner parallel loop. If a parallel loop can always be pushed down, i.e., interchanged with inner loops, the converse is not true. However, detecting parallel loops that contain sequential loops and detecting permutable loops (the base for loop tiling) are two similar problems that can be achieved by variations of the algorithms mentioned hereafter. But optimality for these goals is more difficult to define. Here is a summary of the main optimality results:

*Dependence level* The algorithm of Kennedy and Allen [2] is optimal, which implies that loop distribution is sufficient to detect maximal parallelism for dependence levels.

*Uniform dependences* Hyperplane scheduling (as defined by Lamport [19]) is optimal, for perfectly nested loops, which implies that unimodular transformations with one outermost sequential loop is sufficient for uniform dependences.

*Direction vectors* An adaptation of the algorithm of Wolf and Lam [25] (which detects permutable loops) is optimal, which implies that unimodular

transformations (loop interchange, loop reversal, loop skewing) are sufficient for direction vectors.

*Dependence polyhedron* The algorithm of Darte and Vivien [13] is optimal, which implies that unimodular transformations, combined with loop distribution and loop shifting, are sufficient for dependence polyhedra.

*Affine dependences* The algorithm of Feautrier [16] is optimal among all affine transformations, but these transformations are not sufficient to detect maximal parallelism for affine dependences.

This essay is organized as follows. The first section recalls the model of system of uniform recurrence equations (SURE) and the link between the computability and scheduling problems of such a system. The second section shows how, thanks to a uniformization of dependence distance abstractions, the detection of parallelism in nested DO loops can be transferred to the model of SURE so as to derive optimality results. The last section illustrates yet another connection: the multi-dimensional scheduling techniques used to analyze a SURE and to detect parallelism in nested DO loops can also be used to prove the termination of some WHILE loops.

## The Organization of Computations in a System of Uniform Recurrence Equations

In 1967, Karp, Miller, and Winograd introduced a model (system of uniform recurrence equations or SURE) to describe a set of regular computations as mathematical equations [18]. The goal of their paper, entitled “The Organization of Computations for Uniform Recurrence Equations,” was to study the parallelism that such a description contains implicitly, motivated by “the recent [at this time] development of computers capable of performing many operations concurrently,” in particular for regular applications such as solving partial differential equations by finite-difference methods. This work was purely theoretical: the paper does not even have a conclusion section, which would possibly mention some applications. Nevertheless, it turned out to be very prophetic, when considering the large number of developments for which it served as foundations. For example, it has some connections with accessibility problems in vector addition systems and Petri nets. The developments of systolic arrays and of high-level synthesis from

systems of recurrence equations are directly inspired from it. Most scheduling methods developed for automatically parallelizing DO loops are based on it, directly or indirectly. As a by-product, the theory of unimodular transformations and the different parallelism detection algorithms [2, 13, 16, 19, 25] can be seen as extensions and/or specializations of the technique of Karp, Miller, and Winograd, adapted to specific dependence abstractions and objectives. Last but not least, it has also some connections with techniques to prove the termination of imperative programs, either with the insertion of counters or by the derivation of affine ranking functions [1, 5, 17, 22].

### Definition of a SURE

A SURE is a finite set of  $m$  equations of the form

$$a_i(p) = f_i(a_{i_1}(p - d_{i_1,i}), \dots, a_{i_m}(p - d_{i_m,i})) \quad (1)$$

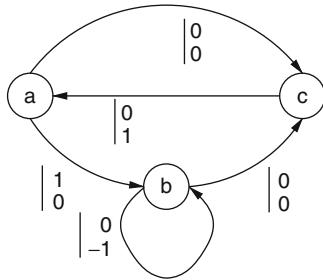
where  $p \in \mathbb{Z}^n$  is called an **iteration vector**,  $d_{i,j} \in \mathbb{Z}^n$  is called a **dependence vector**,  $f_i$  is a strict function with  $m_i$  arguments whose properties are not considered (only the structure of computations is analyzed, not what they do). The value  $a_i(p)$  has to be computed for all integral points  $p$  in a subset  $\mathcal{P}$  of  $\mathbb{Z}^n$ , called the **evaluation region**. Values are supposed to be given at  $p - d_{i,j} \notin \mathcal{P}$  (input variables) wherever required for the evaluation at  $p \in \mathcal{P}$ .

The value  $a_i(p)$  is said to depend on  $a_j(p - d_{j,i})$ , for  $1 \leq j \leq i_m$ . These dependence relations define a graph  $\Gamma$ , called the **expanded dependence graph** (EDG). Its set of vertices is  $\{1, \dots, m\} \times \mathcal{P}$  and there is an edge from  $(j, q)$  to  $(i, p)$  if  $a_i(p)$  depends on  $a_j(q)$ . A SURE can be equivalently defined by a weighted-directed multigraph  $G = (V, E, w)$ , called the **reduced dependence graph** (RDG), as follows:

- For each  $a_i$ , there is a vertex  $v_i$  in  $V$ .
- If  $a_i(p)$  depends on  $a_j(p - d_{j,i})$ ,  $G$  has an edge  $e = (v_j, v_i)$  in  $E$  from  $v_j$  to  $v_i$  with **weight**  $w(e) = d_{j,i}$ .

A SURE is then defined by an RDG  $G = (V, E, w)$  and an evaluation region  $\mathcal{P}$ . For example, the following system, to be evaluated for  $\mathcal{P} = \{(i, j) \mid 1 \leq i, j \leq N\}$ ,

$$\begin{cases} a(i, j) = c(i, j - 1) \\ b(i, j) = a(i - 1, j) + b(i, j + 1) \\ c(i, j) = a(i, j) + b(i, j) \end{cases}$$



**Parallelism Detection in Nested Loops, Optimal.** Fig. 1

Corresponding RDG for the SURE example

defines a SURE with three equations on a square of size  $N$ , corresponding to the RDG of Fig. 1.

### Computability: Definition and Properties

The definition of a SURE, either by Equation (1), or by the reduced dependence graph, is implicit: to compute a value  $a_i(p)$ , one must first compute all values that appear in the right-hand side of the definition of  $a_i(p)$ . These values are computed the same way, by evaluating the right-hand side of their definition, and so on. Thus, two questions arise:

*Computability problem* Does this recursive process end, i.e., are all values  $a_i(p)$  computable?

*Scheduling problem* If the system is computable, how to organize the computations?

A **schedule** is a function  $\theta$  from the vertices of the EDG  $\Gamma$  to  $\mathbb{N}$  such that  $\theta(j, q) < \theta(i, p)$  whenever  $(i, q)$  depends on  $(j, q)$ . A SURE is **computable** (or explicitly defined) if there exists a schedule. It is not computable if there exists a vertex  $(i, p)$  in the EDG  $\Gamma$  such that the length of a dependence path leading to it is unbounded. Then, either  $a_i(p)$  depends on itself (possibly through other computations) or it needs to wait “infinite” time before the complete evaluation of its right-hand side. In this case, because the in-degree in the EDG  $\Gamma$  is finite, there is an infinite path leading to  $(i, p)$ .

To make the analysis simpler, the following discussion focuses on (parametric) bounded evaluation regions. An RDG  $G$  is said computable if all SUREs defined from  $G$  on bounded evaluation regions are computable.

**Theorem 1** An RDG is computable if and only if (iff) it has no cycle of zero weight.

More precisely, the fact that  $G$  has no cycle of zero weight is a sufficient condition for a SURE defined from  $G$  to be computable, whatever the bounded evaluation region. However, it is a necessary condition only if the evaluation region is “sufficiently large” (see [9, 18]).

### The case of a single equation

A uniform recurrence equation (URE) is defined by an evaluation region  $\mathcal{P} \subseteq \mathbb{Z}^n$  and an RDG  $G$  with a single vertex, i.e., by a single equation  $a(p) = f(a(p - d_1), \dots, a(p - d_m))$ . Let  $D$  be the  $n \times m$  matrix whose columns are the dependence vectors. Consider the following sets:

$$T(D) = \{t \in \mathbb{Z}^m \mid tD \geq 1\}$$

$$Q(D) = \{q \in \mathbb{Z}^m \mid q \geq 0, q \neq 0, Dq = 0\}$$

and  $T_{\mathbb{Q}}(D)$  and  $Q_{\mathbb{Q}}(D)$  their rational relaxations.  $G$  is computable iff it has no cycle of zero weight, i.e.,  $Q(D)$  is empty. Farkas Lemma [23] shows the following result:

**Theorem 2**  $Q(D) = \emptyset \Leftrightarrow Q_{\mathbb{Q}}(D) = \emptyset \Leftrightarrow T_{\mathbb{Q}}(D) \neq \emptyset \Leftrightarrow T(D) \neq \emptyset$ .

This property shows that checking if an RDG is computable can be done in polynomial time by checking that  $T_{\mathbb{Q}}(D)$  is non empty, i.e., that the cone generated by the dependence vectors is strictly included in a half-space. Each  $t \in T(D)$  corresponds to a **separating hyperplane** and a schedule  $\theta_t$  defined by  $\theta_t(p) = t.p + K$  where  $t.p$  is the vector product and  $K = -\min_{p \in \mathcal{P}} t.p$ . Indeed, if  $q$  depends on  $p$ ,  $q = p + d_i$ , then  $\theta_t(q) = t.q + K = t.p + t.d_i + K > \theta_t(p)$ . Thus, an RDG is computable iff there is an **affine schedule**, i.e., a way of computing the URE by a regular schedule, whose definition does not depend on the evaluation region  $\mathcal{P}$ .

Given a vector  $t \in T_{\mathbb{Q}}(D)$ , the **latency** of the schedule  $\theta_t$ , i.e., the total number of sequential steps it induces, can be rounded to  $L(\theta_t) = \max_{p, q \in \mathcal{P}} t.(p - q)$ . Finding the “fastest” linear schedule means solving a min-max optimization problem  $L_{\min} = \min\{L(\theta_t) \mid t \in T_{\mathbb{Q}}(D)\}$ . If  $\mathcal{P}$  is a polytope  $\{p \mid Ap \leq b\}$ , then  $L(\theta_t) = \max\{t.(p - q) \mid Ap \leq b, Aq \leq b\}$ . The duality theorem of linear programming [23] leads to:

$$\begin{aligned} L(\theta_t) &= \max\{t.(p - q) \mid Ap \leq b, Aq \leq b\} \\ &= \min\{(t_1 + t_2).b \mid t_1, t_2 \geq 0, t_1 A = t, t_2 A = -t\} \\ L_{\min} &= \min\{(t_1 + t_2).b \mid t_1, t_2 \geq 0, t_1 A = -t_2 A = t, tD \geq 1\} \end{aligned}$$

Solving the previous linear program gives a way to produce a vector  $t$  in  $T_{\mathbb{Q}}(D)$  from which a fast schedule can be built. Its performance can be characterized as follows:

**Theorem 3** *If the evaluation region  $\mathcal{P}$  is sufficiently large, the difference between  $L_{\min}$  (the latency of the fastest affine schedule) and the longest dependence path in  $\Gamma$  (the latency of the fastest schedule) is bounded by a constant that does not depend on the domain size.*

Theorems 2 and 3 together show that only two cases can occur for a URE defined on bounded regions: either the URE is not computable as soon as the evaluation region is large enough, or there is an affine schedule. In the latter case, for a URE defined on polyhedra  $\{Ap \leq Nb\}$ , the length of the longest path is  $kN + O(1)$  for some positive rational  $k$  and an affine schedule with latency  $kN + O(1)$  can be derived. See more details in [9].

#### The case of several equations

For one equation, a vector  $q \in Q(D)$  can be interpreted directly as a cycle in the RDG since all edges are connected to the same vertex: they can be used in any order. For several equations, it is more difficult to express a cycle by linear constraints and to ensure that edges are traversed in a specific order. To detect cycles of zero weight in  $G = (V, E, w)$ , the key is to consider  $G'$  the subgraph of zero-weight multicycles (union of cycles), i.e., the subgraph of  $G$  generated by the edges that belong to a union of cycles whose total weight is zero. The following theorem identifies the links between  $G$  and  $G'$ .

**Theorem 4** (a)  $G$  contains a zero-weight cycle iff its subgraph  $G'$  does. (b)  $G$  contains a zero-weight cycle iff one of its strongly connected components (SCCs) does. (c) If  $G'$  is strongly connected,  $G$  has a zero-weight cycle.

These properties give the hint for solving the problem with a recursive search. To detect a zero-weight cycle in  $G$ , it is sufficient to consider each SCC of  $G'$ . If  $G'$  is empty,  $G$  has no zero-weight multicycle, thus no zero-weight cycle. If  $G'$  has more than one SCC, then  $G$  has a zero-weight multicycle if at least one SCC has a zero-weight cycle. It remains to solve the terminating case, i.e., when  $G'$  is strongly connected, in which case  $G$  has a zero-weight cycle. Finally, this leads to the decomposition of Karp, Miller, and Winograd.

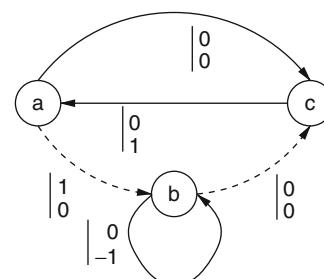
(Decomposition of Karp, Miller, and Winograd)  
Boolean KMW( $G$ ):

- Build the subgraph  $G'$  of zero-weight multicycles in  $G$ .
- Compute  $G'_1, \dots, G'_s$ , the  $s$  SCCs of  $G'$ .
  - If  $s = 0$ ,  $G'$  is empty, return TRUE.
  - If  $s = 1$ ,  $G'$  is strongly connected, return FALSE.
  - Otherwise return  $\wedge_i \text{KMW}(G'_i)$  (logical AND).

Then,  $G$  is computable iff  $\text{KMW}(G)$  returns TRUE.

Consider the SURE whose RDG is depicted in Fig. 1. The two edges  $(a, b)$  and  $(b, c)$  cannot belong to a zero-weight multicycle, since the weight of any multicycle that traverses them has a positive first component. The self-dependence on  $b$  and the cycle formed by the edges  $(a, c)$  and  $(c, a)$  define a zero-weight multicycle, thus  $G'$  is the subgraph of  $G$  obtained by deleting the two edges from  $(a, b)$  and  $(b, c)$  (see Fig. 2). It has two SCCs. For both, the subgraph of zero-weight multicycle is empty, thus the decomposition stops: both SCCs are computable, thus  $G'$  is computable, and finally  $G$  is computable too.

Let us now focus on the construction of the subgraph  $G'$ . The **cycle vector** associated to a cycle in  $G$  is a vector  $q$ , with  $|E|$  components, such that  $q_e$  is the number of times the edge  $e$  is traversed in the cycle. The **connection matrix** is a  $|V| \times |E|$  matrix  $C$  such that  $C_{v,e} = 1$  (resp.  $C_{v,e} = -1$ ) if the edge  $e$  leaves (resp. enters) vertex  $v$ , and  $C_{v,e} = 0$  otherwise. A vector  $q \geq 0$  such that  $Cq = 0$  represents a union of cycles, which is a cycle if the subgraph of  $G$  generated by the edges  $e$  such that  $q_e > 0$  is connected. Let  $W$  be the  $n \times |E|$  weight



Parallelism Detection in Nested Loops, Optimal. Fig. 2

As dotted lines, edges that do not belong to  $G'$

**matrix** whose columns are the edges weights of  $G$ . Then, the edges of  $G'$  are exactly the edges  $e$  for which  $v_e = 0$  in any optimal solution of the following linear program:

$$\min \left\{ \sum_e v_e \mid q \geq 0, v \geq 0, q + v \geq 1, Cq = 0, Wq = 0 \right\}$$

Now, to better understand what is behind this linear program, let us interpret its dual. After algebraic manipulations, it can be written as:

$$\max \left\{ \sum_e z_e \mid 0 \leq z \leq 1, t \cdot w(e) + \rho_{y_e} - \rho_{x_e} \geq z_e, \forall e \in E \right\}$$

where  $e$  goes from  $x_e$  to  $y_e$ . The complementary slackness theorem [23] shows interesting properties in the dual.

**Theorem 5** *For any optimal solution  $(z, t, \rho)$  of the dual:*

$$e \in G' \Leftrightarrow t \cdot w(e) + \rho_{y_e} - \rho_{x_e} = 0 \quad (2)$$

$$e \notin G' \Leftrightarrow t \cdot w(e) + \rho_{y_e} - \rho_{x_e} \geq 1 \quad (3)$$

**Theorem 5** shows how the constraints  $t \cdot D \geq 1$  obtained for linear scheduling in the case of a URE (remember the set  $T(D)$  in Theorem 2) can be generalized to the case of a SURE. For a URE,  $t$  was interpreted as a vector normal to a hyperplane that separates the space into two half-spaces, all dependence vectors being strictly in the same half-space. Here, the vector  $t$  defines (up to a translation given by the constants  $\rho$ ) a hyperplane which is a **strictly separating hyperplane** for the edges not in  $G'$ , see Inequality (3), and a **weakly separating hyperplane** for the edges in  $G'$ , see Equality (2). Furthermore, for each subgraph  $G$  that appears in the decomposition,  $t$  defines a hyperplane that is the “most often strict”: the number of edges, for which such a hyperplane is strict, is maximal (since  $\sum_e z_e$  is maximal). See more details in [11].

Define the **depth**  $d$  of  $G = (V, E, w)$  as the maximal number of recursive calls generated by the initial call  $KMW(G)$  (counting the first one), except if  $G$  is acyclic, in which case  $d = 0$ . The depth  $d$  is a measure of the parallelism described by  $G$ : it is related both to the length of the longest paths (intrinsic sequentiality) and to the minimal latency of particular schedules, called

shifted-linear **multi-dimensional schedules**, i.e., mappings from  $V \times \mathbb{Z}^n$  to  $\mathbb{Z}^d$ . The execution order is the lexicographic order  $\leq_{lex}$  on vectors of dimension  $d$ : the components of the schedule can be interpreted as hours, minutes, seconds, ..., described by nested loops starting from the outermost one.

For each  $v \in V$ , involved in  $d_v$  recursive calls, a sequence of vectors  $t_v^1, \dots, t_v^{d_v}$  and of constants  $\rho_v^1, \dots, \rho_v^{d_v}$  can be built by considering the dual program during the decomposition algorithm. The two sequences can be completed with zeros, if needed, to get sequences of length  $d$ .

**Theorem 6** *Let  $G = (V, E, w)$  be a computable, strongly connected RDG of depth  $d$ . If the evaluation region is a  $n$ -dimensional cube of size  $N$ , the mapping defined by  $\theta(v, p) = (t_v^1 \cdot p + \rho_v^1, \dots, t_v^{d_v} \cdot p + \rho_v^{d_v}, 0, \dots, 0)$  defines a multi-dimensional schedule with latency  $O(N^d)$ . Furthermore, the associated EDG  $\Gamma$  contains a dependence path of length  $\Omega(N^d)$ , whose projection onto  $G$  visits  $\Omega(N^{d_v})$  times each vertex  $v \in V$ .*

A dependence path of length  $\Omega(N^d)$  can be built in  $\Gamma$ , following the hierarchical structure of  $G'$ , by traversing order  $N$  times a cycle that visits all vertices of  $G$  and by plugging, during this traversal, each SCC of  $G'$  order of  $N$  times, in a recursive manner. Consider the example of Fig. 2 again. Go from  $a(1, 1)$  to  $a(1, N-1)$ , following  $(N-1)$  times the cycle between  $a$  and  $c$ . Then, go to  $b(2, N-1)$  following the edge  $(a, b)$ , and to  $b(2, 1)$  following  $(N-1)$  times the self-loop on  $b$ . Finally, go to  $c(2, 1)$  and  $a(2, 2)$  following the edges  $(b, c)$  and  $(c, a)$  once. This makes a path of length  $2(N-1)+1+(N-1)+2 = 3N$ . This pattern can be repeated  $(N-1)$  more times, leading to  $a(N, N)$ , for a path of length  $3N^2$ .

In terms of scheduling, solving the constraints of Theorem 5 shows that  $a(i, j)$  can be computed at time step  $(2i+1, 2j)$ ,  $b(i, j)$  at step  $(2i, -j)$ , and  $c(i, j)$  at step  $(2i+1, 2j+1)$ . Indeed, for the first level of the decomposition, the constraints amount to look for a vector  $t$  such that  $t \cdot (0, 1) = t \cdot (0, -1) = 0$  (for the two cycles of  $G'$ ) and  $t \cdot (1, 0) \geq 2$  (for the cycle  $(a, b, c, a)$  with two edges not in  $G'$ ), which leads, e.g., to  $t = (2, 0)$  and suitable constants  $\rho_a, \rho_b$ , and  $\rho_c$ . The second level leads to  $t = (0, 2)$  for  $a$  and  $c$ , and  $t = (0, -1)$  for  $b$ . This explicit schedule

corresponds to the code below. It is purely sequential (2-dimensional schedule, for a dependence of length order  $N^2$ , in a square of size  $N$ ). In general, if a statement is surrounded in the initial code by  $n$  loops and scheduled with a  $d$ -dimensional schedule, it will be surrounded by  $d$  sequential loops and  $(n - d)$  parallel loops in the resulting code.

```

DO i=1, N
 DO j=N, 1, -1
 b(i,j) = a(i-1,j) + b(i,j+1)
 ENDDO
 DO j=1, N
 a(i,j) = c(i,j-1)
 c(i,j) = a(i,j) + b(i,j)
 ENDDO
ENDDO

```

## Loop Transformations and Automatic Loop Parallelization

Linear programming methods, optimizations on polytopes, manipulations of integral matrices, are now commonly used in the field of automatic parallelization and program transformations, in particular for imperative codes with DO loops. The key is to represent, analyze, and transform loops without unrolling them, in an abstract way, thanks to polyhedral representations and transformations. This way, compilation methods can be developed with a complexity that depends on the (textual) size of the program and not on the number of operations it describes. DO loops are indeed, as SUREs, a condensed way for representing repetitive computations. Such an approach started in 1974 when Lamport introduced the hyperplane method [19] to parallelize perfectly nested loops. Similar techniques were applied for the automatic synthesis of systolic arrays from uniform recurrence equations. In 1988, Feautrier introduced PIP [14], a software tool for parametric (integer) linear programming, and he demonstrated its interest for dependence analysis [15], loop parallelization [16], code generation [4], etc. This work initiated many more developments based on polytopes for detecting dependences, scheduling computations,

mapping data and communications, generating code for computations and communications, etc.

The following presentation, borrowed from [10], recalls the link between the decomposition of Karp, Miller, and Winograd, and different algorithms for transforming (sequential) DO loops into DOALL loops, i.e., loops whose iterations can be computed in any order, in particular in parallel. These algorithms perform high-level source-to-source transformations, in the same way a user inserts parallelization directives in OpenMP. Exploiting the parallelism is another story, which requires data and communication optimizations, depending on the target architecture.

### Representation of DO Loops

Loop transformations apply to codes defined by nested loops, for which the control structure is simple enough to be captured with polytopes. Each loop has its own loop counter that takes, if the loop step is 1, any integer value from the lower to the upper bound, which are defined by affine expressions of the surrounding loop counters. The iterations of  $n$  perfectly nested loops can thus be represented by an **iteration domain**, set of all integer vectors in a polyhedron. When running the program, each statement  $S$  is executed for each value of the surrounding loop counters, represented by an **iteration vector**  $p$ . Such an execution is an **operation**, denoted  $S(p)$ .

Thus, as for SUREs, nested loops have an evaluation region, the iteration domain. However, unlike SUREs, the schedule is explicit and defines the semantics, while the dependences are implicit and must be pre-computed. Indeed, because each inner loop is scanned for each iteration of an outer loop, the operations  $S(p)$  are carried out in the predefined **sequential order**  $<_{seq}$ , given by the lexicographic order defined on iteration vectors plus the textual order:

$$S(p) <_{seq} T(q) \Leftrightarrow (\tilde{p} <_{lex} \tilde{q}) \text{ or } (\tilde{p} = \tilde{q} \text{ and } S <_{text} T)$$

where  $\tilde{p}$  and  $\tilde{q}$  are the vectors  $p$  and  $q$  restricted to the loop counters that surround both  $S$  and  $T$ . There is a **data dependence** from  $S(p)$  to  $T(q)$  (denoted  $S(p) \Rightarrow T(q)$ ) if both operations access the same memory location, at least one access is a write, and  $S(p) <_{seq} T(q)$ . As for SUREs, the relation  $\Rightarrow$  defines a partial order between operations, i.e., an **expanded**

**dependence graph** (EDG). To keep the program semantics, code transformations should preserve this partial order. In general, instead of representing all pairs  $(S(p), T(q))$  for which  $S(p) \Rightarrow T(q)$ , the dependences are approximated by a **reduced dependence graph** (RDG), with one vertex per statement, where the weight  $w(e)$  of each edge  $e$  describes a set  $D_e$  of **dependence distances**  $\tilde{q} - \tilde{p}$ , in a conservative way: if  $S(p) \Rightarrow T(q)$  in the EDG, then there exists  $e = (S, T)$  in the RDG such that  $\tilde{q} - \tilde{p} \in D_e$ . In other words, the RDG describes a superset of the EDG called **apparent dependence graph** (ADG). All loop transformations algorithms have to respect the dependences in the ADG. Thus, their properties, in particular their optimality for detecting parallelism, need to be analyzed with respect to the ADG (and not the EDG, which is not provided), i.e., with respect to the dependence abstraction used.

### Approximations of Distances: Dependence Level and Direction Vector

The simplest way to represent a dependence distance is to use the abstraction by dependence level. A dependence between  $S(p)$  and  $T(q)$  is **loop-independent** if it occurs for a fixed iteration of each loop surrounding both  $S$  and  $T$  (i.e.,  $\tilde{q} = \tilde{p}$ ). Otherwise, it is **loop-carried** and its **level** is the index of the first nonzero component of  $\tilde{q} - \tilde{p}$ . Then, all iterations of a loop  $L$  at depth  $k$  can be executed in any order, i.e.,  $L$  is parallel, if there is no dependence at level  $k$  in the RDG that corresponds to the code surrounded by  $L$ .

The main idea of Allen and Kennedy's parallelization algorithm [2] is to use loop distribution to reduce the number of statements within a loop, and thus the number of potential dependences. Briefly speaking, loop distribution separates, in different loops, the statements of the different SCCs of the RDG. Then, each SCC is treated separately and, according to the dependence levels, the outermost loop is marked as a DOALL or DOSEQ loop. Inner loops are treated the same way, recursively. Here is a sketch of the algorithm (different but equivalent to the original formulation). The initial call is  $AK(G, 1)$ , where  $G$  is the RDG with dependence levels.

A careful analysis [12] reveals that this algorithm is nothing but the decomposition of Karp, Miller, and Winograd, applied to an RDG with levels, except that parallel loops are generated at the outermost level,

---

(Algorithm of Allen and Kennedy)

$AK(G, k)$ :

- Remove from  $G$  all edges of level  $< k$ .
  - Compute the SCCs of  $G$ .
  - For each SCC  $C$  in topological order, do:
    - If  $C$  is reduced to a single statement  $S$ , with no edge, generate DOALL loops in all remaining dimensions, and generate code for  $S$ .
    - Else
      - Let  $l$  be the minimal dependence level in  $C$ .
      - Generate DOALL loops from level  $k$  to level  $l - 1$ , and a DOSEQ loop for level  $l$ .
      - Call  $AK(C, l + 1)$ .
- 

when possible. Indeed, if schedules are searched as in [Theorem 5](#), considering that  $w(e)$  corresponds to any distance vector represented by the dependence level of the edge  $e$ , then the only valid schedules are the elementary schedules that correspond to loop distribution/parallelization. This can also be understood with the uniformization principle mentioned in the next section. Moreover, using a proof technique similar to the one used in [Theorem 6](#), one can even prove the following optimality result.

**Theorem 7** *Allen and Kennedy's algorithm is optimal for parallelism extraction in an RDG labeled with dependence levels.*

Here, the optimality means the following. Let  $d_S$  be the number of DOSEQ loops generated around a statement  $S$ . Assume that each loop has order  $N$  iterations. Then, it is possible to build, in the ADG corresponding to the RDG, a dependence path that visits  $\Omega(N^{d_S})$  times the statement  $S$ . It is even possible to build a code, with the same RDG, whose EDG contains such a path. In other words, without any other information, there is no way to extract more parallelism.

Another popular dependence abstraction is the **direction vector** whose components belong to  $\mathbb{Z} \cup \{*, +, -\} \cup (\mathbb{Z} \times \{+, -\})$ . Its  $i$ th component is an approximation of the  $i$ th components of all possible distance vectors: it is equal to  $z+$  (resp.  $z-$ ) if all  $i$ th components are at least (resp. at most)  $z \in \mathbb{Z}$ . It is equal to  $*$  if the  $i$ th component may take any value and to  $z \in \mathbb{Z}$  if it takes the unique value  $z$ . The notation  $+$  (resp.  $-$ ) is a shortcut for  $1+$  (resp.  $(-1)-$ ). A dependence of level  $k$  corresponds

to a direction vector  $(0, \dots, 0, +, *, \dots, *)$  where  $+$  is the  $k$ -th component. Unlike the dependence level, the direction vector gives information on all dimensions of the distance vectors. But, it is still not powerful enough to express relations between different components.

When loops are perfectly nested and direction vectors are constant, loops are called **uniform**. Since dependences are directed according to the sequential order, the direction vector is always lexicopositive, its first nonzero component is positive, and [Theorem 2](#) applies: there is a linear schedule, and the code can always be rewritten with one outer sequential loop (which carries all dependences) surrounding parallel loops. Lamport's **hyperplane method** [19] is just a different way to build a linear schedule, without requiring linear programming. To reduce the number of sequential steps, [Theorem 3](#) can be applied too. Variants in which not all dependences are carried by the outermost loop lead, among others, to more subtle NP-complete retiming problems (see [8]).

Finally, a more powerful abstraction of dependence distances is to represent them by a **dependence polyhedron**, defined by a set of vertices, a set of rays, and a set of lines. A direction vector is a particular polyhedral representation. For example, the direction vector  $(2+, *, (-1)-, 3)$  defines the polyhedron with one vertex  $(2, 0, -1, 3)$ , two rays  $(1, 0, 0, 0)$  and  $(0, 0, -1, 0)$ , and one line  $(0, 1, 0, 0)$ . Thanks to a uniformization principle, an RDG with dependence polyhedra can be reinterpreted in terms of SUREs as recalled in the next section.

## Uniformization Principle: From Dependence Polyhedra to SUREs

Consider the following code example:

```
DO i=1, N
 DO j=1, N
 a(i,j) = a(i,j-1) + a(j,i)
 ENDDO
ENDDO
```

It has three dependences: a uniform flow dependence of distance  $(0, 1)$  from  $S(i, j)$  to  $S(i, j + 1)$ , a flow dependence from  $S(i, j)$  to  $S(j, i)$  if  $i < j$ , and an anti-dependence from  $S(j, i)$  to  $S(i, j)$  if  $j < i$ . The latter two dependences correspond to the same dependence

distances and can be combined. The uniform dependence has level 2 and the combined dependence has level 1, thus the algorithm of Allen and Kennedy cannot find any parallelism. The corresponding direction vectors are equal to  $(0, 1)$  and  $(+, -)$ . In the second dimension, the “ $1$ ” and the “ $-$ ” are incompatible and prevent the detection of parallelism. However, there is a linear schedule  $\theta(i, j) = 2i + j$ , which leads to a transformed code with one parallel loop:

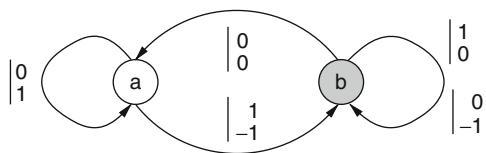
```
DO j = 3, 3N
 DOALL i=max(1, lceil (j-N)/2 rceil), min(N, lceil (j-1)/2 rceil)
 a(i,j-2i) = a(j-2i,i) + a(i,j-2i-1)
 ENDDOALL
ENDDO
```

To find this program transformation, one can notice that the set of distance vectors  $\{(j - i, i - j) \mid 1 \leq j - i \leq N - 1\}$  can be (over)-approximated by  $\mathcal{D} = \{(1, -1) + \lambda(1, -1) \mid \lambda \geq 0\}$ , i.e., a polyhedron with one vertex  $v = (1, -1)$  and one ray  $r = (1, -1)$ . Now, as for [Theorem 2](#), consider  $t$  such that  $t.d \geq 1$  for any dependence vector  $d$ . Thus,  $t.(0, 1) \geq 1$  and  $t.d \geq 1$  for all  $d \in \mathcal{D}$ . The latter inequality is equal to  $t.(1, -1) + \lambda t.(1, -1) \geq 1$  with  $\lambda \geq 0$ , which is equivalent to  $t.(1, -1) \geq 1$  and  $t.(1, -1) \geq 0$ , i.e.,  $t.v \geq 1$  and  $t.r \geq 0$ . Therefore, a valid linear schedule is defined by a vector  $t$  that satisfies the three inequalities  $t.u \geq 1$ ,  $t.v \geq 1$ ,  $t.r \geq 0$ , which leads, as desired, to  $t = (2, 1)$  and  $\theta(i, j) = 2i + j$ .

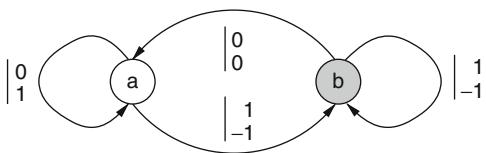
What is important here is the “uniformization” principle, which transforms an inequality on  $\mathcal{D}$  into uniform inequalities on  $v$  and  $r$ . In terms of dependence path, this amounts to consider an edge  $e$ , labeled by the distance vector  $p = v + \lambda r$ , as a path that uses once the “uniform” vector  $v$  and  $\lambda$  times the “uniform” vector  $r$ . Now, the dependence vectors are not necessarily lexicopositive anymore (e.g., a ray can be equal to  $(0, -1)$ ). Thus, the uniformized dependence graph looks more like the RDG of a SURE than the RDG of a uniform loop nest. However, the constraint imposed on a ray  $r$  is weaker, it is  $t.r \geq 0$  instead of  $t.r \geq 1$ , and  $t.l = 0$  for a line  $l$ . This freedom must be taken into account in the parallelization algorithm. This leads to the algorithm of Darte and Vivien, devoted to an RDG with dependence polyhedra [13]. The technique is simple, it consists in uniformizing the dependences so as to transform the initial RDG into the RDG of a SURE, with some new vertices emulating rays and lines. Despite the

fact that these new vertices must be considered in a special way, all results obtained for SUREs, such as Theorems 5 and 6, and the structure of the subgraph  $G'$ , can then be transferred to provide an optimal algorithm for parallelism detection in an RDG with dependence polyhedra. Furthermore, thanks to this uniformization principle, the algorithm can be specialized to a particular dependence abstraction, such as dependence level or direction vector, while keeping optimality with respect to the dependence abstraction. For the dependence level abstraction, it is the algorithm of Allen and Kennedy. For direction vectors and a single instruction (or block of instructions considered atomically), it leads to a variant of the algorithm of Wolf and Lam that would generate parallel loops instead of permutable loops (to find permutable loops, one needs to analyze the cone generated by the cycle weights, those in  $G'$  form the vector space of this cone).

Figures 3 and 4 depict the uniformized RDGs for the previous example when the non-uniform dependence is represented by a direction vector  $(+, -)$  and a dependence polyhedron along  $(1, -1)$ . In the first case, two self-loops on the new vertex  $b$ , of weight  $(1, 0)$  and  $(0, -1)$ , are introduced, resulting in a nonempty subgraph  $G'$ , and no parallelism. In the second case, the self-loop on  $b$  has weight  $(1, -1)$  and the RDG has no multicycle of zero weight, and thus contains some parallelism. Similarly, the graph of Fig. 1 is the uniformized



Parallelism Detection in Nested Loops, Optimal. Fig. 3  
Uniformized graph for direction vectors



Parallelism Detection in Nested Loops, Optimal. Fig. 4  
Uniformized graph for polyhedron vector

RDG for the following code, where  $b$  is the vertex introduced by the uniformization:

```
DO i=1, N
 DO j=1, N
 a(i,j) = c(i,j-1)
 c(i,j) = a(i,j) + a(i-1,N)
 ENDDO
ENDDO
```

### Going Beyond, with the Affine Form of Farkas Lemma

So far, the discussion focused, as in Theorem 6, on loop transformations based on multi-dimensional scheduling functions (with the lexicographic order) called shifted-linear, i.e., of the form  $\theta(v, p) = (t_v^1 \cdot p + \rho_v^1, \dots, t_v^{d_v} \cdot p + \rho_v^{d_v}, 0, \dots, 0)$  where, for all vertices of the same SCC encountered at depth  $i$  of the decomposition, the linear part  $t_v^i$  is the same. Different constants (similar to retiming [20]) can be used for different statements however. The fact that the linear part is the same made life easier. Indeed, to get valid scheduling functions, one had to solve inequalities of the form  $\theta(T, q) - \theta(S, p) \geq \epsilon$  ( $\epsilon = 0$  or  $1$  as in Theorem 5), i.e.,  $t_T(q-p) + \rho_T - \rho_S \geq \epsilon$ . If the dependence distance  $q-p$  is constant, one directly ends up with a system of linear inequalities. Otherwise, as just showed, the set of all  $q-p$  can be approximated by a polyhedron and linear inequalities involving the vertices, rays, and lines of this polyhedron are obtained.

Now, what if even more general functions are searched, i.e., affine functions with a different linear part for each statement? This is the approach of Feautrier [16]. With  $\theta(S, p) = t_S \cdot p + \rho_S$ , the constraints that need to be solved are then of the form  $t_T \cdot q - t_S \cdot p + \rho_T - \rho_S \geq \epsilon$  for all  $p$  and  $q$  such that  $S(p) \Rightarrow T(q)$ . The number of inequalities depends on the number of  $p$  and  $q$ , which is not practical. However, if the set of pairs  $(p, q)$  such that  $S(p) \Rightarrow T(q)$  can be described by a polyhedron, the **affine form of Farkas lemma** can be used to simplify the inequalities. This lemma states that  $c \cdot p \leq \delta$  for all vectors  $p$  in a polyhedron  $\{p \mid Ap \leq b\}$  iff  $c = y \cdot A$  for some vector  $y \geq 0$  such that  $y \cdot b \leq \delta$ . With this mechanism, an inequality involving all  $(p, q)$  in a polyhedron can be replaced by a finite set of inequalities.

The affine form of Farkas lemma is the key tool for writing inequalities in Feautrier's algorithm, which generates general multi-dimensional affine scheduling

functions. The skeleton of the algorithm itself is similar to the decomposition of Karp, Miller, and Winograd: trying to find a function for which as many dependences as possible are satisfied. As for previous algorithms, some optimality result can be formulated, but of a different nature. For affine dependences, i.e., if  $p$  is expressed as an affine function of  $q$ , for  $q$  in a polyhedron, whenever  $S(p) \Rightarrow T(q)$ , optimal parallelism detection requires more than affine functions, in particular index splitting, i.e., piecewise affine functions. Thus Feautrier's algorithm cannot be optimal with respect to the dependence abstraction it was designed for. However, among all affine functions, it can find one with the "right" parallelism extraction, in other words, the algorithm is optimal with respect to the class of functions it considers [24]. Extensions to detect outer parallel loops, to derive permutable loops for tiling, and to generate codes where not all dependences are carried have also been proposed, see, e.g., [3, 21].

### Multi-dimensional Affine Ranking Functions and Program Termination

Recall the graph of Fig. 1. It is the RDG of a SURE with three equations,  $a$ ,  $b$ ,  $c$ . Starting from such a description of repetitive computations, with explicit evaluation region, implicit schedule, and explicit dependences, an explicit schedule for it was derived. As seen in the previous section, this RDG can also be interpreted as the uniformized RDG of two Fortran-like nested loops where  $b$  is a dummy vertex added to emulate the  $(1, 0-)$  direction vector. In this case, from a description of repetitive computations, with explicit iteration domain and explicit initial schedule (the sequential order), dependences that were implicit are first computed and abstracted.

Then, another schedule is derived that respects the dependences and expresses, possibly, some parallelism. Now, consider the following C-like code example:

```

 $y = 0; x = 0;$
while ($x \leq N$ and $y \leq N$) {
 if (unknown) {
 $x = x + 1;$
 while ($y \geq 0$ and unknown) $y = y - 1;$
 }
 $y = y + 1;$
}

```

This code is yet another description of repetitive computations. Here, the schedule is explicit, it is the sequential schedule. However, loop counters are not specified, no iteration domain is specified, and the program may not terminate. The program is controlled by a parameter  $N$  and the integer variables  $x$  and  $y$  whose values are modified by the program. Now, the RDG of Fig. 1 depicts, not the dependences between computations, but how integer variables, implied in the program control, evolve. Each vertex corresponds to a state (program point + values of variables), edges represent transitions, i.e., modifications of variables. Again, this leads to a model of computations similar to the model of SUREs, for which the techniques and results previously exposed can be useful. In this example, the program can be proved to terminate, after performing  $O(N^2)$  operations.

### Integer Interpreted Automata and Invariants

The following presentation is borrowed from [1]. To prove the termination of an imperative program, a standard approach is to transform it into an **affine integer interpreted automaton**  $(\mathcal{K}, n, k_{init}, \mathcal{T})$  defined by (1) a finite set  $\mathcal{K}$  of control points, (2)  $n$  integer variables represented by a vector  $x$  of size  $n$ , (3) an initial control point  $k_{init} \in \mathcal{K}$ , and (4) a finite set  $\mathcal{T}$  of 4-tuples  $(k, g, a, k')$ , called **transitions**, where  $k \in \mathcal{K}$  (resp.  $k' \in \mathcal{K}$ ) is the source (resp. target) control point. The **guard**  $g : \mathbb{Z}^n \mapsto \mathbb{B} = \{\text{true}, \text{false}\}$  is a logical formula expressed with affine inequalities  $Gx + g \geq 0$  and the **action**  $a : \mathbb{Z}^n \mapsto \mathbb{Z}^n$  assigns, to each variable valuation  $x$ , a vector  $x'$  of size  $n$ , expressed by an affine expression  $x' = Ax + a$ .

The guard  $g$  in the transition  $t = (k, g, a, k')$  gives a necessary condition on variables  $x$  to traverse the transition  $t$  from  $k$  to  $k'$ , and to apply its corresponding action  $a$ . If, for two transitions going out of  $k$ , the guards describe non-disjoint conditions, the automaton expresses some non-determinism. To approximate non-affine or non-analyzable assignments in the program, the link between  $x$  and  $x'$  can be described by affine relations instead of functions, which introduces another form of non-determinism. Unlike for SUREs, this non-determinism can be unbounded, i.e., a single transition can give rise to an unbounded number of "successors"  $x'$  for a given  $x$ .

Unlike for DO loops and SUREs where the range of iteration vectors is explicitly defined, with the iteration domain and the evaluation region respectively, here, the set of all possible values for  $x$  at control point  $k$ , denoted  $\mathcal{R}_k$ , is implicit and hard to compute exactly. However, it is possible to over-approximate  $\mathcal{R}_k$  by an **invariant** at control point  $k$ , i.e., a formula true for all reachable states  $(k, x)$ . Polyhedral invariants can be computed with **abstract interpretation** techniques, widely studied since the seminal paper of Cousot and Halbwachs [6]. In this case,  $\mathcal{R}_k$  is over-approximated by the integer points within a polyhedron  $\mathcal{P}_k$ , which represents all the information on the variables at control point  $k$  that can be deduced from the program by state-of-the-art analysis techniques.

### Termination and Ranking Functions

Invariants can only prove partial correctness of a program. The standard technique for proving termination is to consider a **ranking function** to a well-founded set, i.e., a set  $\mathcal{W}$  with a (possibly partial) order  $\leq$  (the notation  $a < b$  means  $a \leq b$  and  $a \neq b$ ) with no infinite descending chain, i.e., no infinite sequence  $(x_i)_{i \in \mathbb{N}}$  with  $x_i \in \mathcal{W}$  and  $x_{i+1} < x_i$  for all  $i \in \mathbb{N}$ . More precisely, a ranking is a function  $\rho : \mathcal{K} \times \mathbb{Z}^n \rightarrow \mathcal{W}$ , from the automaton states to a well-founded set  $(\mathcal{W}, \leq)$ , whose values decrease at each transition  $t = (k, g, a, k')$ :

$$(x \in \mathcal{R}_k) \wedge (g(x) = \text{true}) \wedge (x' = a(x)) \\ \Rightarrow \rho(k', x') < \rho(k, x) \quad (4)$$

The ranking is said to be affine if it is affine in the second parameter (the variables). It is **one-dimensional** if its co-domain is  $(\mathbb{N}, \leq)$  and  **$d$ -dimensional** (or multi-dimensional of dimension  $d$ ) if its co-domain is  $(\mathbb{N}^d, \leq_d)$ , where the order  $\leq_d$  is the standard lexicographic order on integer vectors.

Obviously, the existence of a ranking function implies program termination for any valuation  $v$  at the initial control point  $k_{init}$ . A well-known property is that an integer interpreted automaton terminates for any initial valuation if it has a ranking function. Furthermore, if it terminates and has bounded non-determinism, there is a one-dimensional ranking function (but it is not necessarily affine). The problem is now very similar to the scheduling problem described for SUREs and for DO loops, except that dependences are considered in the opposite direction. In the same way, affine multi-dimensional ranking functions can be derived.

Considering rankings with  $d > 1$  is mandatory to be able to prove the termination of programs that induce a number of transitions, i.e., a trace length, more than linear in the program parameters. Considering rankings with a different affine function for each control point also extends the set of programs whose termination can be determined, compared, e.g., to shifted-linear rankings or to the technique of [5].

### A Greedy Complete Polynomial-Time Procedure

A ranking function  $\rho$  of dimension  $d$  needs to satisfy two properties. First, as  $\rho$  has co-domain  $\mathbb{N}^d$ , it should assign a nonnegative integer vector to each relevant state:

$$x \in \mathcal{P}_k \Rightarrow \rho(k, x) \geq 0 \text{ (component-wise)} \quad (5)$$

Second, it should decrease on transitions. Let  $\mathcal{Q}_t$  be the polyhedron giving the constraints of a transition  $t = (k, g, a, k')$ , i.e.,  $x \in \mathcal{P}_k$ ,  $g(x)$  is true, and  $x' = a(x)$ .  $\mathcal{Q}_t$  is built from the matrices  $A$  and  $G$ , and the vectors  $a$  and  $g$ . For an automaton whose actions are general affine relations,  $\mathcal{Q}_t$  is directly given by the action definitions. With  $\Delta_t(\rho, x, x') = \rho(k, x) - \rho(k', x')$ , Inequality (4) then becomes:

$$(x, x') \in \mathcal{Q}_t \Rightarrow \Delta_t(\rho, x, x') >_d 0 \quad (6)$$

which means  $\Delta_t(\rho, x, x') \neq 0$  and its first nonzero component is positive. If this component is the  $i$ -th, the **level** of  $\Delta_t(\rho, x, x')$  is  $i$ . A transition  $t$  is said to be (fully) **satisfied by the  $i$ -th component** of  $\rho$  (or **at dimension  $i$** ) if the maximal level of all  $\Delta_t(\rho, x, x')$  is  $i$ . To build a ranking  $\rho$ , the same greedy mechanism as in [5, 16, 18] can be used. The components of  $\rho$ , functions from  $\mathcal{K} \times \mathbb{Z}^n$  to  $\mathbb{N}$ , are built from the first one to the last one. For a component  $\sigma$  of  $\rho$  and a transition  $t$  not yet satisfied by one of the previous components of  $\rho$ , the following constraint is considered:

$$(x, x') \in \mathcal{Q}_t \Rightarrow \Delta_t(\sigma, x, x') \geq \epsilon_t \text{ with } 0 \leq \epsilon_t \leq 1 \quad (7)$$

and a ranking is selected for which as many transitions as possible have  $\epsilon_t = 1$ , i.e., are now satisfied. Again, inequalities such as (7) are captured thanks to the affine form of Farkas lemma. The algorithm itself has the same structure as the decomposition of Karp, Miller, and Winograd, and of Feautrier's algorithm.

(Generation of a multi-dimensional affine ranking)

- 1:  $i = 0; T = \mathcal{T}$ ;  $\triangleright$  Initialize  $T$  to all transitions
- 2: **while**  $T$  is not empty **do**
- 3:   Find a 1D affine function  $\sigma$  and values  $\epsilon_t$  such that all inequalities (5) and (7) are satisfied and as many  $\epsilon_t$  as possible are equal to 1;  
       $\triangleright$  This means maximizing  $\sum_{t \in T} \epsilon_t$
- 4:   Let  $\rho_i = \sigma ; i = i + 1$ ;  $\triangleright \sigma$  defines the  $i$ -th component of  $\rho$
- 5:   If no transition  $t$  with  $\epsilon_t = 1$ , **return** false  $\triangleright$  No multi-dimensional affine ranking.
- 6:   Remove from  $T$  all transitions  $t$  such that  $\epsilon_t = 1$ ;  
       $\triangleright$  The transitions have level  $i$
- 7: **end while;**
- 8:  $d = i$ ; **return** true;  $\triangleright d$ -dimensional ranking found

Since nonterminating programs exist, there is no hope of proving that a ranking function always exists. Moreover, there are terminating affine interpreted automata with no multi-dimensional affine ranking. Thus, what can be proved is only that, if a multi-dimensional affine ranking exists, the algorithm finds one, i.e., it is **complete** for the class of multi-dimensional affine rankings. Also, as the sets  $\mathcal{R}_k$  are over-approximated by the invariants  $\mathcal{P}_k$ , completeness has to be understood with respect to these invariants, which means that if the algorithm fails when an affine ranking exists, it is because invariants are not accurate enough.

**Theorem 8** *If an affine interpreted automaton, with associated invariants, has a multi-dimensional affine ranking function, then the greedy algorithm finds one. Moreover, the dimension of the generated ranking is minimal.*

## Conclusion

In [18], Karp, Miller, and Winograd introduced several new concepts and techniques that gave rise to important developments in the context of loop transformations and program analysis. The key is to represent a repetitive scheme of computations, even infinite, by a finite structure, the reduced dependence graph (and its variants). This allows a compiler to manipulate a program in a time that depends on the structure of the code but not on the number of operations that it describes. In other words, there is no need to unroll loops to understand what they do. Linear programming

techniques and polyhedral representations can be used to analyze and optimize such programs, in a parametric way.

This essay mentioned three related problems: determining if a system of uniform recurrence equations is computable, transforming DO loops so as to reveal parallel loops, and proving the termination of programs with IFs and WHILE loops, thanks to affine ranking functions. The link with program termination is still to be explored. Indeed, if the derivation of affine rankings is similar to the derivation of affine schedules, there are many subtle differences, in particular concerning the underlying iteration domains (invariants). One of the most challenging problems is to derive piecewise affine rankings to prove the termination of many more programs. For the detection of parallelism, deriving parallel codes with more data reuse and a better handling of memory transfers is still a challenge.

## Related Entries

- Dependence Abstractions
- Dependence Analysis
- Dependences
- Loop Nest Parallelization
- Loops, Parallel
- Parallelization, Automatic
- Polyhedron Model
- Scheduling Algorithms
- Tiling
- Unimodular Transformations

## Bibliography

1. Alias C, Darte A, Feautrier P, Gonnord L (2010) Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In 17th International Static Analysis Symposium (SAS'10). Lecture notes in computer science, vol 6337. Springer Verlag, Perpignan, pp 117–133
2. Allen JR, Kennedy K (1987) Automatic translation of Fortran programs to vector form. ACM Trans Program Lang Syst 9(4): 491–542
3. Bondhugula U, Baskaran MM, Krishnamoorthy S, Ramanujam J, Rountev A, Sadayappan P (2008) Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In Compiler Construction (CC'08). Lecture notes in computer science, vol 4959. Springer Verlag, pp 132–146
4. Collard J-F, Feautrier P, Risset T (1995) Construction of DO loops from systems of affine constraints. Parallel Process Lett 5(3): 421–436

5. Colón MA, Sipma HB (2002) Practical methods for proving program termination. In 14th International Conference on Computer Aided Verification (CAV). Lecture notes in computer science, vol 2404. Springer Verlag, pp 442–454
6. Cousot P, Halbwachs N (1978) Automatic discovery of linear restraints among variables of a program. In 5th ACM Symposium on Principles of Programming Languages (POPL'78). ACM, Tucson, pp 84–96
7. Darte A (2010) Understanding loops: The influence of the decomposition of Karp, Miller, and Winograd. In 8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'10). IEEE Computer Society, Grenoble, pp 139–148
8. Darte A, Huard G (2002) Complexity of multi-dimensional loop alignment. In 19th International Symposium on Theoretical Aspects of Computer Science (STACS'02), vol 2285. Springer Verlag, pp 179–191
9. Darte A, Khachiyan L, Robert Y (1991) Linear scheduling is nearly optimal. *Parallel Process Lett* 1(2):73–81
10. Darte A, Robert Y, Vivien F (2000) Scheduling and Automatic Parallelization. Birkhauser. ISBN 0-8176-4149-1
11. Darte A, Vivien F (1995) Revisiting the decomposition of Karp, Miller, and Winograd. *Parallel Process Lett* 5(4):551–562
12. Darte A, Vivien F (1997) On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops. *J Parallel Algorithms Appl* 12(1–3):83–112
13. Darte A, Vivien F (1997) Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *Int J Parallel Program* 25(6):447–497
14. Feautrier P (1988) Parametric integer programming. *RAIRO Rech Opérationnelle* 22:243–268
15. Feautrier P (1991) Dataflow analysis of array and scalar references. *Int J Parallel Program* 20(1):23–51
16. Feautrier P (1992) Some efficient solutions to the affine scheduling problem, part II: Multi-dimensional time. *Int J Parallel Program* 21(6):389–420
17. Gulwani S, Mehra KK, Chilimbi T (2009) SPEED: Precise and efficient static estimation of program computational complexity. In 36th ACM Symposium on Principles of Programming Languages (POPL'09). ACM, Savannah, pp 127–139
18. Karp RM, Miller RE, Winograd S (1967) The organization of computations for uniform recurrence equations. *J ACM* 14(3):563–590
19. Lamport L (1974) The parallel execution of DO loops. *Commun ACM* 17(2):83–93
20. Leiserson CE, Saxe JB (1991) Retiming synchronous circuitry. *Algorithmica* 6(1):5–35
21. Lim AW, Lam MS (1997) Maximizing parallelism and minimizing synchronization with affine transforms. In 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97). ACM, New York, pp 201–214
22. Podelski A, Rybalchenko A (2004) A complete method for the synthesis of linear ranking functions. In Verification, Model Checking, and Abstract Interpretation (VMCAI'03). Lecture notes in computer science, vol 2937. Springer Verlag, pp 239–251
23. Schrijver A (1986) Theory of Linear and Integer Programming. Wiley, New York
24. Vivien F (2003) On the optimality of Feautrier's scheduling algorithm. *Concurr Comput* 15(11–12):1047–1068
25. Wolf ME, Lam MS (1991) A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans Parallel Distributed Syst* 2(4):452–471

## Parallelization

- [FORGE](#)
- [Loop Nest Parallelization](#)
- [Parafrase](#)
- [Parallelism Detection in Nested Loops, Optimal](#)
- [Parallelization, Automatic](#)
- [Parallelization, Basic Block](#)
- [Polaris](#)
- [Run Time Parallelization](#)
- [Speculative Parallelization of Loops](#)

## Parallelization, Automatic

DAVID PADUA

University of Illinois at Urbana-Champaign, Urbana, IL, USA

### Synonyms

[Parallelization](#)

### Definition

Autoparallelization is the translation of a sequential program by a compiler into a parallel form that outputs the same values as the original program. For some authors, autoparallelization means only translation for multiprocessors. However, this definition is more general and includes translation for instruction level, vector, or any other form of parallelism.

## Discussion

### Introduction

The compilers of most parallel machines are autoparallelizers, and this has been the case since the earliest parallel supercomputers, the Illiac IV and the TI ASC, were introduced in the 1960s. Today, there are autoparallelizers for vector processors, VLIW processors, microprocessor vector extensions, and multiprocessors.

Autoparallelization is for productivity. Autoparallelizers, when they succeed, enable the programming of parallel machines with conventional languages such as Fortran or C. In this programming paradigm, code is not complicated by parallel constructs and the obfuscation typical of manual tuning.

Inserting explicit parallel constructs and tuning is not only time-consuming but also produces non-portable, machine-dependent code. For example, codes written for multiprocessors and those for SIMD machines have different syntax and organization. On the other hand, with the support of autoparallelization, conventional codes could be portable across machine classes.

Explicit parallelism introduces opportunities for program defects that do not arise in sequential programming. With autoparallelization, the code has sequential semantics. There is no possibility of deadlock and programs are determinate. The downside is that it is not possible to implement asynchronous algorithms, although this limitation does not affect the vast majority of applications.

### Requirements for Autoparallelization

A parallelizing compiler must analyze the program to detect implicit parallelism and identify opportunities for restructuring transformations, and then apply a sequence of transformations.

Detection of implicit parallelism can be accomplished by (1) computing the dependences to determine where the sequential order of the source program can be relaxed, and (2) analyzing the semantics of code segments to enable the selection of alternative parallel algorithms.

The transformation process is restricted by the information provided by this analysis and is guided by heuristics supported by static prediction of execution time or program profiling.

### Dependence Analysis

The dependence relation is a partial order between operations in the program that is computed by analyzing variable and array element accesses. Executing the program following this partial order guarantees that the program will produce the same output as the original code. For example, in

```
for (i=0; i < n; i++) {a[i] += 1;}
for (j=0; j < n; j++) {b[j] = a[j]*2;}
```

corresponding iterations of the first and the second loop must be executed in the specified order. However, these pairs of iterations do not interact with other pairs and therefore do not have to execute in the original order to produce the intended result. Only corresponding iterations of these two loops are ordered. By determining what orders must be enforced, dependence analysis tells us which reordering is valid and what can be done in parallel: two operations that are not related by the partial order resulting from the dependence analysis can be reordered or executed in parallel with each other.

Dependence analysis can be done statically, by a compiler; or dynamically, during program execution.

Static analysis is discussed next, while dynamic analysis is discussed below under the heading of “Runtime Resolution”.

How close the dependences generated by static dependence analysis are to the minimum number of ordered pairs required for correctness depends on the information available at compile time and the algorithms used for the analysis. The loops above are examples of loops that can be analyzed statically with total accuracy because (1) all the information needed for an accurate analysis is available statically, and (2) the subscript expressions are simple, so that most analysis algorithms can analyze them accurately. Accuracy is tremendously important because when the set of dependences computed by a test is not accurate, spurious dependences must be assumed and this may preclude valid transformations including conversion into parallel form.

There are numerous algorithms for dependence analysis that have been developed through the years. They typically trade off accuracy for speed of analysis. For example, some fast tests do not make use of information about the values of the loop indices, while others require them. Ignoring the loop limits works well in some cases. The loops above are an example of this situation. The value of  $n$  in these loops is not required to do an accurate analysis. However, in other cases, knowledge of the loop limits is needed. Consider the loop

```
for (i=10; i<15; i++) {a[i] += a[i-8];}
```

The loop limits, 10 and 14, are necessary to determine that no ordering needs to be enforced between loop iterations since, for these values, the iterations do not

interact with each other. A test that ignores the loop limits will report that (some) iterations in this loop must be executed in order.

Some of the most popular dependence tests require for accuracy that the subscript expressions be affine expression of the loop indices and the values of the coefficients and the constant be known at compile time. For example, a test that requires knowledge of the numerical values of coefficients would have to assume that iterations of the loop

```
if (m > 0) {
 for (i=0; i < n; i+=2){
 a[m*i] += a[m*i+1];
 }
}
```

must be executed in order, while a test with symbolic capabilities would be able to determine that the iterations do not have to be executed in any particular order to obtain correct results. [Table 1](#) presents the main characteristics of a few dependence tests.

When the needed information is not available at compile time or the analysis algorithm is inaccurate, the decision can be postponed to execution time (see “Transformations for Runtime Resolution” below). For example, the loop

```
for (i=0; i < n; i++) {a[i+k] += a[i];}
```

can be transformed into an array operation as long as k is negative, but the compiler will not know that this is the case if k happens to be a function of the input to the program or if the value propagation analysis conducted by the compiler cannot decide that k is negative. A similar situation arises in the loop

```
for (i=0; i < n; i++) {a[m[i]] += a[i];}
```

where m[i] must be  $\leq i$  or  $\geq n$  and all the m[i]’s be different for a transformation into vector operation to be

valid. But this will only be known to the compiler, if it can propagate array values and these values are available in the source code. Otherwise, dependences must be assumed or the analysis postponed to execution time.

## Semantic Analysis

Semantic analysis identifies operators or code sequences that have a parallel implementation. A good example is the analysis of array operations. For example, in the Fortran statement

```
a(1:n) = sin(a(2:n+1))
```

the n evaluations of sin can proceed in parallel since their parameters do not depend on each other.

Although array operations like this can be interpreted as parallel operations, most Fortran 90 compilers at the time of the writing of this entry do not parallelize directly array operations, but instead translate them into loops which are analyzed by later passes for parallelization. So, in effect, they rely on semantic analysis.

The compiler can also apply semantic analysis to sequence of statements with the help of a database of patterns. For example,

```
for (i=0; i < n; i++) {s+=a[i];}
```

cannot be parallelized by relying exclusively on dependence analysis, because this analysis will only state the obvious: that each iteration requires the result of the previous one (the values of sum) to proceed. However, accumulations like this can be parallelized, if assuming that + is associative is acceptable, and are frequently found in real programs. Therefore, this pattern is a natural candidate for inclusion in this database.

Other frequently found patterns include: finding the minimum or maximum of an array, and linear recurrences such as

```
x[i]=a[i]*x[i-1]+b[i]
```

**Parallelization, Automatic. Table 1** Characteristics of a few dependence tests

| Test name     | # of loop indices in subscript | Subscript expressions must be affine? | Uses loop bounds? | Ref. |
|---------------|--------------------------------|---------------------------------------|-------------------|------|
| ZIV           | 0 (constant)                   | Y                                     | N/A               | [5]  |
| SIV           | 1                              | Y                                     | Y                 | [5]  |
| GCD           | Any                            | Y                                     | N                 | [2]  |
| Banerjee      | Any                            | Y                                     | Y                 | [2]  |
| Access Region | Any                            | N                                     | Y                 | [9]  |

Some compilers have been known to recognize more complex patterns such a matrix–matrix multiplication.

Once the compiler knows the type of operation, it can choose to replace the code sequence with a parallel version of the operation.

## Program Transformations

Program transformations are used to

1. Reduce the number of dependences
2. Generate code for runtime resolution, that is, code that at runtime decides whether to execute in parallel
3. Schedule operations to improve locality or parallelism

### Transformations for Reducing the Number of Dependences

This class of transformations aims at reducing the number of ordered pairs to improve parallelism and enable reordering. Induction variable substitution and privatization are two of the most important examples in this class. Induction variables are those that assume values that form an arithmetic sequence. Their computation creates a linear order that must be enforced. In addition, using induction variables in subscripts hinders the dependence analysis of other computations. For example, the loop

```
for (i=0; i<n; i++) {j+=2; a[j]=a[j]*2;}
```

cannot be parallelized in this form since  $j+=2$  must be executed in order. Furthermore, dependence analysis cannot know that each iteration of the loop accesses a different element unless it knows that  $j$  takes a different value in each iteration. Fortunately, in this example, as in most cases, the induction variable can be eliminated to increase parallelism and improve accuracy of analysis. Thus, here  $j$  may be represented in terms of the loop index and forward substituted:

```
for (i=0; i<n; i++) {a[j+2*i+1]=a[j+2*i+1]*2;}
```

The effect of this transformation is that the chain of dependences resulting from the  $j++$  statement goes away with the statement. Also, the removal of the increment makes  $j$  a loop invariant and this enables an accurate dependence analysis at compile time.

The identification of induction variables was originally developed for strength reduction, which replaces

operations with less expensive ones. A typical strength reduction is to replace multiplications with additions. For parallelism, the replacement goes in the opposite direction. For example, additions are replaced by multiplications as shown in the last example. Induction variable identification relies on conventional compiler data-flow analysis.

Privatization can be applied when the  $a$  variable is used to carry values from one statement to another within the one iteration of the loop. For example, in

```
for(i=0; i<n; i++) {a=b[i]*2; c[i] = a*c[i]}
```

the use of a single variable,  $a$ , in all iterations demands that the iterations be executed in order to guarantee correct results because,  $a$  should not be reassigned until its value has been obtained by the second statement of the loop body. The privatization transformation simply makes  $a$  private to the loop iteration and thus eliminates a reason to execute the iterations in order.

An alternative to privatization is expansion. This transformation converts the scalar into an array and has the same effect on the dependence as privatization. For the previous loop, this would be the result:

```
for (i=0; i<n; i++) {
 a1[i]=b[i]*2;
 c[i]=a1[i]*c[i]
}
a=a1[n-1];
```

Privatization is applied when generating code for multiprocessors, and expansion is necessary for vectorization. The main difficulty with expansion is the increase in memory requirements. While privatization increases the memory requirements proportionally to the number of processors, expansion does so proportionally to the number of iterations, a number that is typically much higher.

However, expansion can be applied together with a transformation called stripmining to reduce the amount of additional memory.

Privatization and expansion require analysis to determine that the variable being privatized or expanded is never used to pass information across iterations of the loop. This analysis can be done using conventional data flow analysis techniques.

### Transformations for Runtime Resolution

In its simplest form, runtime resolution transformations generate if statements to select between a parallel or serial version of the code. For example,

```
do i=m,n
 a(i+k)=a(i)*2
end do
```

as discussed above, can be vectorized if  $k \leq 0$ . The compiler may then generate a two-version code

```
if (k<=0) then
 a(k+m:k+n) += a(m:n)
else
 do i=m,n
 a(i+k)=a(i)*2
 end do
end if
```

Two-version code can be also be used in other situations. Thus, if the loop contains an assignment statement that accesses memory through pointers in the right- and left-hand sides, such as the loop

```
for (i=0; i <n; i++) {*(a+i)=*(b+i)+2;}
```

the if statement should check that address a is either less than address b or greater than address  $(b+n-1)$ .

More complex runtime resolution would be needed for loops like

```
for (i=0; i <n; i++) {a[m[i]]+=a[i];}
```

where the  $m[i]$ 's must be  $\leq i$  or  $\geq n$  and all distinct for vectorization to be possible, or  $m[i]$  either =  $i$  or outside the values in the iteration space and all distinct for transformation into a parallel loop. In

```
for (i=0; i <n; i++) {a[m[i]]+=a[q[i]];}
```

the  $m[i]$ 's and  $q[i]$ 's must be such that  $m[i] \leq q[i]$  and the  $m[i]$ 's all distinct for vectorization, or  $m[i] \neq q[j]$  whenever  $i \neq j$  for parallelization. Two-version loops can be generated also in this case, but the if condition is somewhat more complex as it must analyze a collection of addresses. In this last case, the technique is called inspector-executor. Another approach to runtime resolution is speculation, which attempts to execute in parallel and optimistically expects that there will be no conflicts between the different components executing in parallel. During the execution of the speculative parallel code or at the end, the memory references are checked to make sure that the parallel execution was correct.

If it was not, the execution is undone and the components executed at a later time either in the right order or again speculatively, in parallel.

Run time resolution is also used to check for profitability, i.e., that parallel execution will make execution faster. For example, if the number of iterations of a parallel loop is not known at compile time, runtime resolution can be used to decide whether to execute a loop in parallel as a function of the number of iterations. Also, runtime resolution can be used to guarantee that vector operations are only executed if the operands are or can be properly aligned in memory when this is required for performance. For example, SSE vector operations sometimes perform better when the operands are aligned on double word boundaries.

### Scheduling Transformations

An important class are the transformations that schedule the execution of program operations or partition these operations into groups. To enforce the order, the compiler typically uses the barriers implicit in array operations or multiprocessor synchronization instructions. One such transformation is stripmining. It partitions the iterations of a loop into blocks by augmenting the increment of the loop index and adding an inner loop as follows:

```
for (i=0; i <n; i++) {a[i]=a[i]+1;}

↓

for (i=0; i < (n/q)*q; i+=q) {
 for(j=i; j < i+q, j++) {
 a[j]=a[j]+1;
 }
}

for (i=(n/q)*q; i < n, i++) {a[i]=a[i]+1;}
```

Stripmining is useful to enhance locality and reduce the amount of memory required by the program. In particular, it can be used to reduce the memory consumed by expansion. If the goal is vectorization and the size of the vector register is  $q$ , this transformation will not reduce the amount of parallelism.

Another type of loop partitioning transformation is that developed for a class of autoparallelizing compilers targeting distributed memory operations. These compilers, including High-Performance Fortran and Vienna Fortran, flourished in the 1990s but are no longer used. The goal of partitioning was to organize loop iterations groups so that each group could

be scheduled in the node containing the data to be manipulated.

An important sequencing transformation is loop interchange, which changes the order of execution by exchanging loop headers. This transformation can be useful to reduce the overhead when compiling for multiprocessors and to enhance memory behavior by reducing the number of cache misses. For example, the loop

```
for (i=0; i < (n/q)*q; i+=q)
 for(j=i; j< i+q, j++) {
 a[j]=a[j]+1;
 }
```

can be correctly transformed by loop interchange into

```
for (j=0; j<n; j++)
 for(i=0; i<n, i++) {
 a[i][j]=a[i-1][j]+1;
 }
```

The outer loop of the original nest cannot be executed in parallel. If nothing else is done, the only option of the compiler targeting a multiprocessor is to transform the inner loop into parallel form and while this could lead to speedups, the result would suffer of the parallel loop initiation overhead once per iteration of the outer loop. Exchanging the loop headers makes the iteration of the outer loop independent so that the outer loop can be executed in parallel and the overhead is only paid once per execution of the whole loop. Furthermore, the resulting loop has a better locality since the array is traverse in the order it is stored, so that the elements of the array in a cache line are accessed in consecutive order, improving in this way spatial locality.

A third example of sequencing transformation is instruction level parallelization. Consider, for example, a VLIW machine with a fixed point and a floating point unit. The sequence

```
r1=r2+r3
r4=r4+r5
f1=f1+f2
f3=f4+f5
```

contains two fixed point operations (those operating on the r registers) and two floating point operations. Exchanging the second and the third operation is necessary to enable the creation of two (VLIW) instructions each making use of both computational units.

In some cases, the partitioning and sequencing of the operations is not completely determined at compile time. For example the sum reduction

```
for (i=0; i <n; i++) { s+=a[i]; }
```

once identified as such by semantic analysis, may be transformed into a form in which subsets of iterations are executed by different threads and the elements of a are accumulated into different variables, one per thread of execution. These variables are then added to obtain the final sum. The number of these threads can be left undefined until execution time. In OpenMP notation, this can be represented as follows:

```
#pragma omp parallel
{ float sp=0;
#pragma omp for
for (i=0; i <n; i++) {
 sp+=a[i];
}
#pragma omp single
{s+=sp;}
```

or, more simply,

```
#pragma omp parallel for reduction (+: sum)
for (i=0; i <n; i++) {
 sp+=a[i];
}
```

It should be pointed out that in this example, it has been assumed that floating point addition is associative, but because of the finite precision of machines, it is not. In some cases, it is correct to do this transformation, even if the result obtained is not exactly the same as that of the original program.

However, this is not always the case and transformations like this require authorization from the programmer. [Table 2](#) contains a list of important transformations not discussed above.

## Autoparallelization Today

Most of today's compilers that target parallel machines are autoparallelizers. They can generate code for multiprocessors and vector code. Although autoparallelization techniques have become the norm, the few empirical studies that exist as well as anecdotal evidence indicate that these compilers often fail to generate high-quality parallel code. There are two reasons for this. First, sometimes the compiler fails to find parallelism due to limitations of its dependence/semantic analysis or transformation modules. In other cases, it is unable to generate good quality code because of limitations in

**Parallelization, Automatic. Table 2** An incomplete list of transformations for autoparallelization

| Name                | Description                                                                                                     | Example of use                                           |
|---------------------|-----------------------------------------------------------------------------------------------------------------|----------------------------------------------------------|
| Alignment           | Reorganizes computation so that values produced in one iteration are consumed by the same iteration             | Reduce synchronization costs                             |
| Distribution        | Partitions a loop into multiple loops                                                                           | Separates sequential from parallel parts                 |
| Fusion              | Merges two loops                                                                                                | Reduce parallel loop initiation overhead                 |
| Skewing             | Partitions the set of iterations into groups that are not related by dependences (i.e. are not ordered)         | Enhance parallelism                                      |
| Node Splitting      | Breaks a statement into two                                                                                     | Reduce dependence cycles and thus enable transformations |
| Software pipelining | Reorders and partitions the executions of operations in a loop into groups that are independent from each other | Enhance instruction level parallelism                    |
| Tiling              | Partitions the set of iterations of a multiply nested loop into blocks or tiles                                 | Enhance locality                                         |
| Trace scheduling    | Reorder and partition the executions of operations in a loop into groups that are independent from each other   | Enhance instruction level parallelism                    |
| Unroll and Jam      | Partitions the set of iterations of a multiply nested loop into blocks or tiles with reuse of values            | Enhance locality                                         |

its profitability analysis. That is, the compiler incorrectly assumes that transforming into parallel form would slow the program down.

To circumvent these limitations, compilers accept directives from programmers to help the analysis or guide the transformation and code generation process. A few vectorization directives for the Intel C++ compiler and IBM XLC compiler are shown in Table 3. The programmer can also influence the result by modifying the program into a form that can be recognized by the compiler.

Despite their limitations, autoparallelizers today contribute to productivity by

1. Saving labor. As mentioned, manual intervention in the form of directives or rewriting is typically necessary, but programmers can often rely on the autoparallelizing compiler for some sections of code and in some cases the whole program.

2. Portability. Sequential code complemented with directives is portable across classes of machines with the support of compilers. Portability is after all one of the purposes of compilers and autoparallelization brings this capability to the parallel realm.
3. As a training mechanism. Programmers can learn about what can and cannot be parallelized by interacting with an autoparallelizer. Thus, the compiler report to the programmer is not only useful for manual intervention, but also for learning.

## Future Directions

Autoparallelization has only been partially successful. As previously mentioned, in many cases today's compilers fail to recognize the existence of parallelism, or having recognized the parallelism, incorrectly assume that transforming into parallel form is not profitable. Although autoparallelization is useful and

**Parallelization, Automatic. Table 3** Vectorization directives for the IBM (XLC) and Intel (ICC) compilers

| Vectorization directive                                                          | Purpose                                                                                                                                            |
|----------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>#pragma vector always (ICC)</code>                                         | Vectorize the following loop whenever dependences allow it, disregarding profitability analysis                                                    |
| <code>#pragma nosimd (XLC)</code>                                                | Preclude vectorization of the following loop                                                                                                       |
| <code>#pragma novector (ICC)</code>                                              |                                                                                                                                                    |
| <code>_assume_aligned (A, 16); /ICC</code><br><code>_alignx(16, A); (XLC)</code> | The compiler is told to assume that the vector (A in the examples) start at addresses that are a multiple of a given constant (16 in the examples) |

effective when guided by user directives, there is clearly much room for improvement. Research in the area has decreased notably in the recent past, but it is likely that there will be more work in the area due to the renewed interest in parallelism that multicores have initiated. Two promising lines of future studies are

1. Empirical evaluation of compilers to improve parallelism detection, code generation, compiler feedback, and parallelization directives. Evaluating compilers using real applications is necessary to make advances in autoparallelization of conventional languages. Although there has been some work done in this area, much more needs to be done. This type of work is labor intensive since the best and perhaps the only way to do it is for an expert programmer to compare what the compiler does with the best code that the programmer can produce. This process is likely to converge since code patterns repeat across applications [8]. These costs and risks are worthwhile given the importance of the topic and the potential for an immense impact on productivity.
2. Study programming notations and their impact on autoparallelization. Higher level notations, such as those used for array operations, tend to facilitate the task of a compiler while at the same time improving productivity. Language-compiler codesign is an important and promising direction not only for autoparallelization but for compiler optimization in general.

## Related Entries

- [Banerjee's Dependence Test](#)
- [Code Generation](#)
- [Dependence Analysis](#)
- [Dependences](#)

- [GCD Test](#)
- [HPF \(High Performance Fortran\)](#)
- [Loop Nest Parallelization](#)
- [Modulo Scheduling and Loop Pipelining](#)
- [Omega Test](#)
- [Parallelization, Basic Block](#)
- [Run Time Parallelization](#)
- [Scheduling Algorithms](#)
- [Semantic Independence](#)
- [Speculative Parallelization of Loops](#)
- [Speculation, Thread-Level](#)
- [Trace Scheduling](#)
- [Unimodular Transformations](#)

## Bibliographic Notes And Further Reading

As mentioned in the introduction, work on autoparallelization started in the 1960s with the introduction of Illiac IV and the Texas Instrument Advanced Scientific Computer (ASC). The Paralyzer, an autoparallelizer for Illiac IV developed by Massachusetts Computer Associates, is discussed in [10].

This is the earliest description of a commercial autoparallelizer in the literature. Since then, there have been numerous papers and books describing commercial autoparallelizers. For example, [11] describes an IBM vectorizer of the 1980s, [3] discusses Intel's vectorizer for their multimedia extension, and [13] describes the IBM XLC compiler autoparallelization features.

Many of the autoparallelization techniques were developed at universities. Pioneering work was done by David Kuck and his students at the University of Illinois [6, 7]. The field has benefited from the contributions of numerous researchers. The contributions of Ken Kennedy and his coworkers [1] at Rice University have been particularly influential.

There have been only a few papers evaluating the effectiveness of autoparallelizers. In [8], different vectorizing compilers are compared using a collection of snippets, and in [4] the effectiveness of parallelizing compilers is discussed using the Perfect Benchmarks.

More information on autoparallelization, can be found in the related entries or in books devoted to this subject [2, 5, 12, 14]. Reference [5] contains a discussion of compiler techniques for High-Performance Fortran.

## Bibliography

1. Allen R, Kennedy K (1987) Automatic translation of FORTRAN programs to vector form. ACM Trans Program Lang Syst 9(4):491–542. DOI=<http://doi.acm.org/10.1145/29873.29875>
2. Banerjee UK (1997) Loop transformations for restructuring compilers: dependence analysis. Kluwer Academic, Norwell
3. Bik AJC (May 2004) The software vectorization handbook. Intel, Hillsboro
4. Eigenmann R, Hoeflinger J, Padua D (Jan 1998) On the automatic parallelization of the perfect Benchmarks®. IEEE Trans Parallel Distrib Syst 9(1):5–23. DOI=<http://dx.doi.org/10.1109/71.655238>
5. Kennedy K, Allen JR (2002) Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufmann, San Francisco
6. Kuck DJ (1976) Parallel processing of ordinary programs. Adv Comput 15:119–179
7. Kuck DJ, Kuhn RH, Padua DA, Leasure B, Wolfe M (1981) Dependence graphs and compiler optimizations. In: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on principles of programming languages. POPL '81. Williamsburg, 26–28 Jan 1981, ACM, New York, pp 207–218. DOI=<http://doi.acm.org/10.1145/567532.567555>
8. Levine D, Callahan, D, Dongarra, J (1991) A comparative study of automatic vectorizing compilers. Parallel Comput 17:1223–1244
9. Paek Y, Hoeflinger J, Padua D (Jan 2002) Efficient and precise array access analysis. ACM Trans Program Lang Syst 24(1):65–109. DOI=<http://doi.acm.org/10.1145/509705.509708>
10. Presberg DL (1975) The Paralyzer: Ivtran's parallelism analyzer and synthesizer. In: Proceedings of the conference on programming languages and compilers for parallel and vector machines, New York, 18–19 March 1975, pp 9–16. DOI=<http://doi.acm.org/10.1145/800026.808396>
11. Scarborough RG, Kolsky HG (March 1986) A vectorizing Fortran compiler. IBM J Res Dev 30(2):163–171
12. Wolfe M (1996) High performance compilers for parallel computing. Addison-Wesley, Reading
13. Zhang G, Unnikrishnan P, Ren J (2004) Experiments with auto-parallelizing SPEC2000FP benchmarks. LCPC, pp 348–362
14. Zima H, Chapman B (1991) Supercompilers for parallel and vector computers. ACM, New York

## Parallelization, Basic Block

UTPAL BANERJEE

University of California at Irvine, Irvine, CA, USA

## Synonyms

Parallelization

## Definition

A *basic block* in a program is a sequence of consecutive operations, such that control flow enters at the beginning and leaves at the end without halt. *Basic block parallelization* consists of techniques that allow execution of operations in a basic block in an overlapped manner without changing the final results.

## Discussion

### Introduction

The operations in a basic block are to be executed in the prescribed sequential order. This execution order imposes a *dependence structure* on the set of operations, based on how they access different memory locations. A new order of execution is *valid* if whenever an operation  $B$  depends on an operation  $A$  in the block, execution of  $B$  in the new order does not start until after the execution of  $A$  has ended. The basic assumption is that executing the operations in any valid order will not change the final results expected from the basic block.

To reduce the total execution time of the block, one needs to find a new valid order where operations are overlapped. Among all such orders, one must choose only those that are compatible with the physical resources of the given machine. Even when the simultaneous processing of two or more operations is permissible by dependence considerations alone, there may not be enough resources available to process them simultaneously.

There are many algorithms for basic block parallelization. This essay presents four of them: two for a hypothetical machine with unlimited resources, and two for a machine with limited resources. It starts with a section on basic concepts, and after developing the algorithms, ends with a simple example that compares

the actions of all four on a given basic block. References to more algorithms are given in the bibliographic notes.

## Basic Concepts

By an *operation* one means an atomic operation that a machine can perform. An *assignment operation* reads one or more memory locations and writes one location. It has the general form:

$$A : \quad x = E$$

where  $A$  is a label,  $x$  a variable, and  $E$  an expression. Such an operation reads the memory locations specified in  $E$ , and writes the location  $x$ . A *basic block* in a program is a sequence of assignment operations, where flow of control enters at the top and leaves at the bottom. There are no entry points except at the beginning, and no branches, except possibly at the end. The object of study in this essay is a basic block of  $n$  operations. The set of those operations is denoted by  $\mathcal{B}$ . There is a mapping  $c : \mathcal{B} \rightarrow \{0, 1, 2, \dots\}$  that gives the cycle times of the operations.

An operation  $B$  in the block *depends* on another operation  $A$ , and one writes  $A \delta B$ , if  $A$  is executed before  $B$ , and one of the following holds:

1.  $B$  reads the memory location written by  $A$ .
2.  $B$  writes a location read by  $A$ .
3.  $A$  and  $B$  both write the same location.

An operation  $B$  is *indirectly dependent* on an operation  $A$ , and one writes  $A \bar{\delta} B$ , if there exists a sequence of operations  $A_1, A_2, \dots, A_k$ , such that

$$A = A_1, A_1 \delta A_2, \dots, A_{k-1} \delta A_k, A_k = B.$$

Two operations  $A$  and  $B$  are *mutually independent* if  $A \bar{\delta} B$  and  $B \bar{\delta} A$  are both false. The *dependence graph* of the basic block is a directed acyclic graph, such that the nodes correspond to the operations, and there is an edge from a node  $A$  to a node  $B$  if and only if  $A \delta B$ . Thus,  $A \bar{\delta} B$  means there is a directed path from the node  $A$  to the node  $B$  in the dependence graph.

A new execution order for the operations in  $\mathcal{B}$  is *valid* if whenever  $A, B$  in  $\mathcal{B}$  are such that  $A \delta B$ ,  $B$  is executed after  $A$  in the new order. If the prescribed sequential order of execution for  $\mathcal{B}$  is changed to any valid order, the results would still be the same. The goal is to find a valid execution order where operations

are overlapped as much as possible. However, any such order must also be compatible with the resources of the given machine.

Control steps for execution of the basic block are numbered  $1, 2, 3, \dots$ . *Scheduling* an operation in the block means assigning a control step to it where it can start executing. An *instruction* for a given machine is a set of operations that the machine can perform simultaneously. An instruction may be empty. *Scheduling* the basic block means creating a sequence of  $m$  instructions  $(I_1, I_2, \dots, I_m)$ , where  $I_k$  starts in control step  $k$ , such that

1. Each instruction consists of operations in  $\mathcal{B}$ , and each operation in  $\mathcal{B}$  appears in exactly one instruction.
2. The operations in each instruction are pairwise mutually independent.
3. If an operation  $B$  in an instruction  $I_k$  depends on an operation  $A$  in an instruction  $I_j$ , then  $j + c(A) \leq k$ .
4. In any control step, the given machine has enough resources to process simultaneously all operations being executed.

Such a sequence of instructions is a *schedule* for the basic block. A schedule for the block can be specified indirectly by scheduling each individual operation. Then all operations starting in control step  $k$  constitute the instruction  $I_k$ .

The *weight* of a path  $(A_1, A_2, \dots, A_k)$  in the dependence graph for the basic block  $\mathcal{B}$  is the expression  $[c(A_1) + c(A_2) + \dots + c(A_k)]$ . A path is *critical* if it has the greatest possible weight among all paths in the graph. Let  $T_0$  denote the weight of a critical path. Then, any schedule for  $\mathcal{B}$  will need at least  $T_0$  cycles to finish.

For each operation  $A \in \mathcal{B}$ , the set of all immediate predecessors (in the dependence graph) is denoted by  $\text{Pred}(A)$  and the set of all immediate successors by  $\text{Succ}(A)$ :

$$\text{Pred}(A) = \{B \in \mathcal{B} : B \bar{\delta} A\}, \quad \text{Succ}(A) = \{B \in \mathcal{B} : A \bar{\delta} B\}.$$

The number of members of a set  $S$  is denoted by  $|S|$ .

## Unlimited Resources

In this section, it is assumed that the given machine has an unlimited supply of resources (functional units, registers, etc.). Consequently, operations in the basic block

can be scheduled subject only to the dependence constraints between them. The two algorithms considered in this section complete the execution of  $\mathcal{B}$  in exactly  $T_0$  cycles.

The ASAP algorithm schedules an operation *as soon as possible* so that the basic block can be processed in the shortest possible time. It creates a simple function  $\ell : \mathcal{B} \rightarrow \{1, 2, \dots\}$  such that  $\ell(A)$  is the earliest possible step when  $A$  can start executing. The ALAP algorithm schedules an operation *as late as possible* within the constraint of executing  $\mathcal{B}$  in the shortest possible time. It creates a function  $L : \mathcal{B} \rightarrow \{1, 2, \dots\}$  such that  $L(A)$  is the latest possible step when  $A$  can start executing. The *ASAP label* of  $A$  is  $\ell(A)$  and its *ALAP label* is  $L(A)$ . It is clear that  $\ell(A) \leq L(A)$  for each operation  $A$ . The range of consecutive integers from which the control step for  $A$  may be chosen is  $\{\ell(A), \ell(A) + 1, \dots, L(A)\}$ .

### ASAP Algorithm

The goal of the ASAP algorithm (Fig. 1) is to compute the ASAP label  $\ell$  for each operation in the given basic block  $\mathcal{B}$ . If  $A$  and  $B$  are two operations such that  $A \delta B$ , then after starting  $A$ , one must wait at least until  $A$  has finished before starting  $B$ . This means one must have  $\ell(B) \geq \ell(A) + c(A)$ .

**Algorithm 1** Given a basic block  $\mathcal{B}$ , its dependence graph, and a machine with unlimited resources, this algorithm computes the ASAP label  $\ell$  of each operation, and the total number  $m$  of instructions needed to replace  $\mathcal{B}$ . For each  $A \in \mathcal{B}$ , the sets  $\text{Pred}(A)$  and  $\text{Succ}(A)$  are assumed to be known.

```

 $m \leftarrow 1$
 $V \leftarrow \mathcal{B}$
for each operation $A \in V$ do
 $\text{Pcount}(A) \leftarrow |\text{Pred}(A)|$
 $\ell(A) \leftarrow 1$
endfor
while $V \neq \emptyset$ do
 for each operation $A \in V$ do
 if $\text{Pcount}(A) = 0$ then
 for each $B \in \text{Succ}(A)$ do
 $\text{Pcount}(B) \leftarrow \text{Pcount}(B) - 1$
 $\ell(B) \leftarrow \max\{\ell(B), \ell(A) + c(A)\}$
 endfor
 $V \leftarrow V - \{A\}$
 $m \leftarrow \max\{m, \ell(A)\}$
 endif
 endfor
endwhile
```

Parallelization, Basic Block. Fig. 1 The ASAP algorithm

An operation is scheduled only after all its predecessors have been scheduled. An integer-valued function  $\text{Pcount}$  on  $\mathcal{B}$  is defined as follows: at any point in the algorithm,  $\text{Pcount}(A)$  is the number of immediate predecessors of an operation  $A$  that have not been scheduled yet.

Initially, all operations are assigned the control step 1, that is,  $\ell(A)$  is initialized to 1 for each  $A \in \mathcal{B}$ . Operations  $A$  for which  $\text{Pcount}(A) = 0$  keep this value of  $\ell(A)$ ; they have been scheduled. If  $\text{Pcount}(A) = 0$  and  $B$  is a successor of  $A$ , then reduce  $\text{Pcount}(B)$  by 1, and increase  $\ell(B)$  to  $[\ell(A) + c(A)]$  if it is smaller. The operations whose  $\text{Pcount}$  is now zero keep their ASAP label; they have been scheduled. This process continues until all operations in  $\mathcal{B}$  have been scheduled.

The earliest control step where an operation  $B$  can start is given by

$$\ell(B) = \max_{A \in \text{Pred}(B)} [\ell(A) + c(A)].$$

The total number of cycles needed to execute the block  $\mathcal{B}$  is

$$\max_{A \in \mathcal{B}} [\ell(A) + c(A)] - 1$$

which is clearly equal to the weight  $T_0$  of a critical path in the dependence graph. The total number of instructions needed to replace the basic block is given by  $m = \max_{A \in \mathcal{B}} \ell(A)$ .

### ALAP Algorithm

The goal of the ALAP algorithm (Fig. 2) is to compute the ALAP label  $L$  for each operation in the given basic block  $\mathcal{B}$ . The entire block has to be completed in the shortest possible time in such a way that each operation starts as late as possible. For each operation  $A$ , let  $f(A)$  denote the number of cycles from the point when  $A$  is scheduled to start to the point when execution of the entire block has been completed. The idea then is to minimize  $f(A)$  for each  $A$ . When operation  $A$  starts,  $[L(A) - 1]$  cycles have already elapsed. Hence, the total number of cycles  $T$  needed to complete  $\mathcal{B}$  is  $[L(A) - 1 + f(A)]$ , so that

$$L(A) = T + 1 - f(A). \quad (1)$$

The ALAP algorithm first computes  $T$ , and  $f(A)$  for each  $A$ , and then evaluates  $L(A)$  from this equation. If  $A$  and  $B$  are two operations such that  $A \delta B$ , then after starting  $A$ , one must wait at least until  $A$  has finished

**Algorithm 2** Given a basic block  $\mathcal{B}$ , its dependence graph, and a machine with unlimited resources, this algorithm computes the ALAP label  $L$  of each operation, and the total number  $m$  of instructions needed to replace  $\mathcal{B}$ . For each  $A \in \mathcal{B}$ , the sets  $\text{Pred}(A)$  and  $\text{Succ}(A)$  are assumed to be known.

```

 $T \leftarrow 0$
 $V \leftarrow \mathcal{B}$
for each operation $A \in V$ do
 $\text{Scount}(A) \leftarrow |\text{Succ}(A)|$
 $f(A) \leftarrow 0$
endfor
while $V \neq \emptyset$ do
 for each operation $A \in V$ do
 if $\text{Scount}(A) = 0$ then
 $f(A) \leftarrow f(A) + c(A)$
 $T \leftarrow \max\{T, f(A)\}$
 for each $B \in \text{Pred}(A)$ do
 $\text{Scount}(B) \leftarrow \text{Scount}(B) - 1$
 $f(B) \leftarrow \max\{f(B), f(A)\}$
 endfor
 $V \leftarrow V - \{A\}$
 endif
 endfor
endwhile
for each operation $A \in V$ do
 $L(A) \leftarrow T + 1 - f(A)$
endfor
 $m \leftarrow \max_{A \in \mathcal{B}} L(A)$

```

Parallelization, Basic Block. Fig. 2 The ALAP algorithm

before starting  $B$ . This means  $L(B) \geq L(A) + c(A)$ , or  $f(A) \geq f(B) + c(A)$  by (1). Thus, the minimum possible value for  $f(A)$  is

$$f(A) = c(A) + \max_{B \in \text{Succ}(A)} f(B).$$

An operation is scheduled only after all its successors have been scheduled. An integer-valued function  $\text{Scount}$  on  $\mathcal{B}$  is defined as follows: at any point in the algorithm,  $\text{Scount}(B)$  is the number of immediate successors of an operation  $B$ , that have not been scheduled yet.

Initialize  $T$  to 0, and  $f(A)$  to 0 for each  $A \in \mathcal{B}$ . If an operation  $A$  has  $\text{Scount}(A) = 0$ , then  $f(A)$  is increased by  $c(A)$  to reach  $f(A) = 0 + c(A) = c(A)$ . This operation has now been scheduled. The value of  $T$  is also increased to  $f(A)$  if  $T < f(A)$ . If  $\text{Succ}(A) = 0$  and  $B$  is a predecessor of  $A$ , then reduce  $\text{Scount}(B)$  by 1, and increase  $f(B)$  to  $f(A)$  if  $f(B) < f(A)$ . The operations  $A$  for which  $\text{Scount}(A)$  is now zero are handled next. This process continues until all operations in  $\mathcal{B}$  have been

processed. When the final value of  $f(A)$  for each operation  $A$  and the final value of  $T$  are known, the ALAP labels are found from the equation  $L(A) = T + 1 - f(A)$ .

The total number of cycles needed to execute the block is  $T = \max_{A \in \mathcal{B}} f(A)$  which is equal to the weight  $T_0$  of a critical path in the dependence graph. The total number of instructions needed to replace the basic block is given by  $m = \max_{A \in \mathcal{B}} L(A)$ .

## Limited Resources

In this section, the reality is acknowledged that any given machine has limited amount of functional resources. While scheduling the operations in the basic block  $\mathcal{B}$ , one now needs to worry about the potential resource conflicts between two operations, in addition to the dependence constraints that may exist between them. For simplicity, a pipelined implementation is assumed for each multi-cycle operation. Register allocation is not treated here; it should be done either before or after scheduling.

## List Scheduling

*List Scheduling* employs a greedy approach to schedule as many operations as possible among those whose predecessors have been scheduled. Each operation is assigned a priority. Operations that are ready to be scheduled are placed on a *ready list* ordered by their priorities. At each control step, the operation with the highest priority is scheduled first. If there are two or more operations with the same priority, then a selection is made at random. List scheduling encompasses a family of different algorithms based on the choice of the priority function. In the algorithm described here (Fig. 3), the priority of an operation  $A$  is defined by the difference  $\mu(A) = L(A) - \ell(A)$  between its ALAP and ASAP labels, called the *mobility* of the operation. An operation with a lower mobility has a higher priority.

First, Algorithm 1 and Algorithm 2 are used to find the ASAP and ALAP labels of each operation in the given basic block. Operations without predecessors are placed on a ready list  $\mathcal{S}$  and arranged in the order of increasing mobility. They are taken from the ready list and scheduled one by one subject to the availability of machine resources. After one round, if there is still an operation  $A$  left over in  $\mathcal{S}$ , then its ASAP label is increased by 1 without exceeding its ALAP label. This will reduce the mobility of  $A$ , if it is not already zero.

**Algorithm 3** Given a basic block  $\mathcal{B}$ , its dependence graph, and a machine with limited resources, this algorithm finds a schedule for  $\mathcal{B}$ . For each  $A \in \mathcal{B}$ , the sets  $\text{Pred}(A)$  and  $\text{Succ}(A)$  are assumed to be known.

```

 $V \leftarrow \mathcal{B}$
 $k \leftarrow 1$
for each operation $A \in V$ do
 Compute the labels $\ell(A)$ and $L(A)$ by Algorithm 1 and Algorithm 2
endfor
while $V \neq \emptyset$ do
 $S \leftarrow$ all operations in V whose predecessors have finished
 executing before control step k
 for each $A \in S$ do
 $\mu(A) \leftarrow L(A) - \ell(A)$
 endfor
 Arrange the operations of S in a sequence $(A_{r_1}, A_{r_2}, \dots, A_{r_{|S|}})$
 in the increasing order of their mobilities
 Create an empty instruction \mathbf{I}_k
 for $i = 1$ to $|S|$ do
 if A_{r_i} does not have resource conflicts with operations in \mathbf{I}_k then
 Put A_{r_i} in \mathbf{I}_k
 $V \leftarrow V - \{A_{r_i}\}$
 $S \leftarrow S - \{A_{r_i}\}$
 endif
 endfor
 $k \leftarrow k + 1$
 for each $A \in S$ do
 $\ell(A) \leftarrow \min\{\ell(A) + 1, L(A)\}$
 endfor
endwhile

```

Parallelization, Basic Block. Fig. 3 List scheduling algorithm

### Linear Analysis

The Linear Algorithm (Fig. 4) checks the operations of the basic block *linearly* in their order of appearance, and puts them into instructions observing dependence constraints and avoiding resource conflicts.

Arrange the operations in the block in their prescribed sequential order:  $A_1, A_2, \dots, A_n$ . For an easy description of the algorithm, it is convenient to assume that a control step could be any integer. Start with a sequence of empty instructions  $\{\mathbf{I}_k : -\infty < k < \infty\}$  arranged in an imaginary vertical column. The operations  $A_1, A_2, \dots, A_n$  are taken in this order and put in instructions one by one. At any point, all operations already scheduled are in a range of instructions  $(\mathbf{I}_t, \mathbf{I}_{t+1}, \dots, \mathbf{I}_b)$ , where  $t \leq 1$  and  $b \geq 1$ . The value of  $t$  keeps decreasing and the value of  $b$  keeps increasing. At the end of the algorithm, the final sequence  $(\mathbf{I}_t, \mathbf{I}_{t+1}, \dots, \mathbf{I}_b)$  can be renumbered to get a sequence of instructions  $(\mathbf{I}'_1, \mathbf{I}'_2, \dots, \mathbf{I}'_m)$ , where  $m = b - t + 1$ .

Both  $t$  and  $b$  are initialized to 1. The first operation  $A_1$  is put in instruction  $\mathbf{I}_1$ . Suppose operations  $A_1, A_2, \dots, A_{i-1}$  have already been scheduled, and the time has come to schedule  $A_i$ , where  $2 \leq i \leq n$ . Let  $k$  denote the smallest integer  $\geq t$ , such that all predecessors of  $A_i$  finish executing before control step  $k$ . If  $A_i$  has no predecessors, then  $k = t$ . Otherwise, if a predecessor  $A_r$  is in an instruction  $\mathbf{I}_j$ , where  $t \leq j \leq b$ , then  $k \geq j + c(A_r)$ . The exact value of  $k$  is found by taking the maximum of all such expressions.

So, operation  $A_i$  can be put in instruction  $\mathbf{I}_k$  without violating any dependence constraints. If  $k \leq b$ , start checking the instructions  $\mathbf{I}_k, \mathbf{I}_{k+1}, \dots, \mathbf{I}_b$ , in this order, for potential resource conflicts. If an instruction is there in this range with which  $A_i$  does not conflict, then put  $A_i$  in the first such instruction. Otherwise,  $A_i$  conflicts with all instructions in the range and  $k = b + 1$ . If  $k > b$  was true before the checking could start, then its value did not change. At this point, if  $A_i$  had no predecessors

**Algorithm 4** Given a basic block  $(A_1, A_2, \dots, A_n)$  of  $n$  operations, its dependence graph, and a machine with limited resources, this algorithm finds a schedule for the block. At any point in the algorithm, all scheduled operations lie in the sequence of instructions  $(\mathbf{I}_t, \mathbf{I}_{t+1}, \dots, \mathbf{I}_b)$ , where  $t \leq 1$  and  $b \geq 1$ .

```

 $t \leftarrow 1$
 $b \leftarrow 1$
Put the operation A_1 in instruction \mathbf{I}_1
for $i = 2$ to n do
 $k \leftarrow$ earliest control step $\geq t$ before which all predecessors of A_i finish executing
 while $k \leq b$ and A_i has a resource conflict with operations in \mathbf{I}_k do
 $k \leftarrow k + 1$
 endwhile
 if $k \leq b$ then
 put A_i in \mathbf{I}_k
 else
 if $\text{Pred}(A_i) = \emptyset$ then
 Put A_i in \mathbf{I}_{t-1}
 $t \leftarrow t - 1$
 else
 Put A_i in \mathbf{I}_k
 $b \leftarrow k$
 endif
 endif
endfor

```

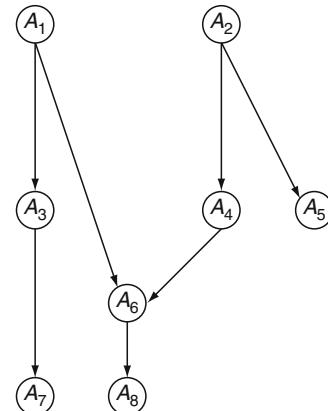
Parallelization, Basic Block. Fig. 4 The linear algorithm

in the first place, then put it at the top in instruction  $\mathbf{I}_{t-1}$  and decrease  $t$  to  $t - 1$ . If  $A_i$  had predecessors, then put it in instruction  $\mathbf{I}_k$  and increase  $b$  to  $k$ . The process ends when all operations in the given basic block have been scheduled.

### An Example

Consider a basic block  $\mathcal{B}$  consisting of eight operations  $A_1, A_2, \dots, A_8$  arranged in this order. The dependence graph of  $\mathcal{B}$  is given in Fig. 5. The type, cycle time, the immediate predecessors, and the immediate successors of each operation are listed in Table 1. It is assumed that the cycle time of an addition is 1, and that of a multiplication is 4. The four algorithms described in this essay are applied to  $\mathcal{B}$  one by one. The list scheduling and linear algorithms are customized for a machine with one adder and one multiplier. Note that the critical path in the dependence graph is  $A_2 \rightarrow A_4 \rightarrow A_6 \rightarrow A_8$ . Since its weight is  $T_0 = 10$ , the total number of cycles taken by each algorithm to process  $\mathcal{B}$  must be at least 10.

*ASAP Algorithm.* At the beginning, the ASAP label  $\ell(A_i)$  of each operation  $A_i$  is initialized to 1. Operations  $A_1$  and  $A_2$  have no predecessors. They keep their ASAP labels, that is, they are scheduled to start in control step 1. Since  $A_3$  and  $A_6$  are successors of  $A_1$ , and



Parallelization, Basic Block. Fig. 5 Dependence graph of basic block  $\mathcal{B}$

$A_4$  and  $A_5$  are successors of  $A_2$ , their ASAP values are increased as follows:

$$\begin{aligned}
 \ell(A_3) &\leftarrow \max\{\ell(A_3), \ell(A_1) + c(A_1)\} = \max\{1, 2\} = 2 \\
 \ell(A_6) &\leftarrow \max\{\ell(A_6), \ell(A_1) + c(A_1)\} = \max\{1, 2\} = 2 \\
 \ell(A_4) &\leftarrow \max\{\ell(A_4), \ell(A_2) + c(A_2)\} = \max\{1, 5\} = 5 \\
 \ell(A_5) &\leftarrow \max\{\ell(A_5), \ell(A_2) + c(A_2)\} = \max\{1, 5\} = 5.
 \end{aligned}$$

**Parallelization, Basic Block. Table 1** Details of the basic block of example 1

| OP A  | Type | c(A) | Pred(A)    | Succ(A)    | $\ell(A)$ | f(A) | L(A) | $\mu(A)$ |
|-------|------|------|------------|------------|-----------|------|------|----------|
| $A_1$ | +    | 1    |            | $A_3, A_6$ | 1         | 6    | 5    | 4        |
| $A_2$ | *    | 4    |            | $A_4, A_5$ | 1         | 10   | 1    | 0        |
| $A_3$ | +    | 1    | $A_1$      | $A_7$      | 2         | 2    | 9    | 7        |
| $A_4$ | +    | 1    | $A_2$      | $A_6$      | 5         | 6    | 5    | 0        |
| $A_5$ | +    | 1    | $A_2$      |            | 5         | 1    | 10   | 5        |
| $A_6$ | +    | 1    | $A_1, A_4$ | $A_8$      | 6         | 5    | 6    | 0        |
| $A_7$ | +    | 1    | $A_3$      |            | 3         | 1    | 10   | 7        |
| $A_8$ | *    | 4    | $A_6$      |            | 7         | 4    | 7    | 0        |

After  $A_1$  and  $A_2$  have been scheduled, operations  $A_3$ ,  $A_4$ , and  $A_5$  do not have any predecessors remaining to be scheduled. So, they can be scheduled in steps determined by their current ASAP values. Since  $A_7$  is a successor of  $A_3$  and  $A_6$  a successor of  $A_4$ , their ASAP values are increased as follows:

$$\begin{aligned}\ell(A_7) &\leftarrow \max\{\ell(A_7), \ell(A_3) + c(A_3)\} = \max\{1, 3\} = 3 \\ \ell(A_6) &\leftarrow \max\{\ell(A_6), \ell(A_4) + c(A_4)\} = \max\{2, 6\} = 6.\end{aligned}$$

After  $A_3$  and  $A_4$  have been scheduled, operations  $A_7$  and  $A_6$  do not have any predecessors remaining to be scheduled. So, they can be scheduled in steps determined by their current ASAP values. Since  $A_8$  is a successor of  $A_6$ , its ASAP value is increased as follows:

$$\ell(A_8) \leftarrow \max\{\ell(A_8), \ell(A_6) + c(A_6)\} = \max\{1, 7\} = 7.$$

After  $A_6$  has been scheduled, operation  $A_8$  does not have any predecessors remaining to be scheduled. So,  $A_8$  keeps its current ASAP value, and is scheduled to start in step 7. All the ASAP values are shown in Table 1.

The total number of control steps taken to execute  $\mathcal{B}$  is

$$\max_{1 \leq i \leq 8} [\ell(A_i) + c(A_i)] - 1 = 10.$$

The total number of instructions in the schedule is  $\max_i \ell(A_i) = 7$ . The instructions are  $(I_1, I_2, \dots, I_7)$ , where  $I_4$  is empty, and

$$\begin{aligned}I_1 &= \{A_1, A_2\}, I_2 = \{A_3\}, I_3 = \{A_7\}, \\ I_5 &= \{A_4, A_5\}, I_6 = \{A_6\}, I_7 = \{A_8\}.\end{aligned}$$

*ALAP Algorithm.* At the beginning,  $T$  is initialized to 0, and  $f(A_i)$  is initialized to 0 for each  $A_i$ . Since  $A_5$ ,  $A_7$ ,

and  $A_8$  have no successors,  $f(A_5)$ ,  $f(A_7)$ , and  $f(A_8)$  are increased as follows:

$$\begin{aligned}f(A_5) &\leftarrow f(A_5) + c(A_5) = 0 + 1 = 1 \\ f(A_7) &\leftarrow f(A_7) + c(A_7) = 0 + 1 = 1 \\ f(A_8) &\leftarrow f(A_8) + c(A_8) = 0 + 4 = 4.\end{aligned}$$

These operations have now been scheduled. The value of  $T$  is increased to 4. Since  $A_2$  is a predecessor of  $A_5$ ,  $A_3$  of  $A_7$ , and  $A_6$  of  $A_8$ , their  $f$  values are increased as follows:

$$\begin{aligned}f(A_2) &\leftarrow \max\{f(A_2), f(A_5)\} = \max\{0, 1\} = 1 \\ f(A_3) &\leftarrow \max\{f(A_3), f(A_7)\} = \max\{0, 1\} = 1 \\ f(A_6) &\leftarrow \max\{f(A_6), f(A_8)\} = \max\{0, 4\} = 4.\end{aligned}$$

Since  $A_3$  has no successors remaining to be scheduled,  $f(A_3)$  is changed as follows:

$$f(A_3) \leftarrow f(A_3) + c(A_3) = 1 + 1 = 2.$$

$A_3$  is now scheduled. The value of  $T$  remains unchanged at 4. Since  $A_1$  is a predecessor of  $A_3$ ,  $f(A_1)$  is increased as follows:

$$f(A_1) \leftarrow \max\{f(A_1), f(A_3)\} = \max\{0, 2\} = 2.$$

Since  $A_6$  has no successors remaining to be scheduled,  $f(A_6)$  is changed as follows:

$$f(A_6) \leftarrow f(A_6) + c(A_6) = 4 + 1 = 5.$$

$A_6$  is now scheduled. The value of  $T$  is increased from 4 to 5. Since  $A_1$  and  $A_4$  are predecessors of  $A_6$ , their  $f$  values are increased as follows:

$$\begin{aligned}f(A_1) &\leftarrow \max\{f(A_1), f(A_6)\} = \max\{2, 5\} = 5 \\ f(A_4) &\leftarrow \max\{f(A_4), f(A_6)\} = \max\{0, 5\} = 5.\end{aligned}$$

Now  $A_1$  and  $A_4$  do not have any successors remaining to be scheduled. Their  $f$  values are increased again as follows:

$$\begin{aligned}f(A_1) &\leftarrow f(A_1) + c(A_1) = 5 + 1 = 6 \\f(A_4) &\leftarrow f(A_4) + c(A_4) = 5 + 1 = 6.\end{aligned}$$

These two operations are now scheduled. The value of  $T$  is increased from 5 to 6. Since  $A_2$  is a predecessor of  $A_4$ ,  $f(A_2)$  is increased from 1 to  $f(A_4)$  or 6.

Since  $A_2$  now has no successors remaining to be scheduled,  $f(A_2)$  is increased again by  $c(A_2)$  to 6 + 4, or 10. The value of  $T$  is also increased from 6 to 10. Now that the value of  $T$  (total number of cycles), and the value of  $f(A_i)$  for each  $A_i$  are known, the ALAP labels are computed from the equation  $L(A) = T+1-f(A)$ . The values of  $f(A_i)$  and  $L(A_i)$  for each  $A_i$  are shown in Table 1.

The total number of instructions in the schedule is  $\max_i L(A_i) = 10$ . The instructions are  $(I_1, I_2, \dots, I_{10})$ , where  $I_2, I_3, I_4, I_8$  are empty, and

$$\begin{aligned}I_1 &= \{A_2\}, \quad I_5 = \{A_1, A_4\}, \quad I_6 = \{A_6\}, \\I_7 &= \{A_8\}, \quad I_9 = \{A_3\}, \quad I_{10} = \{A_5, A_7\}.\end{aligned}$$

*List Scheduling Algorithm.* Initially,  $V$  has all 8 operations. The initial value of the mobility of each operation is listed in Table 1. At step 1,  $S = \{A_1, A_2\}$ . The operations are arranged in the order  $(A_2, A_1)$ , since  $\mu(A_2) < \mu(A_1)$ . Both operations can be put in instruction  $I_1$ , since there is no resource conflict between them. At step 2,  $S = \{A_3\}$ . So,  $A_3$  goes into  $I_2$ . Similarly,  $A_7$  goes into  $I_3$ . At step 4,  $S = \emptyset$ . At step 5,  $S = \{A_4, A_5\}$ . The operations are arranged in this order, since  $\mu(A_4) < \mu(A_5)$ . Only  $A_4$  can be put into  $I_5$ . Take  $A_4$  out of  $S$  and decrease  $\mu(A_5)$  to 4. At step 6,  $S = \{A_5, A_6\}$ . The operations are arranged in the order  $(A_6, A_5)$ , since  $\mu(A_6) < \mu(A_5)$ . Only  $A_6$  can be put into  $I_6$ . Decrease  $\mu(A_5)$  to 3. At step 7,  $S = \{A_5, A_8\}$ . The operations are arranged in the order  $(A_8, A_5)$ , since  $\mu(A_8) < \mu(A_5)$ . Both operations can be put in instruction  $I_7$  since there is no resource conflict between them. The total number of instructions in the schedule for  $B$  is 7. Those instructions are  $(I_1, I_2, \dots, I_7)$ , where

$$\begin{aligned}I_1 &= \{A_1, A_2\}, \quad I_2 = \{A_3\}, \quad I_3 = \{A_7\}, \quad I_4 = \emptyset, \\I_5 &= \{A_4\}, \quad I_6 = \{A_6\}, \quad I_7 = \{A_5, A_8\}.\end{aligned}$$

Total number of cycles =  $[7 + \max\{c(A_5), c(A_8)\} - 1] = 10$ .

*Linear Algorithm.* Initially,  $t = b = 1$ . Put operation  $A_1$  in instruction  $I_1$ . Since  $A_2$  has no predecessors and does not conflict with  $A_1$ , put  $A_2$  also in  $I_1$ . Operation  $A_3$  has only one predecessor, namely,  $A_1$ . Since  $c(A_1) = 1$ ,  $A_3$  can be put in instruction  $I_2$ . Increase  $b$  to 2. The sole predecessor of  $A_4$  is  $A_2$  and  $c(A_2) = 4$ . Hence,  $A_4$  can go into instruction  $I_5$ . Increase  $b$  to 5. Operation  $A_5$  also has  $A_2$  as its only predecessor. But,  $A_5$  cannot go into  $I_5$ , since it conflicts with  $A_4$ . So, put  $A_5$  in  $I_6$  and increase  $b$  to 6. Operation  $A_6$  has two predecessors:  $A_1$  and  $A_4$ . The earliest instruction that can take  $A_6$  without violating dependence constraints is  $I_6$ . But  $I_6$  already has  $A_5$  and it conflicts with  $A_6$ . So, put  $A_6$  in  $I_7$  and increase  $b$  to 7. Now it is easy to see that  $A_7$  can go into  $I_3$  and  $A_8$  into  $I_8$ . Increase  $b$  to 8.

The total number of instructions in the schedule for  $B$  is 8. Those instructions are  $(I_1, I_2, \dots, I_8)$ , where

$$\begin{aligned}I_1 &= \{A_1, A_2\}, \quad I_2 = \{A_3\}, \quad I_3 = \{A_7\}, \quad I_4 = \emptyset, \\I_5 &= \{A_4\}, \quad I_6 = \{A_5\}, \quad I_7 = \{A_6\}, \quad I_8 = \{A_8\}.\end{aligned}$$

Total number of cycles =  $[8 + c(A_8) - 1] = 11$ .

## Related Entries

- [Code Generation](#)
- [Loop Nest Parallelization](#)
- [Parallelization, Automatic](#)
- [Unimodular Transformations](#)

## Bibliographic Notes and Further Reading

Basic block parallelization was first studied in the context of microprogramming. The book by Agerwala and Rauscher [1] gives a good introduction to microprogramming. The theory of job scheduling in operations research turned out to be a rich source of algorithms for the microprogramming research community [3, 4, 7]. For linear analysis and list scheduling as covered in this entry, a good place to start is the paper by Landskov and others [8]. (The algorithms presented above try to factor in explicitly the cycle times of operations.) Other standard references are [2, 5, 6, 10].

Only two algorithms are given in this entry for the limited resource case; there are many more. For example, for Force-directed Scheduling, see [9].

The ten references listed here constitute a small percentage of what is available.

## Bibliography

1. Agerwala AK, Rauscher TG (1976) Foundations of microprogramming architecture, software, and applications. Academic, New York
2. Agerwala T (Oct 1976) Microprogram optimization: a survey. *IEEE Trans Comput C-25(10)*:962–973
3. Coffman EG Jr (1976) Computer and job-shop scheduling theory. Wiley, New York
4. Conway RW, Maxwell WL, Miller LW (1967) Theory of scheduling. Addison-Wesley, Reading
5. Dasgupta S, Tartar J (Oct 1976) The identification of maximal parallelism in straightline microprograms. *IEEE Trans Comput C-25(10)*:986–992
6. Gonzalez MJ Jr (Sep 1977) Deterministic processor scheduling. *ACM Comput Surv 9(3)*:173–204
7. Hu TC (Nov-Dec 1961) Parallel sequencing and assembly line problems. *Oper Res 9(6)*:841–848
8. Landskov D, Davidson S, Shriver B, Mallett PW (Sep 1980) Local microcode compaction techniques. *Comput Surv 12(3)*:261–294
9. Paulin PG, Knight JP (June 1989) Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Trans CAD Integ Circ Syst 8(6)*:661–679
10. Ramamoorthy CV, Chandy KM, Gonzalez MJ (Feb 1972) Optimal scheduling strategies in a multiprocessor system. *IEEE Trans Comput C-21(2)*:137–146

## Parallelization, Loop Nest

### ► Loop Nest Parallelization

## ParaMETIS

### ► METIS and ParMETIS

## PARDISO

OLAF SCHENK<sup>1</sup>, KLAUS GÄRTNER<sup>2</sup>

<sup>1</sup>University of Basel, Basel, Switzerland

<sup>2</sup>Weierstrass Institute for Applied Analysis and Stochastics, Berlin, Germany

## Definition

PARDISO, short for “PARallel DIrect SOLver,” is a thread-safe software library for the solution of large

sparse linear systems of equations on shared-memory multicore architectures. It is written in Fortran and C and it is available at [www.pardiso-project.org](http://www.pardiso-project.org). The solver implements an efficient supernodal method, which is a version of Gaussian elimination for large sparse systems of equations, especially those arising, for example, from the finite element method or in nonlinear optimization. It is the only sparse solver package that supports all kinds of matrices such as complex, real, symmetric, nonsymmetric, or indefinite. PARDISO can be called from various environments including MATLAB (via MEX), Python (via pypardiso), C/C++, and Fortran. PARDISO version 4.0.0 was released in October 2009.

## Discussion

### Introduction

The solution of large sparse linear systems lies at the heart of many calculations in computational science and engineering and is also of increasing importance in computations in the medical imaging and financial sectors. Today, systems of equations with millions to hundreds of millions of unknowns are solved. To do this within reasonable time requires efficient use of powerful parallel computers and advanced combinatorial and numerical algorithms based on direct or approximate direct factorizations. To date, only very limited software for such large systems is generally available. The PARDISO software addresses this issue.

In this chapter, some important combinatorial aspects and main algorithmic features for solving sparse systems will be reviewed. The algorithmic improvements of the past 20 years have reduced the time required to factor general sparse matrices by almost three orders of magnitude. Combined with significant advances in the performance to cost ratio of computing hardware during this period, current sparse solver technology makes it possible to solve those problems quickly and easily, which might have been considered by far too large until recently. This chapter discusses the basic and the latest developments for sparse direct solution methods that have lead to modern *LU* decomposition techniques.

The PARDISO development started in the context of a PhD Project [12] at ETH Zurich in Switzerland.

The first aim was to improve parallel sparse factorization methods for highly ill-conditioned matrices arising in the semiconductor device simulation area. The close collaboration with the simulation community resulted in a large amount of test cases and industrial use of the solver.

The library version of PARDISO is publicly released since March 2004. Ongoing research projects resulted in the current version 4.0.0, available since October 2009. This new version also includes novel state-of-the-art incomplete factorization-type preconditioners. The results of the research have appeared in several scientific journals, including the paper “On Large Scale Diagonalization Techniques for the Anderson Model of Localization” in the SIGEST section of the SIAM Review Journal [13].

It is the purpose of this chapter to describe the main combinatorial and numerical algorithms used in an efficient parallel solver.

### Sparse Gaussian Elimination in PARDISO

To introduce the notations, the LU-factorization of a nonsymmetric matrix  $A$  with pivoting is described. A simple description of the algorithm for solving sparse linear equations by sparse Gaussian elimination in PARDISO is as follows:

- Compute the triangular factorization  $P_r D_r P_f A P_f^T = L U$ . Here  $D_r$  and  $D_c$  are diagonal matrices to equilibrate the system,  $P_f$  is a permutation matrix that minimizes the number of nonzeros in the factor, and  $P_r$  and  $P_c$  are permutation matrices. Premultiplying  $A$  by  $P_r$  reorders the rows of  $A$ , and premultiplying  $A$  by  $P_c$  reorders the columns of  $A$ .  $P_r$  and  $P_c$  are chosen to enhance sparsity, numerical stability, and parallelism.  $L$  is a unit lower triangular matrix with  $L_{ii} = 1$  and  $U$  is an upper triangular matrix.
- Solve  $AX = B$  by evaluating

$$X = A^{-1}B = (P_f^{-1}D_r^{-1}P_r^{-1}LUP_c^{-1}D_c^{-1}P_f^{-T})^{-1}B,$$

$$X = P_f^T D_c P_c U^{-1} L^{-1} P_r D_r P_f B.$$

This is done efficiently by multiplying from right to left in the last expression: the rows of  $B$  are permuted with  $P_f$  and scaled by  $D_r$ . Multiplying  $P_r B$  means permuting the rows of  $D_r B$ . Multiplying  $L^{-1}(P_r D_r B)$  means solving triangular systems of equations with

$n_r$  right-hand sides with matrix  $L$  by substitution. Similarly, multiplying  $U^{-1}(L^{-1}(P_r D_r P_f B))$  means solving triangular systems with  $U$ .

For symmetric indefinite matrices, PARDISO performs a numerical factorization into  $LDL^T$ , in which the factorization is also stabilized by extended numerical pivoting techniques. In addition to the complete factorization, advanced support for incomplete inverse-based factorization preconditioners, in which the factors are computed approximately, are also included in PARDISO.

The efficient computation of the  $LU$  decomposition is of utmost importance in Gaussian elimination. Typically, a compact representation of the elimination tree can be used to derive all information concerning fill-in and numerical dependencies. In particular, the fact that pivoting and the factorization must be interlaced requires a completely different treatment than in the case without pivoting. Consider the situation when computing the  $LU$  decomposition column by column.

---

for  $k = 1, \dots, n$

update column  $k$  of  $L$  and  $U$  via the equation  $A_{1:n,k}$   
 $= L_{1:n,1:k-1} U_{1:k-1,k} - L_{1:n,k} u_{kk}$

1. step. solve  $L_{1:k-1,1:k-1} U_{1:k-1,k} = A_{1:k-1,k}$
2. step.  $L_{k:n,k} := A_{k:n,k}$   
 for  $i < k$  such that  $u_{ik} \neq 0$   
 $L_{k:n,k} := L_{k:n,k} - L_{k:n,i} u_{ik}$
3. step.  $u_{kk} = l_{kk}$   
 $L_{k:n,k} = L_{k:n,k} / u_{kk}$

---

Note that it is easy to add pivoting after step 2 by interchanging  $l_{kk}$  with some sufficiently large  $|l_{mk}|$ , where  $m \geq k$  before  $u_{kk}$  is defined. The art of efficiently computing column  $k$  of  $L$  and  $U$  consists of how the sparse forward solve  $L_{1:k-1,1:k-1} U_{1:k-1,k} = A_{1:k-1,k}$  in step 1 is efficiently implemented and, a fast update of  $L_{k:n,k}$  in step 2.

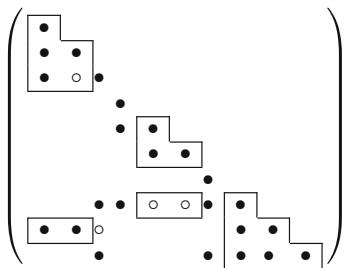
The dense  $LU$  decomposition as it is described so far can be implemented in a way that it makes heavily use of level-3 BLAS. One important aspect that allows raising efficiency and speeding up the sparse numerical factorization will be discussed now. It is the recognition

of an underlying block structure with dense submatrices caused by the factorization and the fill. The block structure allows to collect parts of the matrix in dense blocks and to treat them commonly using higher levels of BLAS. As a consequence of the LU decomposition, parts of the triangular factors can be encountered that are dense or become dense by the factorization. This key structure in sparse Gaussian elimination is based on supernodal representation of the columns and rows in the factors  $L$  and  $U$ . A sequence  $k, k+1, \dots, k+s-1$  of  $s$  subsequent columns in  $L$  that form a dense lower triangular matrix is called a supernode.

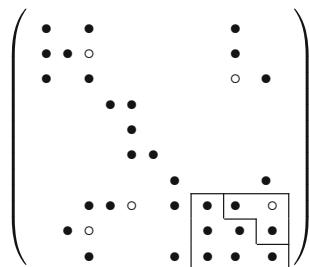
To illustrate the use of supernodes, Figs. 1 and 2 illustrate the underlying dense block structure.

Supernodes in PARDISO are stored in rectangular dense matrices. Beside the storage scheme as dense matrices, the nonzero row indices for these blocks need only be stored once. Next, the use of dense submatrices allows the usage of level-3 BLAS routines. To understand this, one can easily verify that the update process

$$L_{k:n,k} := L_{k:n,k} - L_{k:n,i} u_{ik}$$



**PARDISO. Fig. 1** Supernodes in  $L$ , symmetric case  
("○" denotes fill-in)



**PARDISO. Fig. 2** Supernodes in  $L + U$ , general case  
("○" denotes fill-in)

that computes  $L_{k:n,k}$  in the algorithm can easily be extended to the block case. Assuming that columns  $1, \dots, k+s-1$  are collected in  $p$  supernodes with column indices  $\mathcal{K}_1, \dots, \mathcal{K}_p$ . Apparently, the column set is  $\mathcal{K}_p = \{1, \dots, k+s-1\}$ . Then updating  $L_{k:n,\mathcal{K}_p}$  can be rewritten as

$$L_{k:n,\mathcal{K}_p} := L_{k:n,\mathcal{K}_p} - L_{k:n,i} U_{\mathcal{K}_i, \mathcal{K}_p},$$

where the sum has to be taken over all  $i$  in the block version of the elimination tree. Depending on whether one would like to compute the diagonal block  $L_{\mathcal{K}_p, \mathcal{K}_p}$  as full or as lower triangular matrix one is able to use level-2 BLAS or even level-3 BLAS subroutines. This allows exploiting machine-specific properties, such as caches to accelerate the computation.

As discussed above, dynamic pivoting has been a central tool by which nonsymmetric sparse linear solvers gain stability. Therefore, improvements in speeding up direct factorization methods were limited to the uncertainties that have arisen from using pivoting. Certain techniques, like the column elimination tree [2], have been useful for predicting the sparsity pattern despite pivoting. However, in the symmetric case, the situation becomes more complicated since only symmetric reorderings, applied to both columns and rows, are required, and no a priori choice of pivots is given. This makes it almost impossible to predict the elimination tree in a sensible manner, and the use of cache-oriented level-3 BLAS is impossible.

With the introduction of symmetric maximum weighted matchings [16] as an alternative to complete pivoting [3], it is now possible to treat symmetric indefinite systems similarly to symmetric positive definite systems. This allows to predict fill using the elimination tree, and thus allows to set up the data structures that are required to predict dense submatrices (also known as supernodes). This in turn means that one is able to exploit level-3 BLAS applied to the supernodes. Consequently, the classical Bunch–Kaufman pivoting approach needs to be performed only inside the supernodes.

This approach has recently been successfully implemented in symmetric indefinite version of PARDISO [16]. As a major consequence of this novel approach, the sparse indefinite solver has been improved to become almost as efficient as its symmetric positive counterpart.

Finally, iterative refinement is the last option to test and enhance the precision of the solution. To encourage the use of  $L$  and  $U$  as preconditioner in solving continuously dependent families of problems, PARDISO also offers a CG and CGS branch.

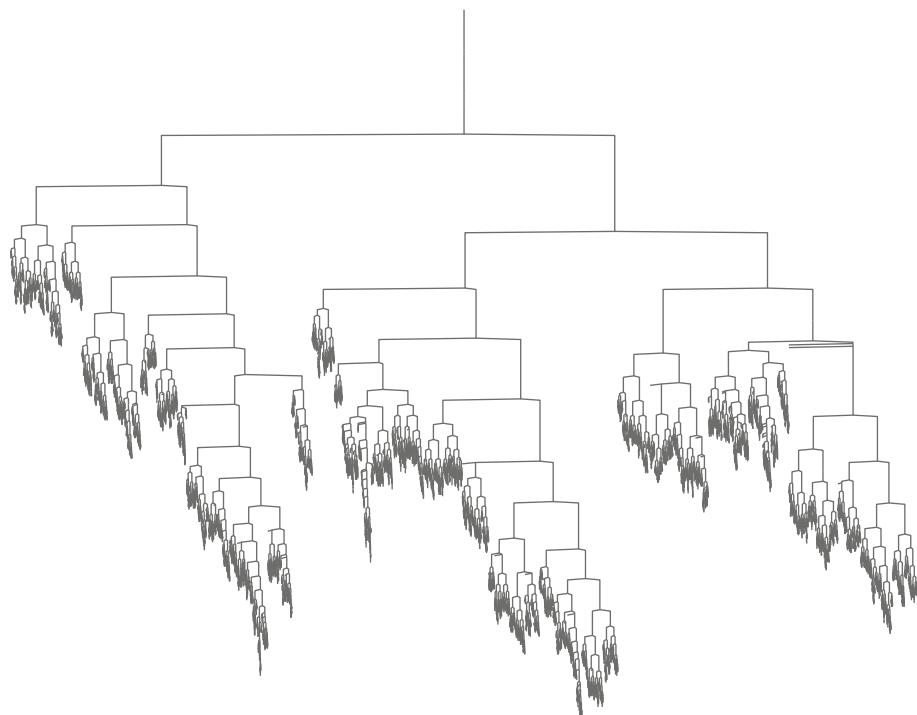
## Reordering Algorithms and Software in PARDISO

The efficiency of a direct solver depends strongly on the order in which the variables of the matrix are eliminated. This largely determines the computational amount of work executed during the numerical factorization and hence the overall time for the solution. Furthermore, the elimination order of the variables also determines the number of entries in the computed factors and the amount of main storage required. Experiments have shown that the multiple minimum degree (MMD) algorithm of J. Liu [8] is very well suited for 2D problems, but nested dissection type orderings do a much better job on larger problems from 3D discretizations. Thus, the ordering METIS package from Karypis and Kumar [7] and a constrained minimum fill-in orderings developed by Schenk and Gärtner [12] have

been added to PARDISO. In a somewhat more loosely coupled way, orderings from other packages such as approximate minimum degree [1] can also be used.

## Parallelization Strategies in PARDISO

Parallelism can be expressed by the elimination tree: different branches are independent (see Fig. 3). The length of each vertical line is proportional to the size of a dense diagonal block. The height counted in levels characterizes the longest chain of sequentially depending steps. Memory writes should be minimized and asynchronous scheduling is possible in PARDISO on SMP and NUMA architectures. It is used to reduce load balance requirements. Hence, the small amount of synchronization data is passed towards the root (or to the right in  $L$ ,  $U$ ), while the numerical data is read from the left of the actual supernode. Especially large supernodes close to the root of the elimination tree are split into panels to increase the number of parallel tasks. PARDISO uses METIS by default to generate nested dissection permutations and maps all sparse computations on dense matrix operations to achieve level-3 BLAS performance for the numerical factorization. Scheduling



PARDISO. Fig. 3 Typical elimination tree with a MMD reordering

distinguishes two levels in the elimination tree to reduce synchronization events: updates within the lower level are conflict free, hence must not be synchronized. In PARDISO, complete lower level subtrees of the elimination are mapped onto a single core of the multiprocessing architecture. The columns and rows associated with this complete subtree are factorized independently with respect to other subtrees. The nodes near the root of the elimination tree normally involve more computation than supernodes further away from the root. In practical examples more than 75% of the floating-point computations are performed on the supernodes close to the root of the tree. Unfortunately, the number of independent supernodes near the root of the tree is small, and so there is less parallelism to exploit. For example, the root in Fig. 3 has only two neighboring nodes and hence only two processors would perform the corresponding work while the other processors remain idle. The introduction of the panels and the execution of partial updates as tasks for supernodes close to the root increases the parallelism.

An additional constraint that has to be taken into account is that PARDISO is designed to solve a wide range of application problems including symmetric and nonsymmetric matrices, and numerical pivoting is performed within the numerical factorization. This means that only a static analysis of the sparsity pattern and a static allocation of supernodes to cores could be very inefficient if numerical pivoting is required. As a solution, a novel left-right looking factorization has been implemented that uses a dynamic allocation of threads during the numerical factorization. This has the benefit of enabling the linear scalability of the code to perform well on multicore architectures with up to 16 cores. The parallel execution in PARDISO does not change the constant in the leading order of the complexity bound compared with that of the sequential algorithm.

### Approximate Sparse Gaussian Factorization in PARDISO

The solver also includes a novel preconditioning solver [13]. The preconditioning approach for symmetric indefinite linear system is based on maximum weighted matchings and algebraic multilevel incomplete  $LDL^T$  factorization. These techniques can be seen as a complement to the alternative idea of using more complete

pivoting techniques for the highly ill-conditioned symmetric indefinite matrices. In considering how to solve the linear systems in a manner that exploits sparsity as well as symmetry, a diverse range of algorithms is used that includes preprocessing the matrix with symmetric weighted matchings, solving the linear system with Krylov subspace methods, and accelerating the linear system solution with multilevel preconditioners based upon incomplete inverse-based factorizations.

### General Software Issues in PARDISO

The PARDISO package was originally designed for structurally symmetric matrices arising in the semiconductor device simulation area, but, in the newer version, all other types of matrices such as real and complex symmetric, real or complex nonsymmetric systems, or complex Hermitian systems are permitted. The PARDISO software is written in C and Fortran. However, in recognition that some users prefer a Matlab or Python programming environment, an appropriate interface has been developed for these programming languages.

It requires OpenMP threading capabilities and makes heavy use of level-3 BLAS and LAPACK subroutines. For the parallel version of PARDISO, a single-threaded version of level-3 BLAS and LAPACK routines is used. PARDISO has been ported to a wide range of computers from previous generations of vector-computers such as Cray or NEC, to all kind of multicore architectures including Intel, AMD, IBM, SUN, and SGI [15] and also to recent throughput manycore processors such as GPUs from NVIDIA [14].

The PARDISO package has an excellent performance relative to other parallel sparse solvers. An extensive evaluation of the performance of the numerical factorization in comparison to a wide range of other sparse direct linear solver packages is given by Gould, Hu, and Scott [4]. Other independent comparisons can be found in [5]. PARDISO is currently heavily used in linear and nonlinear optimization solvers. Recent results can be found in [17]. The current version of the solver and a manual including several examples is available at [www.pardiso-project.org](http://www.pardiso-project.org).

### Example

The example illustrates the complexity gap between numerical factorization and solution in the sparse direct

solution process. As shown in [Table 1](#), the numerical factorization is the most time-consuming phase. This is demonstrated on a 3D eigenvalue problem. The symmetric eigenvalue problem  $Ax = \lambda Bx$  is chosen with up to  $3.5 \times 10^6$  unknowns to be close to the size limits of a today's 128 GB SMP machines.

The eigenvalue problem is solved by inverse iteration, polynomial acceleration, a few different spectral shifts, and the same number of factorizations. On average, five eigenvalues are computed per factorization. The spectral shifts can be chosen to result in

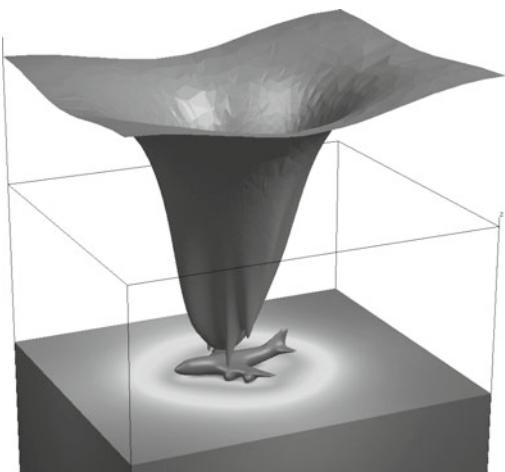
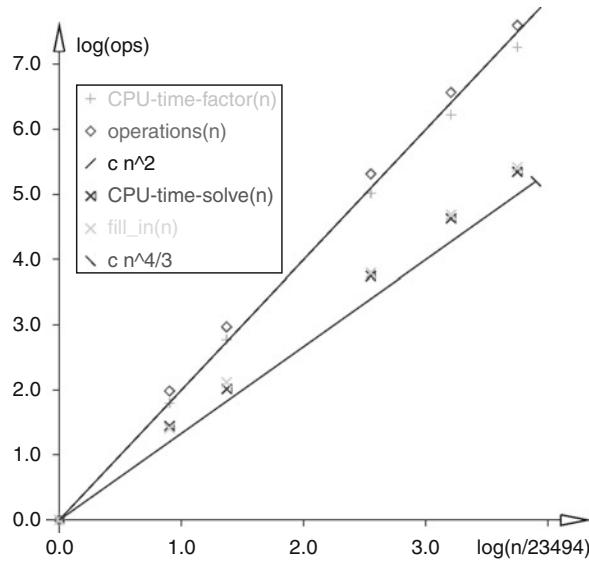
**PARDISO. Table 1** The serial computational complexity of the various phases of solving a sparse system of linear equations arising from 2D and 3D constant node-degree graphs with  $n$  vertices

| Phase                   | Dense    | 2D complexity | 3D complexity |
|-------------------------|----------|---------------|---------------|
| Reordering:             | –        | $O(n)$        | $O(n)$        |
| Symbolic factorization  | –        | $O(n \log n)$ | $O(n^{4/3})$  |
| Numerical factorization | $O(n^3)$ | $O(n^{3/2})$  | $O(n^2)$      |
| Triangular solution     | $O(n^2)$ | $O(n \log n)$ | $O(n^{4/3})$  |

highly ill-conditioned but sufficiently regular sparse linear systems. The symmetric indefinite linear systems are solved by the Bunch-Kaufman pivoting. The quadratic complexity of the sparse direct factorization is shown in [Fig. 4](#).

## Future Research Directions

Clearly, the quadratic complexity bound is limiting the problem size in 3D for sparse direct methods. However, from the user perspective, the robustness and the nearly achieved black box behavior are very convenient. The emergence of multicore architectures and scalable petascale architectures does not change both points. Instead, it motivates the development of novel algorithms and techniques that emphasize both concurrency and robustness for the solution of sparse linear systems. Since direct solvers do not generally scale to large problems and machine configurations, efficient application of preconditioned iterative solvers are warranted but involve more a priori knowledge. These hybrid solvers must optimize parallel performance, processor (serial) performance, as well as memory requirements, while being robust across specific classes of applications and systems.



**PARDISO. Fig. 4** Unstructured tetrahedral meshes with  $n = 23'494, 57'869, 92'328, 300'608, 579'150$  and  $996'557$  tetrahedra for a discrete Laplace problem. The left graphic shows the measured times for the numerical factorization and the solution, the number of floating-point operations, and the complexity bounds. The first eigenvector is shown in the right graphic. The selfsimilar tetrahedral meshes are generated by TetGen, <http://www.tetgen.org>

The research groups at Purdue University and University of Basel are currently developing a new parallel solver PSPIKE [10] that combines the desirable characteristics of direct methods (robustness) and effective iterative solvers (low computational cost), while alleviating their drawbacks (limited scalability, memory requirements, lack of robustness). The hybrid solver is based on the general sparse solver PARDISO and the Spike family of hybrid solvers [11]. The resulting PSPIKE algorithm is an extension of PARDISO to distributed-memory architectures. Results can be found in, for example, [10].

## Related Entries

- [BLAS \(Basic Linear Algebra Subprograms\)](#)
- [Graph Partitioning](#)
- [Load Balancing, Distributed Memory](#)
- [Linear Algebra Software](#)
- [Nonuniform Memory Access \(NUMA\) Machines](#)
- [Shared-Memory Multiprocessors](#)
- [LAPACK](#)

## Further Reading

1. Davis T (2006) Direct methods for sparse linear systems. Society for industrial mathematics, ISBN:0898716136

## Bibliography

1. Amestoy R, Davis TA, Duff IS (1996) An approximate minimum degree ordering algorithm. *SIAM J Matrix Anal Appl* 17:886–905
2. Demmel JW, Eisenstat SC, Gilbert JR, Li XS, Liu JWH (1999) A supernodal approach to sparse partial pivoting. *SIAM J Matrix Anal Appl* 20:720–755
3. Duff IS, Koster J (1999) The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J Matrix Anal Appl* 20(4):889–901
4. Gould NIM, Hu Y, Scott JA (2007) A numerical evaluation of sparse direct solvers for the solution of large sparse, symmetric linear systems of equations. *ACM Trans Math Software (TOMS)* 33(2):1–32
5. Grasedyck L, Hackbusch W, Kriemann R (2008) Performance of H-LU preconditioning for sparse matrices. *Comput Methods Appl Math* 8(4):336–349
6. Hagemann M, Schenk O (2006) Weighted matchings for preconditioning of symmetric indefinite linear systems. *SIAM J Sci Comput* 28:403–420
7. Karypis G, Kumar V (1998) A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J Sci Comput* 20:359–392
8. Liu J (1985) Modification of the minimum-degree algorithm by multiple elimination. *ACM Trans Math Software* 11(2):141–153

9. Ng E, Peyton B (1993) Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM J Sci Comput* 14: 1034–1056
10. Manguoglu M, Sameh A, Schenk O (2009) PSPIKE Parallel sparse linear system solver. In: Proceedings of the 15th international Euro-Par conference on parallel processing. Lecture Notes in Computer Science, vol 5704, pp 797–808, DOI 10.1007/978-3-642-03869-3 (vol 14, pp 1034–1056)
11. Polizzetti E, Sameh AH (2006) A parallel hybrid banded system solver: the SPIKE algorithm. *Parallel Comput* 32(2):177–194
12. Schenk O (2000) Scalable parallel sparse LU factorization methods on shared memory multiprocessors. PhD thesis, ETH Zürich
13. Schenk O, Bollhöfer M, Römer RA (2008) On large-scale diagonalization techniques for the Anderson model of localization. *SIAM Rev* 50:91–112
14. Schenk O, Christen M, Burkhart H (2008) Algorithmic performance studies on graphics processing unit. *J Parallel Distrib Comput* 28:1360–1369
15. Schenk O, Gärtner K (2004) Solving unsymmetric sparse systems of linear equations with PARDISO. *J Future Gener Comput Syst* 20(3):475–487
16. Schenk O, Gärtner K (2006) On fast factorization pivoting methods for symmetric indefinite systems. *Electron Trans Numer Anal* 23:158–179
17. Schenk O, Wächter A, Hagemann M (2007) Matching-based preprocessing algorithms to the solution of saddle-point problems in large-scale nonconvex interior-point optimization. *Comput Optim Appl* 36(2–3):321–341

## PARSEC Benchmarks

The PARSEC benchmarks [1] are multithreaded codes that represent important applications of multicores. PARSEC stands for Princeton Application Repository for Shared-Memory Computers. Currently, the benchmark suite contains 13 programs: blackscholes, bodytrack, canneal, dedup, facesim, ferret, fluidanimate, freqmine, raytrace, streamcluster, swaptions, vips, and x264.

## Bibliography

1. Bienia C (2011) Benchmarking modern multiprocessors. Ph.D. thesis, Princeton University

## Partial Computation

- [Trace Theory](#)

## Particle Dynamics

- ▶ [N-Body Computational Methods](#)

## Particle Methods

- ▶ [N-Body Computational Methods](#)

## Partitioned Global Address Space (PGAS) Languages

- ▶ [PGAS \(Partitioned Global Address Space\) Languages](#)

## PASM Parallel Processing System

HOWARD JAY SIEGEL, BOBBY DALTON YOUNG  
Colorado State University, Fort Collins, CO, USA

### Definition

PASM was a partitionable mixed-mode parallel system designed and prototyped in the 1980s at Purdue University to study three dimensions of dynamic reorganization: mixed-mode parallelism, partitionability, and flexible interprocessor communications.

### Discussion

#### Introduction

PASM was a partitionable mixed-mode parallel system designed and prototyped in the 1980s at Purdue University [20, 21]. Research was conducted about numerous software, hardware, parallel algorithm, and application aspects of PASM.

In the 1980s, two of the dominant organizations for parallel machines were “SIMD” and “MIMD” [8]. Be forewarned that our use of the term “SIMD” is more general than the way it is used with current multi-core systems, where it refers to operating on subfields of a long data word within a single processor. We will use the following definition: An SIMD (single instruction stream, multiple data stream) machine consists

of N processor-memory pairs, an interconnection network, and a control unit. The *control unit* broadcasts a sequence of instructions to the N processors that execute these instructions in lockstep (this is the “single instruction stream”). All “enabled” (active) processors execute the same instruction at the same time, but each processor does it on its own data. The operands for these instructions are fetched from the local memory associated with each processor (the fetching of operands from the collection of local memories form the “multiple data streams”). The *interconnection network* supports inter-processor communication. Thus, the SIMD definition used here assumes a collection of separate processors operating in lockstep, with all enabled processors following the same single instruction stream, but each processor operating on its own local data, resulting in multiple data streams. Examples of SIMD systems that have been constructed are Illiac IV [6], MasPar MP-1 and MP-2 [5], and MPP [4].

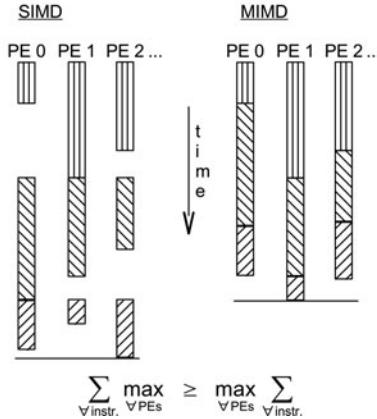
A *multiple-SIMD* system can be structured as a single SIMD machine or as two or more independent SIMD machines of various sizes. The Thinking Machines CM-2 [23] and original design of the Illiac IV [3] are examples of MSIMD systems.

The *MIMD* (multiple instruction stream, multiple data stream) mode of parallelism [8] uses N independent processor-memory pairs that can communicate via an interconnection network (i.e., a multicomputer). Each processor fetches its own instructions and its own operands from its local memory; thus, there are “multiple instruction streams” and “multiple data streams.” The nCUBE 2 [9] and IBM RP3 [15] are examples of MIMD systems that have been constructed. One mode of operation possible for MIMD is *SPMD* (single program, multiple data stream), where all processors independently execute the same program on different data sets [13].

A *mixed-mode* system can dynamically change between the SIMD and MIMD modes of parallelism at instruction-level granularity with negligible overhead. This allows different modes of parallelism to be used to execute various portions of an algorithm. Because the mode of parallelism has an impact on performance (see Fig. 1), a mixed-mode system may outperform a single-mode machine with the same number of processors for a given algorithm. A *partitionable mixed-mode* system can dynamically restructure to form independent

- SIMD Advantages**
- Ease of programming and debugging
    - SIMD: single program, PEs operate synchronously
    - MIMD: multiple interacting programs, PEs operate asynchronously
  - Overlap loop control with operations
    - SIMD: control unit does increment and compare while PEs “compute”
    - MIMD: same processor does both
  - Overlap operations on common data
    - SIMD: control unit performs operations that all PEs need (e.g., common local array addresses) while PEs “compute”
    - MIMD: same processor does all
  - Reduced inter-PE transfer overhead
    - SIMD: “send” and “receive” automatically synchronized
    - MIMD: need explicit synchronization and identification protocol
  - Minimal synchronization overhead
    - SIMD: implicit in program
    - MIMD: need explicit statements (e.g., semaphores)
  - Less program memory space required
    - SIMD: store one copy of program
    - MIMD: each PE stores own copy
  - Minimal instruction decoder cost
    - SIMD: decoder in control unit
    - MIMD: decoder in each PE

- MIMD Advantages**
- More flexible
    - No constraints on operations that can be performed concurrently
  - Conditional statements more efficient
    - SIMD: “then” and “else” execution serialized
    - MIMD: each PE executes as if uniprocessor
  - No SIMD control unit cost
  - Variable-time instructions more efficient
    - Assume there is a block of instructions where the execution time of each instruction is data dependent
    - SIMD: waits for slowest PE to execute each instruction (“sum of maxes”)
 
$$T_{\text{SIMD}} = \sum_{\forall \text{ instr. } \forall \text{ PEs}} \max(\text{instr. time})$$
    - MIMD: waits for slowest PE to execute block of instructions (“max of sums”)
 
$$T_{\text{MIMD}} = \max_{\forall \text{ PEs}} \sum_{\forall \text{ instr. }} (\text{instr. time})$$
  - Example: execution of three instructions in SIMD mode and MIMD mode



**PASM Parallel Processing System.** Fig. 1 Trade-offs between the SIMD and MIMD modes of parallelism [2]. Processing Elements (PEs) are processor-memory pairs. Examples of variable-time instructions are floating-point operations on the prototype’s Motorola MC68000 processors, and function calls (such as floating-point trigonometric operations) on current GPUs

or communicating submachines of various sizes, where each submachine can independently perform mixed-mode parallelism (e.g., TRAC [10]).

PASM is a PArtitionable-SIMD/MIMD system concept that was developed at Purdue University as a design for a large-scale partitionable mixed-mode machine based on commodity microprocessors [21]. PASM used a flexible multistage interconnection network for interprocessor communication. Thus, PASM could be dynamically reorganized along these three dimensions: partitionability, mode of parallelism, and connections among processors. This ability to be reorganized

allowed the system to match the computational structure of a problem, and also provided for fault tolerance in many situations. This fault tolerance is a form of robustness [1, 17], where the robust behavior requirement is continued system operation given the uncertainty of any single component failure, and quantified as the portion of the system that can still be used after a fault occurs.

A small-scale prototype was completed in 1987 as a proof-of-concept machine for the PASM design ideas, and was still running in 1995, with over 36,000 hours of execution time logged (Fig. 2) [20]. It was used



**PASM Parallel Processing System. Fig. 2** A photograph of the completed PASM prototype circa 1987 and some of the PASM team. *Left to right: Pierre Perrot (Technician), Tom Casavant (Professor), Wayne Nation (PhD Student), H.J. Siegel (Professor, team leader)*

for research and in a parallel programming course at Purdue. The goal of the PASM research team was to design, develop, and build a unique research tool for studying the combined three dimensions of dynamic reorganization mentioned above: mixed-mode parallelism, partitionability, and flexible interprocessor communications.

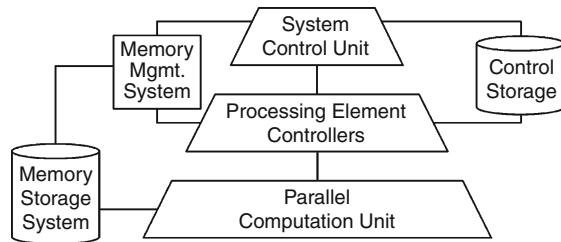
An overview of the organization of PASM is given in section “►The Overall PASM Organization.” Section “►The Parallel Computation Unit” describes the processing elements, memory, and interconnection network in the Parallel Computation Unit. The Memory Storage and Memory Management Systems are discussed in section “►The Memory Storage and Management Systems.” Section “►Using the PASM System” mentions some PASM prototype software tools and application studies. We conclude in section “►Conclusions” with a list of advantages of the PASM approach. A list of PASM-related publications is available at <http://hdl.handle.net/10217/34662>.

### The Overall PASM Organization

The PASM concept was a distributed-memory machine with a computational engine consisting of processor-memory pairs referred to as *PEs* (Processing Elements). The PASM design concepts could support at least 1,024 PEs. The small-scale prototype built at Purdue University (30 Motorola MC 68,000 processors, 16 in

the computational engine) supported experimentation with all three dimensions of reconfigurability mentioned earlier, and produced insights not observed from earlier simulations and theoretical studies.

The PASM system concept consists of six basic components shown in Fig. 3. The *System Control Unit* is the overall system coordinator and is the part of PASM with which the user directly interacts. The *Q PE Controllers (PECs)* serve as the PE control units in SIMD mode and may coordinate the PEs in MIMD mode. The *Parallel Computation Unit* contains the  $N = 2^n$  PEs, physically numbered 0 to  $N - 1$ , and the interconnection network used by the PEs. The *Memory Storage System* provides secondary storage for the PEs, and consists of  $N/Q$  secondary storage devices, each with an associated processor for file management. It is used to store data files for SIMD mode and both program and data files for MIMD mode. The *Memory Management System* contains multiple processors for controlling the transferring of files between the Memory Storage System and the PEs. *Control Storage* consists of a secondary storage device and an associated file server processor. It supports the PECs and the System Control Unit by holding all code and data used by these components, including the instructions to be broadcast to the PEs from the PECs in SIMD mode. The PASM prototype had a total of 30 processors:  $N = 16$  PEs,  $Q = 4$  PECs,  $N/Q = 4$  Memory Storage System processors, four Memory Management



**PASM Parallel Processing System.** Fig. 3 The high-level architectural organization of the PASM design concept

System processors, the Control Storage processor, and the System Control Unit.

The tasks to be performed by the System Control Unit include support for program development, job scheduling, general system coordination, management of system configuration and partitioning, assignment of user jobs to submachines, and connection to the host computer network. The hardware needed to combine and synchronize the PECs and PEs to form SIMD submachines of various sizes resides in the System Control Unit. Its functions include combining information from multiple PECs when collective conditionals, discussed in the next section, are performed. It also is responsible for coordinating the loading of the PE memories from the Memory Storage System with the loading of the PEC memories from the Control Storage.

We now describe how the PECs are connected to the PEs, and how this connection scheme is used to form independent mixed-mode submachines. The organization we use provides an efficient PEC/PE interface and supports the partitioning of the PE Interconnection Network.

The PECs, shown in Fig. 4, are the multiple control units required to form a multiple-SIMD system. There are  $Q = 2^q$  PECs, physically numbered from 0 to  $Q - 1$ . Each PEC controls a fixed group of  $N/Q$  PEs in the Parallel Computation Unit. A PEC and its associated PEs form a *PEC Group*. PEC  $i$  is connected to the  $N/Q$  PEs whose low-order  $q$  bits of their PE physical number have the value  $I$  (for reasons discussed in section “►The Parallel Computation Unit”). In an  $N = 1,024$  system,  $Q$  may be 32; for the  $N = 16$  prototype,  $Q = 4$ . Figure 5 shows the composition of the physical number of a PE.

The memory units in each PEC are double-buffered so that computation and memory I/O can be overlapped

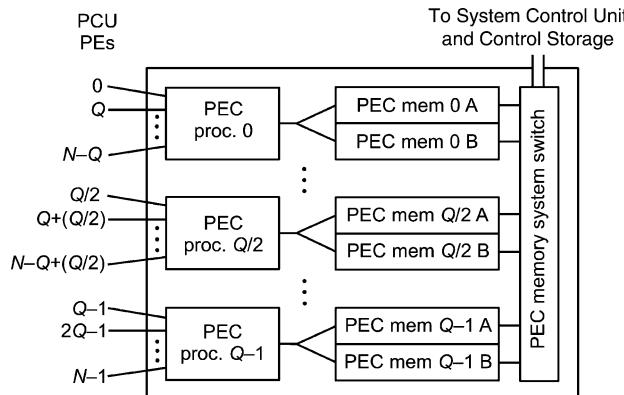
(see Fig. 4). For example, the PEC processor can execute a job in one memory unit while the next job is preloaded into the other memory unit from PEC secondary storage (the Control Storage). In MIMD mode, a PEC fetches from its memory the instructions and data used to coordinate the operation of its PEs. In SIMD mode, a PEC fetches the instructions and common PE data from its memory units. In general, in SIMD mode, control-flow instructions are executed in the PEC, and data processing instructions are broadcast to the PEC’s group of PEs.

*Submachines* are formed by one or more PEC Groups. The partitioning rule in PASM is that the numbers of all PEs in a submachine of size  $2^p$  must agree in their  $n - p$  low-order bit positions (for the reasons discussed in section “►The Parallel Computation Unit”). The  $p$  high-order bits of the physical number of a PE correspond to the PE’s logical number within the partition. Thus, a submachine containing  $R = 2^r$  PEC Groups ( $R^*N/Q = 2^p$  PEs), where  $0 \leq r \leq q$ , is formed by combining the PEs connected to the  $R$  PECs whose addresses agree in their  $q - r$  low-order bits. The PECs within a submachine of  $R^*N/Q$  PEs are logically numbered from 0 to  $R - 1$ . For  $R > 1$ , the logical number of a PEC is the high-order  $r$  bits of its physical number. Similarly, the PEs assigned to a submachine are logically numbered from 0 to  $(R^*N/Q) - 1$  (0 to  $2^p - 1$ ). The logical number of a PE is the high-order  $r + n - q = p$  bits of its physical number (see Fig. 5). There is a maximum of  $Q$  submachines, each of size  $N/Q$ .

Each submachine can operate as an independent mixed-mode system. PEs can switch between modes as often as desired; however, all PEs in a submachine must be in the same mode (SIMD or MIMD) at any point in time.

When a submachine is operating in SIMD mode, the  $R$  PECs must execute and broadcast the same instructions, and all PEs in the submachine must be synchronized. The PECs are given the same instruction simply by loading the memory units of the  $R$  PECs with the same program. The PEs are synchronized by providing a small amount of special circuitry in the System Control Unit that coordinates instruction broadcasts among these PECs.

Some advantages of this fixed PE to PEC mapping as compared with a dynamic PE to PEC interconnection (e.g., a crossbar switch [14]) include reducing



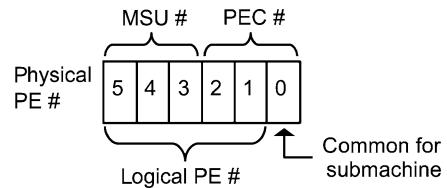
**PASM Parallel Processing System. Fig. 4** PASM Processing Element Controllers (PECs), and how they are connected to the Parallel Computation Unit (PCU) Processing Elements (PEs)

PEC/PE interface hardware, eliminating the overhead of maintaining a record of PE to PEC assignments, scheduling only  $Q$  PECs instead of  $N$  PEs, allowing the partitioning of the PE interconnection network into independent subnetworks (section “► [The Parallel Computation Unit](#)”), and supporting the structure of the efficient parallel primary to secondary memory connections (section “► [The Memory Storage and Management Systems](#)”). The main disadvantage to this approach is that the size of each submachine must be a power of two, with a minimum of  $N/Q$  PEs. However, for PASM’s intended experimental environment, flexibility at a “reasonable” cost is the goal, not maximum PE utilization.

## The Parallel Computation Unit

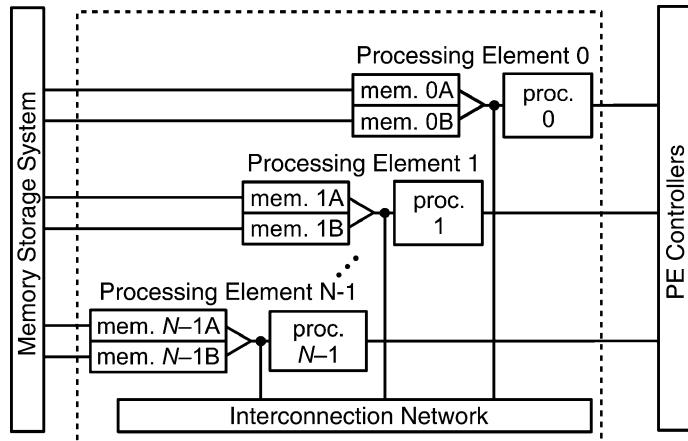
The Parallel Computation Unit, shown in [Fig. 6](#), contains  $N = 2^n$  PEs and an interconnection network. Similar to the double-buffering for the PECs, two memory units are used in each PE so that computation and memory I/O can be overlapped. For example, the PE processor can execute a job in one memory unit while the next job is preloaded into the other memory unit from PE secondary storage (the Memory Storage System). These memory units compose the primary memory of the system. In SIMD or MIMD mode, each PE can perform indirect memory addressing using its own PE logical number or local data.

Consider how PASM switches dynamically between the SIMD and MIMD modes of parallelism in our implemented prototype ([Fig. 7](#)). In MIMD mode, a PE processor fetches instructions from its local memory by

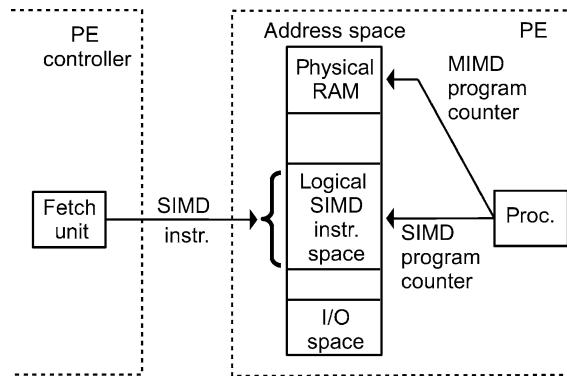


**PASM Parallel Processing System. Fig. 5** Physical number of a Processing Element (PE) for a system where  $N = 64 = 2^6$  PEs,  $Q = 8 = 2^3$  Processing Element Controllers (PECs), and  $N/Q = 8 = 2^3$  Memory Storage Units (MSUs). The PE is in a submachine of size  $R^*N/Q = 4$  PEC Groups (corresponding to  $32 = 2^5$  PEs). For this example, the physical numbers of all PEs in this submachine agree in bit position 0

placing its program counter value on the address bus and latching the data placed on the data bus by the memory device holding that instruction word. A PE is forced into SIMD mode by executing a jump to the “logical” SIMD instruction space, which is not a physical space. [Figure 7](#) illustrates the address space of a PE, showing the logical SIMD instruction space, physical RAM, and I/O space. Logic in the PE detects read accesses to the logical SIMD instruction space, and any such access sends a SIMD instruction request to the PE’s PEC. The SIMD instruction is issued by the submachine’s PECs only after all the PEs of a submachine have requested the instruction (recall the System Control Unit has special hardware to synchronize the PECs that are part of the same submachine). The instructions are sent by the submachine PECs’ Fetch Units to the PEs ([Fig. 7](#)). All enabled PEs in the submachine



PASM Parallel Processing System. Fig. 6 The PASM Parallel Computation Unit



PASM Parallel Processing System. Fig. 7 How the Processing Element (PE) logical address space in the PASM prototype is used to support mixed-mode computation

then execute the instruction simultaneously, each PE on its own data. Thus, all the PEs of the submachine are synchronized when a SIMD instruction is broadcast. The PE continues to access instructions from the SIMD instruction space until an instruction is broadcast that causes each PE to jump to an instruction in its physical RAM (Fig. 7), changing to MIMD mode. Such flexibility in mode switching allows mixed-mode programs to be written that change modes at instruction-level granularity with negligible overhead. The SIMD instruction fetch mechanism also can be used to support a form of MIMD “barrier synchronization” [20].

Masking schemes are used in SIMD mode to enable and disable PEs. The PASM system uses PE-address

masks (originated by the PASM project [21]) and data-conditional masks. A *PE-address mask* enables a set of PEs based solely on the logical addresses (numbers) of PEs; it does not depend on local PE data. A PE-address mask has  $n$  positions, where each position contains either a 0, 1, or X (“don’t care”). A PE is enabled only if the binary representation of its number matches the mask. For example, for  $N = 64$ ,  $[5\{X\}0]$  is equivalent to  $[XXXXX0]$  and enables all even-numbered PEs. A *negative PE-address mask* enables all PEs whose addresses do not match the mask. For example, for  $N = 64$ ,  $[4\{X\}2\{0\}] = [XXXXX0]$  enables all PEs whose numbers are not multiples of four. PE-address masks are a convenient notation for enabling PEs in a large-scale parallel machine, and proved very useful in image processing applications.

*Data-conditional masks* enable PEs based on some condition that is dependent on local PE data. The resulting data condition may be true in some PEs and false in other PEs. An example use of data-conditional masking is the *where* statement, the SIMD counterpart to the *if-then-else* statement. When the segment

```
where <data-condition> do <where-part>
elsewhere <else-part>
```

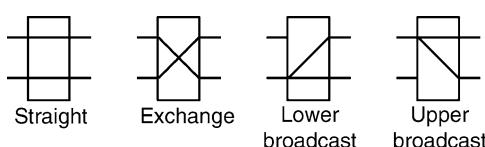
is executed, each PE independently evaluates the *<data-condition>*. The PEs where the condition evaluates true execute the *<where-part>*, and the PEs where the condition evaluates to false are idle. Next, the PEs where the condition evaluates to false

execute the `<else-part>`, and the PEs where the condition evaluates true are idle. This type of masking is used in many SIMD machines (e.g., CM-2, MP-1). Data-conditional and PE-address masks can be used together.

There are situations in which the combined conditional status of all enabled PEs in a SIMD submachine is needed, e.g., `if-none`, `if-any`, and `if-all1`. For example, if the PEs of a submachine are each examining a portion of an image to find certain objects, it may be necessary to know whether any of the PEs have found such an object. Because PASM is a partitionable system, operations such as these require conditional results to be communicated from the PEs to their PECs and subsequently combined among the PECs comprising a SIMD submachine. These operations are efficiently supported by providing a small amount of additional circuitry in the System Control Unit that combines PEC Group results according to the current system partitioning.

The *Interconnection Network* allows PEs to communicate with each other. As described earlier, the partitioning rule in PASM requires that the physical numbers of all PEs in a submachine of  $2^p$  PEs agree in their  $n - p$  low-order bit positions (see Fig. 5). Thus, the  $p$  high-order bits of a PE's physical number form its logical number within a submachine of  $2^p$  PEs. The low-order partitioning rule was chosen for PASM so that either the multistage cube [18] or the augmented data manipulator (ADM) [18] networks could be used. However, the multistage cube was selected because of its comparative cost-effectiveness.

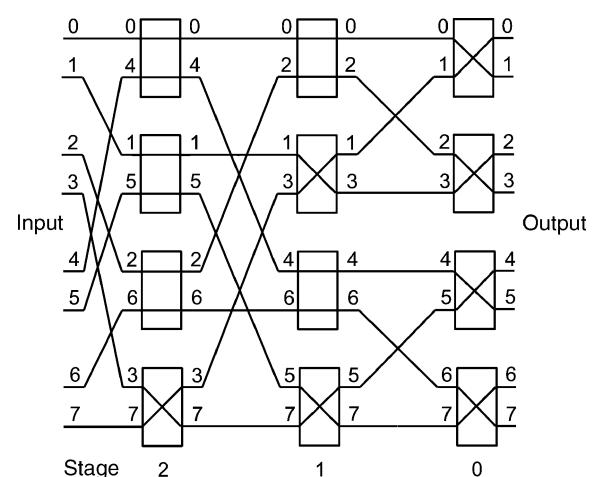
The *multistage cube network* has  $N$  input ports,  $N$  output ports, and contains  $n = \log_2 N$  stages of  $N/2$  two-input/two-output *interchange boxes*. Each interchange box can be set to one of the four states shown in Fig. 8. The network can be used in both the SIMD and MIMD modes of parallelism.



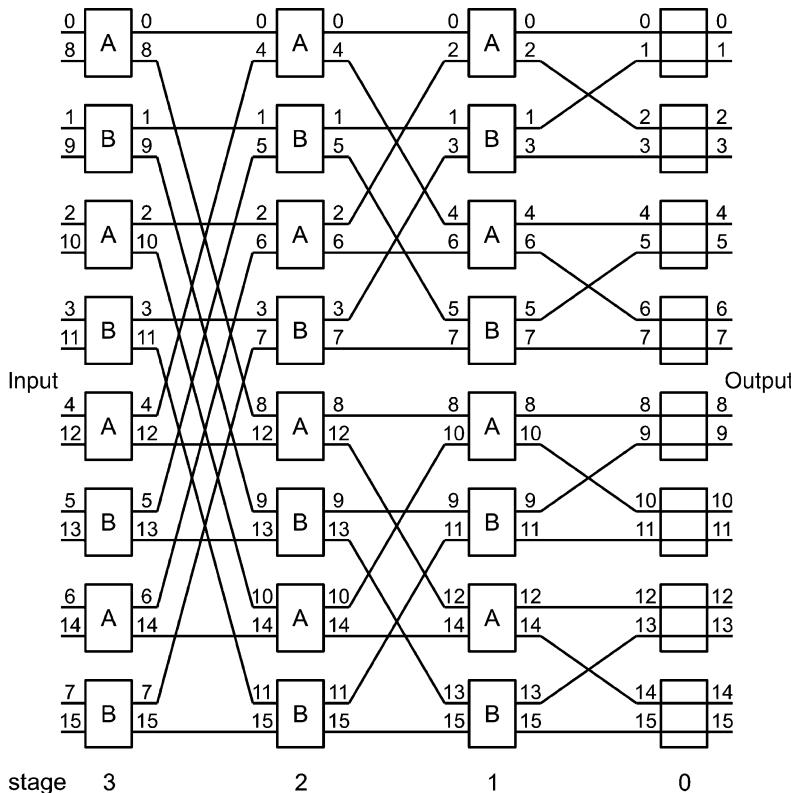
**PASM Parallel Processing System. Fig. 8** The four valid states of a multistage cube network interchange box

Figure 9 shows the multistage cube network for  $N = 8$ . PE  $i$  is connected to network input port  $i$  and output port  $i$ . The stages are numbered consecutively from  $n - 1$  at the input stage to 0 at the output stage. The upper interchange box output label is the same as the upper input, and the lower interchange box output label is the same as the lower input. The interconnection pattern between stages is such that at stage  $j$  the two links whose labels differ only in bit  $j$  are connected to the same interchange box. Figure 9 shows the network set for a “permutation” connection ( $\text{input } i \rightarrow \text{output } (i + 1) \bmod 8$ ) for an  $N = 8$  multistage cube network. The network can be controlled in a distributed fashion using routing tags for specifying permutations, one-to-one connections (e.g., PE  $a$  to PE  $b$ ), one PE to many multicast connections, and combinations of these [18].

A network is *partitionable* if it can be divided into independent subnetworks of smaller sizes that have all the properties of the original network [18]. In general, a size  $N$  multistage cube network can be partitioned into multiple subnetworks of different sizes, where the size of each subnetwork is a power of two. For example, Fig. 10 shows an  $N = 16$  network partitioned into two subnetworks by setting the interchange boxes in stage 0 to straight. Because stage 0 is straight, even-numbered inputs cannot reach odd-numbered outputs, and odd



**PASM Parallel Processing System. Fig. 9** A multistage cube network for  $N = 8$  Processing Elements (PEs) set to perform the “permutation” of sending data from PE  $i$  to PE  $(i + 1) \bmod N$  for  $0 \leq i < N$



**PASM Parallel Processing System. Fig. 10** A multistage cube network for  $N = 16$  Processing Elements (PEs) partitioned into two independent subnetworks, each of size eight. Subnetwork A consists of the even-numbered input and output ports, and subnetwork B the odd

inputs cannot reach even outputs. The two resulting subnetworks are subnetwork *A*, which contains physical ports 0, 2, ..., 14, and subnetwork *B*, which contains physical ports 1, 3, ..., 15. Because each of these subnetworks is an independent multistage cube network (of size  $N/2 = 8$ ), either or both subnetworks may be further partitioned by forcing all interchange boxes in stage 1 of the subnetwork to be straight. This process can be applied recursively.

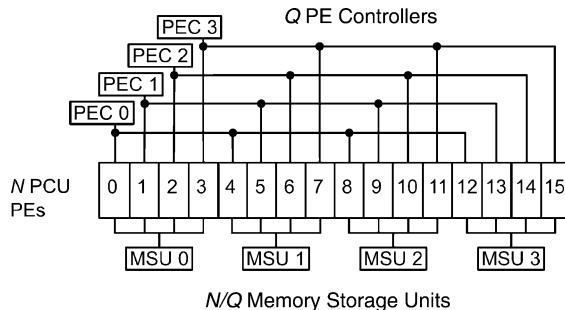
The *Extra Stage Cube network* [18] (designed as a part of the PASM project) is a single-fault tolerant variation of the multistage cube network. There is an extra stage of interchange boxes at the input, labeled stage  $n$ . Links whose labels differ in the 0th bit position are paired at these extra stage boxes in the same way that they are at stage 0. This Extra Stage Cube network fault-tolerant variation has all of the useful properties described above for the multistage cube, and was constructed in the PASM prototype. It is robust in the sense

that it can still provide all of the capabilities of a fully functional multistage network despite the failure of any single network component.

# The Memory Storage and Management Systems

The Memory Storage System (Fig. 6) is the secondary storage for the Parallel Computation Unit, storing data files in SIMD mode and both program and data files in MIMD mode (programs for SIMD mode are stored in the Control Storage). The Memory Storage System is comprised of  $N/Q$  independent *Memory Storage Units* (MSUs), numbered from 0 to  $(N/Q) - 1$ .

Each of the  $N/Q$  MSUs is connected to the memory modules of  $Q$  PEs. An example for the prototype size of  $N = 16$  and  $Q = 4$  is shown in Fig. 11. The benefit of this MSU to PE connection scheme is that the memories of all  $N/Q$  PEs connected to any one PEC can be loaded or unloaded in one parallel block transfer. For



**PASM Parallel Processing System. Fig. 11** The organization of the PASM Memory Storage System for the prototype size of  $N = 16$  Processing Elements (PEs),  $Q = 4$  Processing Element Controllers (PECs), and  $N/Q$  Memory Storage Units (MSUs)

example, in [Fig. 11](#), PEC 2's group of PEs (PEs 2, 6, 10, and 14) are loaded by having MSU 0 load PE 2, MSU 1 load PE 6, MSU 2 load PE 10, and MSU 3 load PE 14, all simultaneously.

In the general case, MSU  $i$  is connected to and stores files for the  $Q$  PEs whose  $n - q$  high-order bits of their PE physical numbers are equal to  $i$  (see [Fig. 5](#)). This high-order mapping is used so that each of the  $N/Q$  PEs connected to a given PEC is connected to a different MSU. Recall that the low-order  $q$  bits of the physical number of a PE correspond to the number of the PEC to which it is attached (see [Fig. 5](#)). Thus, the full bandwidth of the Memory Storage System can be used when transferring files between the Memory Storage System and the Parallel Computation Unit. If there are only  $N/(Q^*D)$  distinct MSUs, where  $1 \leq D \leq N/Q$ , then this scheme can be scaled so that  $D$  parallel block transfers are required to load or unload the memories of the PEs connected to any one PEC. Each MSU contains a mass storage unit and a processor to manage the file system and to transfer files to and from its associated PE memory units.

Similarly, a submachine formed by combining the  $(R^*N)/Q$  PEs controlled by  $R$  PECs can be loaded or unloaded in  $R$  parallel block transfers if there are  $N/Q$  MSUs. For example, in [Fig. 11](#), a submachine comprised of the PEs of PEC 0 and PEC 2 can be loaded in  $R = 2$  parallel block loads as follows: First PEC 0's PEs are loaded in one parallel block transfer, and then PEC

2's PEs are loaded. If there are only  $N/(Q^*D)$  distinct MSUs, only  $R^*D$  parallel block transfers are required.

[Figure 5](#) demonstrates which MSU connects to which PE. MSU  $i$  is connected to the PE whose high-order  $n - q$  bits of its logical number are  $i$  (which equal the  $n - q$  high-order bits of its physical number). As a result, no matter which PECs are assigned to a task, the data will be in the appropriate MSUs. For example, in [Fig. 11](#), for any submachine of size  $N/Q = 4$ , MSU 0 is connected to logical PE 0 (which could be physical PE 0, 1, 2, or 3).

The Memory Management System ([Fig. 3](#)) is a set of processors that send file system requests from the System Control Unit, PECs, and PEs to the appropriate MSUs; control data transfers between the Memory Storage System and the PEs; supervise input/output operations involving peripheral devices; and enforce consistent file naming and placement across the multiple Memory Storage System disks.

## Using the PASM System

ELP (Explicit Language for Parallelism) was a C-based language and associated compiler designed as part of the PASM project for programming mixed-mode parallel machines [13]. ELP provided constructs for both SIMD and MIMD parallelism, and an ELP application program could perform computations that use these parallelism modes in an instruction-level interleaved fashion for mixed-mode operation. ELP provided a vehicle for the exploration of and experimentation with mixed-mode parallelism on the PASM prototype.

CAPS (Coding Aid for the PASM System) was designed to assist in the development and evaluation of application and system software for the PASM prototype [11]. CAPS integrated hardware support and software tools to provide a remote execution and program debugging/monitoring environment for the PASM prototype. The PASM prototype was accessible over the Internet using CAPS, and multiple windows could be displayed to monitor different processors simultaneously.

The programming challenges for partitionable mixed-mode systems are a superset of those for SIMD and MIMD single-mode systems. Application and algorithm research activities to explore PASM's three dimensions of flexibility included theoretical analyses,

simulations, and experiments on the PASM prototype [19]. These studies examined issues such as mapping tasks onto multistage cube network-based parallel processing systems; trade-offs among the SIMD, MIMD, and mixed-mode classes of parallelism; PEC/PE computational overlap in SIMD mode; impact of increasing the number of PEs used; and partitioning for improved performance. Applications considered include edge-guided thresholding, FFTs, global histogramming, image correlation, image smoothing, matrix multiplication, and range-image segmentation (references are given in Armstrong, Watson, and Siegel [2]). Here, we will summarize three application studies that utilized the PASM prototype.

Fineberg, Casavant, and Siegel [7] used the PASM prototype to study the benefits of mixed-mode parallel architectures for bitonic sequence sorting. Four variations of the bitonic sequence sorting algorithm were developed: a SIMD version, a MIMD version, a MIMD version with hardware barrier synchronizations, and a mixed-mode version that allowed switching between SIMD and MIMD modes during execution. The algorithm variations were then coded and profiled on the PASM prototype using varying problem and partition sizes.

The research demonstrated that, by supporting hardware barrier synchronization and mode switching, a machine can achieve better performance than with only SIMD or MIMD parallelism on problems that are computationally similar to bitonic sequence sorting. The authors also proposed a modification to the PASM prototype condition code logic that would increase the performance of the SIMD version.

Saghi, Siegel, and Gray [16] programmed cyclic reduction (a method for solving a general tridiagonal set of irreducible linear algebraic equations) on three parallel systems, including the PASM prototype, to study the trade-offs between SIMD and MIMD parallelism, the effects of increasing the number of processors used on execution time, the impact of the interconnection network on performance, and the advantages of a partitionable system. The cyclic reduction algorithm was implemented using SIMD parallelism on the MasPar MP-1 [5], using MIMD parallelism on the nCUBE 2 [9], and using the same four versions of parallelism on the PASM prototype as mentioned above. The authors also developed a mechanism

to predict the algorithm performance on each machine using algorithm analysis and performance measurements from each machine.

By using the PASM prototype as a common basis to compare the modes of parallelism, the authors concluded that a cyclic reduction algorithm using mixed-mode was better than a purely SIMD or purely MIMD algorithm. Execution-time measurements from the other two parallel machines emphasized the need to carefully choose the number of processing elements and distribution of data to obtain efficient computation.

Tan, Siegel, and Siegel [22] used several parallel machines to study block-based motion vector estimation. A SIMD MasPar MP-1 [5], a MIMD Intel Paragon XP/S, a MIMD IBM SP2, and the mixed-mode PASM prototype were used with different data partitioning schemes to perform the estimation, and the results were analyzed to contrast the benefits of each mode of parallelism. In this research, the PASM prototype used SIMD, SPMD, and mixed-mode parallelism.

The research demonstrated a method to analytically predict the performance of various parallel implementations of the algorithm. It also described the impact of the number of processors and mode of parallelism used on algorithm performance. The authors found that, using the PASM prototype, the mixed-mode implementation slightly outperformed the pure SPMD implementation and significantly outperformed the pure SIMD implementation.

## Conclusions

Designing, simulating, prototyping, using, and evaluating the PASM partitionable SIMD/MIMD mixed-mode system, based on a multistage cube network, was a great research and educational experience for a large group of faculty and students at Purdue University from the 1970s through the 1990s. As stated earlier, PASM was flexible along three dimensions: partitionability, mode of parallelism, and variable connectivity among PEs. We found that advantages of systems with such flexibility over a pure SIMD machine or a pure MIMD machine included the following [20]:

1. *Multiple simultaneous users:* Because there can be multiple simultaneous independent submachines, there can be multiple simultaneous users of the

system, each executing a different program (not allowed in a pure SIMD machine).

2. *Program development*: Rather than trying to debug a new parallel program on, for example, 1,024 PEs, it can be debugged on a smaller size submachine of 32 PEs, and then extended to 1,024 PEs.
3. *Variable submachine size for increased utilization*: If a task requires only  $N'$  of  $N$  available PEs, the other  $N - N'$  can be used for another task (not allowed in a pure SIMD machine).
4. *Variable submachine size for decreased execution time*: There are some algorithms for which the minimum execution time is obtained when fewer than  $N$  PEs are used due to inter-PE communication overhead (e.g., image smoothing [19]); thus, it is desirable to create a submachine consisting of the optimal number of PEs.
5. *Subtask parallelism*: Two independent subtasks that are part of the same job can be executed in parallel, sharing resources if necessary, which may result in improved overall task execution time [12] (not allowed in a pure SIMD machine).
6. *Multiple processing modes*: An algorithm can be executed by using a combination of SIMD and MIMD control with the same set of PEs (mixed-mode parallelism), using the mode that best matches the computations required at each step of the program (Fig. 1).
7. *Matching inter-PE connectivity to the task*: The multistage cube allows different connection patterns among PEs to be established depending on the task (as opposed to, for example, having a fixed mesh network).
8. *System fault tolerance*: If a single PE fails, only those submachines that include the failed PE are affected. This provides some robustness, as mentioned in section “►Introduction,” and is not allowed in a pure SIMD machine.
9. *Submachine fault tolerance*: If a PE in a submachine fails, it may be possible to redistribute data and make use of mixed-mode parallelism (i.e., changing from SIMD to MIMD mode) and the variable connectivity (i.e., to establish connection patterns that do not include the faulty PE) so that the job executing on the submachine may continue on that submachine with minimal degradation (this is another form of robustness).

The references cited in this list of advantages, and the complete reading list of PASM-related publications (<http://hdl.handle.net/10217/34662>), give much more detail about our experiences.

## Related Entries

- [Connection Machine](#)
- [Distributed-Memory Multiprocessor](#)
- [Flynn's Taxonomy](#)
- [Illiac IV](#)
- [MasPar](#)
- [MPP](#)
- [nCube](#)
- [Networks, Multistage](#)

## Acknowledgments

The preparation of this entry was supported by the National Science Foundation under grants CNS-0615170 and CNS-0905399, and by the Colorado State University George T. Abell Endowment. The large group of faculty and students who have participated in the PASM project are the coauthors of the papers listed in the PASM-related reading list (<http://hdl.handle.net/10217/34662>). Numerous agencies supported aspects of PASM-related research: Air Force Office of Scientific Research, Army Research Office, Ballistic Missile Defense Agency, Defense Mapping Agency, Naval Ocean Systems Center, Naval Research Laboratory, National Science Foundation, Office of Naval Research, and Rome Laboratory. IBM provided a grant for much of the prototype equipment. Donations for various parts for the prototype were provided by Amphenol Products, Augat Inc., Belden, Motorola, and Power One.

## Bibliographic Notes and Further Reading

This entry is a brief summary of the PASM architecture; details are available in the papers in the PASM-related reading list available at <http://hdl.handle.net/10217/34662>.

## Bibliography

1. Ali S, Maciejewski AA, Siegel HJ, Kim J (2004) Measuring the robustness of a resource allocation. *IEEE Trans Parallel Distrib Syst* 15(7):630–641
2. Armstrong JB, Watson DW, Siegel HJ (1993) Software issues for the PASM parallel processing system. *Software for parallel computation*. Springer, Berlin

3. Barnes GH, Brown RM, Kato M, Kuck DJ, Slotnick DL, Stokes RA (1968) The ILLIAC IV computer. *IEEE Trans Comput C* 17(8):746–757
4. Batcher KE (1982) Bit serial parallel processing systems. *IEEE Trans Comp C-31*(5):377–384
5. Blank T (1990) The MasPar MP-1 architecture. In: Proceedings IEEE compcon spring '90, pp 20–24
6. Bouknight WJ, Denenberg SA, McIntyre DE, Randall JM, Sameh AH, Slotnick DL (1972) The ILLIAC IV system. *Proc IEEE* 60(4):369–388
7. Fineberg SA, Casavant TL, Siegel HJ (1991) Experimental analysis of a mixed-mode parallel architecture using bitonic sequence sorting. *J Parallel Distrib Comput* 11(3):239–251
8. Flynn MJ (1966) Very high-speed computing systems. *Proc IEEE* 54(12):1901–1909
9. Hayes JP, Mudge T (1989) Hypercube supercomputers. *Proc IEEE* 77(12):1829–1841
10. Lipovski GJ, Malek M (1987) Parallel computing: theory and comparisons. Wiley, New York
11. Lump JE, Fineberg SA, Nation WG, Casavant TL, Bronson EC, Siegel HJ, Pero PH, Schwederski T, Marinescu DC (1991) CAPS – a coding aid used with the PASM parallel processing system. *Commun ACM* 34(11):104–117
12. Nation WG, Maciejewski AA, Siegel HJ (1993) A methodology for exploiting concurrency among independent tasks in partitionable parallel processing systems. *J Parallel Distrib Comput* 16(3): 271–278
13. Nichols MA, Siegel HJ, Dietz HG (1993) Data management and control-flow aspects of an SIMD/SPMD parallel language/compiler. *IEEE Trans Parallel Distrib Syst* 4(2):222–234
14. Nutt GJ (1977) Microprocessor implementation of a parallel processor. In: Proceedings 4th annual symposium on computer architecture, pp 147–152
15. Pfister GF, Brantley WC, George DA, Harvey SL, Kleinfelder WJ, McAuliffe KP, Melton ES, Norton VA, Weiss J (1985) The IBM research parallel processor prototype (RP3): introduction and architecture. In: Proceedings 1985 International conference parallel processing, pp 764–771
16. Saghi G, Siegel HJ, Gray JL (1993) Predicting performance and selecting modes of parallelism: a case study using cyclic reduction on three parallel machines. *J Parallel Distrib Comput* 19(3): 219–233
17. Shestak V, Smith J, Maciejewski AA, Siegel HJ (2008) Stochastic robustness metric and its use for static resource allocations. *J Parallel Distrib Comput* 68(8):1157–1173
18. Siegel HJ (1990) Interconnection networks for large-scale parallel processing: theory and case studies, 2nd edn. McGraw-Hill, New York
19. Siegel HJ, Armstrong JB, Watson DW (1992) Mapping computer-vision-related tasks onto reconfigurable parallel-processing systems. *IEEE Comput* 25(2):54–63
20. Siegel HJ, Schwederski T, Nation WG, Armstrong JB, Wang L, Kuehn JT, Gupta R, Allemand MD, Meyer DG, Watson DW (1996) The design and prototyping of the PASM reconfigurable parallel processing system. *Parallel computing: paradigms and applications*. International Thomson Computer Press, London
21. Siegel HJ, Siegel LJ, Kemmerer F, Mueller PT Jr, Smalley HE Jr, Smith SD (1981) PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Trans Comput C* 30(12):934–947
22. Tan M, Siegel JM, Siegel HJ (1999) Parallel implementations of block-based motion vector estimation for video compression on four parallel processing systems. *Int J Parallel Program* 27(3): 195–225
23. Tucker LW, Robertson GG (1988) Architecture and applications of the connection machine. *IEEE Comput* 21(8):26–38

## Path Expressions

ROY H. CAMPBELL

University of Illinois at Urbana-Champaign, Urbana, IL, USA

## Synonyms

**Coordination;** **Concurrency control;** **Mutual exclusion;** **Process synchronization**

## Definition

A path expression is a declaration of the permitted sequences of operations on an object that is shared by parallel processes.

For example, consider a path expression based on a regular expression notation for a shared buffer. The shared buffer may have operations read and write that may only occur in a sequence specified by the regular expression (write; read)\* where a write to the buffer may be followed by a read before it can be repeated. The reads and writes are mutually exclusive. The expression indicates the sequence of actions that may occur independent of the number of processes or the order in which the processes invoke the operations.

## Discussion

Path expressions provide a declarative specification of synchronization and coordination of parallel processes performing operations on an object in parallel computing systems [4]. They provide a language mechanism to separate the specification of synchronization from the algorithmic means of providing that synchronization [5]. Although the regular expression is used in our example, other languages that are well defined can be used instead.

In general, the synchronization for an object being manipulated by parallel processes involves mutual exclusion, sequence, and repetition and these may be combined to form path expressions. In more detail, a path expression language based on regular expressions can be described as:

1. A sequence of operations on a resource.
2. A selection from one or more operations on a resource.
3. A repetition of a sequence or selection.
4. A path expression composed of (1), (2), or (3) above.

Other primitives besides these three have been considered including:

1. A burst of parallel sequences of executions: If there is one execution of the operations in a path expression, then there can be parallel executions of the sequences of operations in a path expression up until the event when there are no more parallel executions of those operations occurring. For example, the expression  $(\text{write}, \{\text{read}\})^*$  for a buffer would specify that a write or a burst of parallel reads can occur, but not a read and a write in parallel [4].
2. A concurrency restriction: This allows parallel executions of the path expression up to the limit of the restriction. For example, the expression  $5:(\text{write}; \text{read})$  would allow up to 5 parallel instances of a write followed by a read [13].
3. Guarded selection: A path expression selection that describes the synchronization of the parallel execution of its operations when a guard is true. For example, the expression  $\#\text{buffers} < 5 | (\text{put}), \#\text{buffers} > 0 | (\text{get})$  would describe synchronization where if the variable  $\#\text{buffers}$  is less than 5, a put may occur or if the variable  $\#\text{buffers}$  is greater than 0, a get may occur (see also [2]).

More general forms of path expression may also be described [4]. For example, a general path expression can be composed of independent path expressions such that if all the path expressions allow the execution of an operation, it may occur. The general path expression:

```
path (A; B)* end
path (A; C)* end
```

requires an operation A to be followed by operation B and C. Operations B and C may possibly occur in parallel since there are no constraints between B and C but there are constraints between A and B and between A and C.

## Implementation

An operation on an object can be considered as a transition that changes the state of the object. When a process invokes an operation on an object, it can be viewed as a request for a transition from one state of the object to another. The implementation of a path expression decides whether to accept the request and proceed or whether to delay the request until an appropriate state and block the process invocation of that operation. When a path expression is built upon a regular expression that has the property that an operation can only occur when the object is in one particular state, then a simple implementation can be built using semaphores to represent the states.

In this exposition, the implementation for open path expressions is described. An open path expression allows much of the power of a semaphore to be used in a declarative expression and forms the basis for the implementation of “Path Pascal,” a version of Pascal that includes concurrency and synchronization [13]. Open path expressions are composed of four constraints: sequence, selection, restriction, and derestriction.

The open path expression notation allows a simple rewriting scheme to translate a path expression synchronization declaration into the prologues and the epilogues of semaphore operations of each of the operations mentioned in the path expression:

```
[path<op_expr>end] ←
|α <op_expr>β
```

Where  $\leftarrow$  designates a rewrite rule and  $\alpha^i, \beta^i$  are empty strings of semaphore operations forming the prologue and epilogue to be inserted in each of the operations named in op\_expr.

The following set of path expression rewrite rules evaluated in the order specified either by parentheses or by following precedence from left to right generate prologues and epilogues that implement the four synchronization constraints: sequence, selection, restriction, and burst:

*Rule Selection (comma):*  $\alpha <\text{op\_expr}_1>, <\text{op\_expr}_2> \beta \vdash$

|  $\alpha <\text{op\_expr}_1> \beta$   
|  $\alpha <\text{op\_expr}_2> \beta$

*Rule Sequence (semicolon):*  $\alpha <\text{op\_expr}>; <\text{op\_expr}_2>$

$\beta \vdash$

|  $\alpha <\text{op\_expr}_1> "S_j.V();"$   
| "S\_j.P();"  $<\text{op\_expr}_2> \beta$

where Semaphore  $S_j$  is initialized to 0

*Rule Restriction (colon):*  $\alpha n: (<\text{op\_expr}_1>) \beta \vdash$

|  $\alpha || "S_k.P();" <\text{op\_expr}_1> "S_k.V();" || \beta$

where Semaphore  $S_k$  is initialized to n and  $||$  expresses the concatenation of two strings making up a prologue or epilogue.

*Rule Derestriction (brace):*  $\alpha \{<\text{op\_expr}_1>\} \beta \vdash$

| "  $S_k.P();$  if count $++ == 1$  then [ "  $\alpha || "S_k.V();]$ ; " $<\text{op\_expr}_1> " S_k.P();$  if count $-- == 1$  then [ "  $\beta || "S_k.V();]$  ];

where Semaphore  $S_k$  is initialized to 1, integer count is initialized to 0, the  $++$  and  $--$  operators denote incrementing or decrementing a counter count by 1, and square parentheses within the strings denote a block of instructions.

The derestriction rule applies the synchronization prologue  $\alpha$  if count is incremented through 1 in the prologue and applies the synchronization epilogue  $\beta$  if count is decremented through 1 in the epilogue.

Finally, the prologues and epilogues are complete when  $<\text{op\_expr}>$  contains only a method\_name of an operation declared in an object.

*Operation:*  $\alpha <\text{method\_name}> \beta$

| insert prologue  $\alpha$ ; < method\_body>; insert epilogue  $\beta$

The following are some examples of the implementations of open path expressions:

## 1. **path x, y end**

creates empty prologues and epilogues in the methods x and y allowing them to be executed without synchronization constraints.

## 2. **path 1: (x) end**

rewrites to:

|  $S_1.P(); <x\_body>; S_1.V();$

Semaphore  $S_1 = 1$ ;

The body of x is executed in mutual exclusion.

## 3. **path 1: (x, y) end**

rewrites to:

|  $S_1.P(); <x\_body>; S_1.V();$

|  $S_1.P(); <y\_body>; S_1.V();$

Semaphore  $S_1 = 1$ ;

and creates prologues and epilogues in the methods x and y constraining their execution to mutual exclusion.

## 4. **path 1: (x; y) end**

rewrites to:

|  $S_1.P(); <x\_body>; S_2.V();$

|  $S_2.P(); <y\_body>; S_1.V();$

Semaphore  $S_1, S_2 = (1, 0)$ ;

and creates prologues and epilogues in the methods x and y constraining their execution to mutual exclusion and a sequence that alternates the execution of x with one of y.

## 5. **path 1: (write; {read}) end**

rewrites to:

|  $S_1.P(); <write\_body>; S_2.V();$

|  $S_3.P();$  if count $++ == 1$  then [  $S_2.P(); S_3.V();$  ]

| <read\_body>  $S_3.P();$  if count $-- == 1$  then [  $S_2.V(); S_3.V();$  ];

Semaphore  $S_1, S_2, S_3 = (1, 0, 1)$ ; Integer count = 0; and allows one write operation to be followed by a burst of read operations.

## 6. **path 5: (1: (write); 1: (read)) end**

rewrites to:

|  $[S_1.P(); S_2.P(); <write\_body>; S_2.V(); S_4.V();]$

|  $[S_4.P(); S_3.P(); <read\_body>; S_3.V(); S_1.V();]$

Semaphore  $S_1, S_2, S_3, S_4 = (5, 1, 1, 0)$ ;

The body of write is executed in mutual exclusion as is the body of read. However, there must always be more writes than reads and there may be up to 5 more writes than reads.

## Uses of Path Expressions

Path expressions as a declarative form of specifying synchronization have found use in several parallel programming languages [2, 14, 18]. They have also been employed in single-assignment languages [7], real-time control languages [17], distributed objects [16], event systems [12], multimedia systems [11], debugging [3], VLSI design [1], synchronization contracts for web services [15], and workflow languages [10]. Typically,

processes and path expressions are dual approaches to specifying the traces created in a computation; the processes generate sequences of actions which are constrained by path expressions to achieve particular synchronization constraints associated with the operations on abstract data types.

## Summary

In practice, synchronization of parallel processes remains a difficult problem that becomes ever more complex with increased concurrency in architecture. Path expressions provide a separate specification of synchronization from the code of processes that, under some circumstances, can simplify parallel programming. Although not widely adopted, their declarative approach to specifying synchronization has been found useful in various parallel applications. Path expressions have been used to synchronize parallel operations on objects in parallel languages, real-time languages, event systems, VSLI, workflow, and debugging.

## Bibliographic Notes and Further Reading

The semantics of parallel programs may be described using trace-based semantics. Path expression implementations restrict or recognize the permitted traces of parallel programs as described by Lauer and Campbell [6]. A semantics for path expressions is given by Dinning and Mishra using partially ordered multisets [8] and the authors provide a fully parallel implementation for a path expression language on MIMD shared memory architectures. Path Pascal and open path expressions were used as pedagogical tools for teaching parallel programs [9, 13].

## Bibliography

1. Anantharaman TS, Clarke EM, Mishra B (1986) Compiling path expressions into VLSI circuits. *Distrib Comput* 1:150–166
2. Andler S (1979) Predicate path expressions. In: POPL'79 proceedings of the 6th ACM SIGACT-SIGPLAN symposium on principles of programming languages. ACM Press, New York, pp 226–236
3. Bruegge B, Hibbard P (1983) Generalized path expressions: a high-level debugging mechanism. *J Syst Softw* 3:265–276 (Elsevier Science Publishing)
4. Campbell RH (1976) Path expressions: a technique for specifying process synchronization. PhD. Thesis, The University of Newcastle Upon Tyne
5. Campbell RH, Habermann AN (1974) The specification of process synchronization by path expressions. In: Gelenbe E, Kaiser C (eds) Operating systems. Lecture notes in computer science, vol 16. Springer, Berlin, pp 89–102
6. Lauer PE, Campbell RH (1979) Formal semantics of a class of high-level primitives for coordinating concurrent processes, *Acta Informatica*, 5(4):297–332
7. Comte D, Durrieu G, Gelly O, Plas A, Syre JC (1978) Parallelism, control and synchronization expression in a single assignment language. *ACM SIGPLAN Not* 13(1): 25–33
8. Dinning A, Mishra B (1990) A fully parallel algorithm for implementing path expressions. *J Parallel Distrib Comput* 10:205–221
9. Dowsing RD, Elliott R (1986) Programming a bounded buffer using the object and path expression constructs of path pascal. *Comput J* 29(5):423–429
10. Heinlein C (2000) Workflow and process synchronization with interaction expressions and graphs. Ph. D. Thesis (in German), Fakultät für Informatik, Universität Ulm
11. Hoepner P (1992) Synchronizing the presentation of multimedia objects. *Comput Commun* 15(9):557–564
12. Kidd M-EC (1994) Ensuring critical event sequences in high integrity software by applying path expressions. Sandia Labs, Albuquerque
13. Kolstad RB, Campbell RH (1980) Path Pascal user manual. *SIGPLAN Not* 15(9):15–24
14. Laure E (1999) ParBlocks – a new methodology for specifying concurrent method executions in opus. In: Amestoy P, Berger P, Dayde M, Ruiz D, Duff I, Fraysse V, Giraud L (eds) Euro-Par'99. Lecture notes in computer science, vol 1685. Springer, Berlin, pp 925–929
15. Preiss O, Shah AP, Wegmann A (2003) Generating synchronization contracts for web services. In: Khosrow-Pour M (ed) Information technology and organizations: trends, issues, challenges & solutions, vol 1. Idea Group Publishing, Hershey, pp 593–596
16. Rees O (1993) Using path expressions as concurrency guards. Technical report, ANSA
17. Schoute AL, Luursema JJ (1990) Realtime system control by means of path expressions. In: Proceedings Euromicro '90 Workshop on Real Time, Horsholm, Denmark, pp 79–86
18. Shaw AC (1978) Software description with flow expressions. *IEEE Trans Softw Eng SE-4(3):242–254*

## PaToH (Partitioning Tool for Hypergraphs)

ÜMIT ÇATALYÜREK<sup>1</sup>, CEVDET AYKANAT<sup>2</sup>

<sup>1</sup>The Ohio State University, Columbus, OH, USA

<sup>2</sup>Bilkent University, Ankara, Turkey

## Synonyms

Partitioning tool for hypergraphs (PaToH)

## Definition

PaToH is a sequential, multilevel, hypergraph partitioning tool that can be used to solve various combinatorial scientific computing problems that could be modeled as hypergraph partitioning problem, including sparse matrix partitioning, ordering, and load balancing for parallel processing.

## Discussion

### Introduction

Hypergraph partitioning has been an important problem widely encountered in VLSI layout design [22]. Recent works since the late 1990s have introduced new application areas, including one-dimensional and two-dimensional partitioning of sparse matrices for parallel sparse-matrix vector multiplication [6–8, 12], sparse matrix reordering [6, 11], permuting sparse rectangular matrices into singly bordered block-diagonal form for parallel solution of LP problems [3], and static and dynamic load balancing for parallel processing [5]. PaToH [9] has been developed to provide fast and high-quality solutions for these motivating applications.

In simple terms, the hypergraph partitioning problem can be defined as the task of dividing a hypergraph into two or more roughly equal sized parts such that a cost function on the hyperedges connecting vertices in different parts is minimized. The hypergraph partitioning problem is known to be NP-hard [22], therefore a wide variety of heuristic algorithms have been developed in the literature to solve this complex problem [1, 15, 21, 23, 25]. Following the success of multilevel partitioning schemes in ordering and graph partitioning [4, 16, 18], PaToH [9] has been developed as one of the first multilevel hypergraph partitioning tools.

### Preliminaries

A hypergraph  $\mathcal{H} = (\mathcal{V}, \mathcal{N})$  is defined as a set of vertices (also called cells)  $\mathcal{V}$  and a set of nets (hyperedges)  $\mathcal{N}$  among those vertices. Every net  $n \in \mathcal{N}$  is a subset of vertices, that is,  $n \subseteq \mathcal{V}$ . The vertices in a net  $n$  are called its *pins* in PaToH. The *size* of a net,  $s[n]$ , is equal to the number of its pins. The *degree* of a vertex is equal to the number of nets it is connected to. Graph is a special instance of hypergraph such that each net has exactly two pins. Vertices and nets of a hypergraph can be associated with weights. For simplicity in the presentation,

net weights are referred as *cost* here and denoted with  $c[\cdot]$ , whereas  $w[\cdot]$  will be used for vertex weights.

$\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \dots, \mathcal{V}_K\}$  is a *K-way partition* of  $\mathcal{H}$  if the following conditions hold:

- Each part  $\mathcal{V}_k$  is a nonempty subset of  $\mathcal{V}$ , that is,  $\mathcal{V}_k \subseteq \mathcal{V}$  and  $\mathcal{V}_k \neq \emptyset$  for  $1 \leq k \leq K$ .
- Parts are pairwise disjoint, that is,  $\mathcal{V}_k \cap \mathcal{V}_\ell = \emptyset$  for all  $1 \leq k < \ell \leq K$ .
- Union of  $K$  parts is equal to  $\mathcal{V}$ , that is,  $\bigcup_{k=1}^K \mathcal{V}_k = \mathcal{V}$ .

In a partition  $\Pi$  of  $\mathcal{H}$ , a net that has at least one pin (vertex) in a part is said to *connect* that part. *Connectivity*  $\lambda_n$  of a net  $n$  denotes the number of parts connected by  $n$ . A net  $n$  is said to be *cut (external)* if it connects more than one part (i.e.,  $\lambda_n > 1$ ), and *uncut (internal)* otherwise (i.e.,  $\lambda_n = 1$ ). In a partition  $\Pi$  of  $\mathcal{H}$ , a vertex is said to be a *boundary* vertex if it is incident to a cut net. A *K-way partition* is also called a *multiway partition* if  $K > 2$  and a *bipartition* if  $K = 2$ . A partition is said to be balanced if each part  $\mathcal{V}_k$  satisfies the *balance criterion*:

$$W_k \leq W_{avg}(1 + \varepsilon), \quad \text{for } k = 1, 2, \dots, K. \quad (1)$$

In (1), weight  $W_k$  of a part  $\mathcal{V}_k$  is defined as the sum of the weights of the vertices in that part (i.e.,  $W_k = \sum_{v \in \mathcal{V}_k} w[v]$ ),  $W_{avg}$  denotes the weight of each part under the perfect load balance condition (i.e.,  $W_{avg} = (\sum_{v \in \mathcal{V}} w[v])/K$ ), and  $\varepsilon$  represents the predetermined maximum imbalance ratio allowed.

The set of external nets of a partition  $\Pi$  is denoted as  $\mathcal{N}_E$ . There are various [13, 27] *cutsizes* definitions for representing the cost  $\chi(\Pi)$  of a partition  $\Pi$ . Two relevant definitions are:

$$(a) \quad \chi(\Pi) = \sum_{n \in \mathcal{N}_E} c[n] \quad \text{and} \\ (b) \quad \chi(\Pi) = \sum_{n \in \mathcal{N}_E} c[n](\lambda_n - 1). \quad (2)$$

In (2a), the cutsize is equal to the sum of the costs of the cut nets. In (2b), each cut net  $n$  contributes  $c[n](\lambda_n - 1)$  to the cutsize. The cutsize metrics given in (2a) and (2b) will be referred to here as *cut-net* and *connectivity* metrics, respectively. The hypergraph partitioning problem can be defined as the task of dividing a hypergraph into two or more parts such that the cutsize is minimized, while a given balance criterion (1) among part weights is maintained.

A recent variant of the above problem is the *multi-constraint hypergraph* partitioning [2, 6, 10, 19, 24] in which each vertex has a vector of weights associated

with it. The partitioning objective is the same as above, and the partitioning constraint is to satisfy a balancing constraint associated with each weight. Let  $w[v, i]$  denote the  $C$  weights of a vertex  $v$  for  $i = 1, \dots, C$ . Then balance criterion (1) can be rewritten as:

$$W_{k,i} \leq W_{avg,i} (1 + \epsilon) \text{ for } k = 1, \dots, K \text{ and } i = 1, \dots, C, \quad (3)$$

where the  $i$ th weight  $W_{k,i}$  of a part  $\mathcal{V}_k$  is defined as the sum of the  $i$ th weights of the vertices in that part (i.e.,  $W_{k,i} = \sum_{v \in \mathcal{V}_k} w[v, i]$ ), and  $W_{avg,i}$  is the average part weight for the  $i$ th weight (i.e.,  $W_{avg,i} = (\sum_{v \in \mathcal{V}} w[v, i]) / K$ ), and  $\epsilon$  again represents allowed imbalance ratio.

Another variant is the *hypergraph partitioning with fixed vertices*, in which some of the vertices are fixed in some parts before partitioning. In other words, in this problem, a *fixed-part* function is provided as an input to the problem. A vertex is said to be *free* if it is allowed to be in any part in the final partition, and it is said to be fixed in part  $k$  if it is required to be in  $\mathcal{V}_k$  in the final partition  $\Pi$ .

## Using PaToH

PaToH provides a set of functions to read, write, and partition a given hypergraph, and evaluate the quality of a given partition. In terms of partitioning, PaToH provides a user customizable hypergraph partitioning via multilevel partitioning scheme. In addition, PaToH provides hypergraph partitioning with fixed cells and multi-constraint hypergraph partitioning.

Application developers who would like to use PaToH can either directly use PaToH through a simple, easy-to-use C library interface in their applications, or they can use stand-alone executable.

## PaToH Library Interface

PaToH library interface consists of two files: a header file `patoh.h` which contains constants, structure definitions, and functions proto-types, and a library file `libpatoh.a`.

Before starting to discuss the details, it is instructive to have a look at a simple C program that partitions an input hypergraph using PaToH functions. The program is displayed in Fig. 1. The first statement is a function call to read the input hypergraph file which is given by the first command line

argument. PaToH partition functions are customizable through a set of parameters. Although the application user can set each of these parameters one by one, it is a good habit to call PaToH function `PaToH_Initialize_Parameters` to set all parameters to one of the three preset default values by specifying `PATOH_SUGPARAM_<preset>`, where `<preset>` is `DEFAULT`, `SPEED`, or `QUALITY`. After this call, the user may prefer to modify the parameters according to his/her need before calling `PaToH_Alloc`. All memory that will be used by PaToH partitioning functions is allocated by `PaToH_Alloc` function, that is, there will be no more dynamic memory allocation inside the partitioning functions. Now, everything is set to partition the hypergraph using PaToH's multilevel hypergraph partitioning functions. A call to `PaToH_Partition` (or `PaToH_MultiConst_Partition`) will partition the hypergraph, and the resulting partition vector, part weights, and cutsizewill be returned in the parameters. Here, variable `cut` will hold the cutsizew of the computed partition according to cutsizew definition (2b) since, this metric is specified by initializing the parameters with constant `PATOH_CONPART`. The user may call partitioning functions as many times as he/she wants before calling function `PaToH_Free`. There is no need to reallocate the memory before each partitioning call, unless either the hypergraph or the desired customization (like changing coarsening algorithm, or number of parts) is changed.

A hypergraph and its representation can be seen in Fig. 2. In the figure, large circles are cells (vertices) of the hypergraph, and small circles are nets. `xpins` and `pins` arrays store the beginning index of pins (cells) connected to each net, and IDs of the pins, respectively. Hence, `xpins` is an array of size equal to the number of nets plus one (11 in this example), and `pins` is an array of size equal to the number of pins in the hypergraph (31 in this example). Cells connected to net  $n_j$  are stored in `pins[xpins[j]]` through `pins[xpins[j+1]-1]`.

## Stand-Alone Program

Distribution includes a stand-alone program, called `patoh`, for single constraint partitioning (this executable will not work with multiple vertex weights; for multi-constraint partitioning there is an interface and some sample source codes). The program `patoh` gets

its parameters from command line arguments. PaToH can be run from command line as follows:

```
> patch <hypergraph-file>
<number-of-parts> [[parameter1]
[parameter2]].
```

Partitioning can be customized by using the optional [parameter] arguments. The syntax of these optional parameters is as follows: two-letter abbreviation of a parameter is followed by an equal sign and a value. For example, if the user wishes to change refinement algorithm (abbreviated as “RA”) to “Kernighan–Lin with dynamic locking” (sixth algorithm out of 12 implemented in PaToH), the user should specify “RA=6.” For a complete example, consider the sample hypergraph displayed in Fig. 2. In order to partition this hypergraph into three parts by using the Kernighan–Lin refinement algorithm with cut-net metric (the default is connectivity metric (Equation (2b)), one has to issue the following command whose output is shown next:

```
> patch sample.u 3 RA=6 UM=U

+++++
+++ PaToH v3 (c) Nov 1999-, by Umit V. Catalyurek
+++ Build # 872 Date: Fri, 09 Oct 2009
++++

Hypergraph : sample.u #Cells : 12 #Nets : 11 #Pins : 31

3-way partitioning results of PaToH:

Cut Cost: 2
Part Weights : Min= 4 (0.000) Max= 4 (0.000)

I/O : 0.000 sec
I.Perm/Cons.H: 0.000 sec (2.9%)
Coarsening : 0.000 sec (1.1%)
Partitioning : 0.000 sec (75.8%)
Uncoarsening : 0.000 sec (3.7%)
Total : 0.001 sec
Total (w I/O): 0.001 sec

```

This output shows that the cutsize (cut cost) according to cut-net metric is 2. Final imbalance ratios (in parentheses) for the least loaded and the most loaded parts are 0% (perfect balance with four vertices in each part), and partitioning only took about 1 ms. The input hypergraph and resulting partition is displayed in Fig. 3. A quick summary of the input file format (the details are provided in the PaToH manual [9]) is as follows: the first non-comment line of the file is a header containing the index base (0 or 1) and the size of the hypergraph, and information for each net (only pins in this case) and cells (none in this example) follows.

All of the PaToH customization parameters that are available through library interface are also available as command line options. PaToH manual [9] contains details of each of those customization parameters.

## Customizing PaToH’s Hypergraph Partitioning

PaToH achieves  $K$ -way hypergraph partitioning through recursive bisection (two-way partition), and at each bisection step it uses a multilevel hypergraph bisection

---

```

#include <stdio.h>
#include "patch.h"

int main(int argc,char *argv[])
{
 PaToH_Parameters args;
 int _c, _n, _nconst, *cwghts, *nwghts,
 *xpins, *pins, *partvec, cut, *partweights;

 PaToH_Read_Hypergraph(argv[1], &_c, &n, &_nconst, &cwghts, &nwghts,
 &xpins, &pins);

 printf("Hypergraph %10s -- #Cells=%6d #Nets=%6d #Pins=%8d #Const=%2d\n",
 argv[1], _c, _n, xpins[_n], _nconst);

 PaToH_Initialize_Parameters(&args, PATOH_CONPART, PATOH_SUGPARAM_DEFAULT);

 args._k = atoi(argv[2]);
 partvec = (int *) malloc(_c*sizeof(int));
 partweights = (int *) malloc(args._k*sizeof(int));

 PaToH_Alloc(&args, _c, _n, _nconst, cwghts, nwghts, xpins, pins);

 if (_nconst==1)
 PaToH_Partition(&args, _c, _n, cwghts, nwghts,
 xpins, pins, partvec, partweights, &cut);
 else
 PaToH_MultiConst_Partition(&args, _c, _n, _nconst, cwghts,
 xpins, pins, partvec, partweights, &cut);

 printf("%d-way cutsize is: %d\n", args._k, cut);

 free(cwghts); free(nwghts);
 free(xpins); free(pins);
 free(partweights); free(partvec);

 PaToH_Free();
 return 0;
}

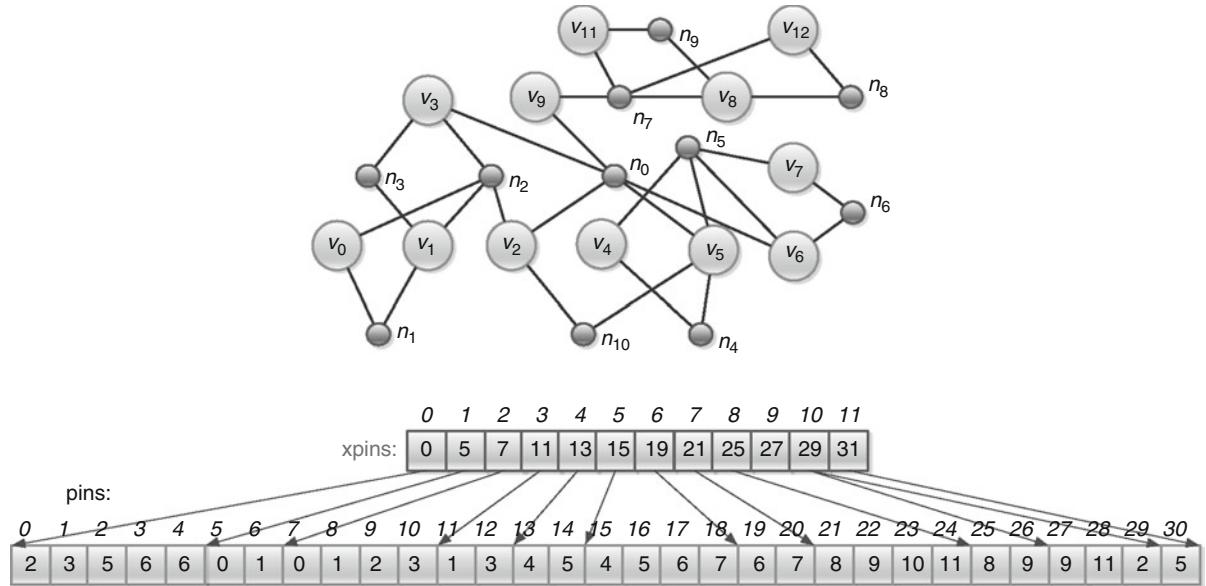
```

---

**PaToH (Partitioning Tool for Hypergraphs).** Fig. 1 A simple C program that partitions an input hypergraph using PaToH functions

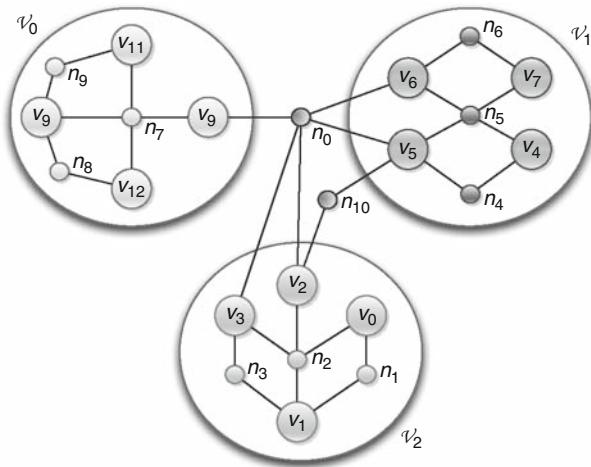
algorithm. In the recursive bisection, first a bisection of  $\mathcal{H}$  is obtained, and then each part of this bipartition is further partitioned recursively. After  $\lg_2 K$  steps, hypergraph  $\mathcal{H}$  is partitioned into  $K$  parts. Please note that,  $K$  is not restricted to be a power of 2. For any  $K > 1$ , one can achieve  $K$ -way hypergraph partitioning through recursive bisection by first partitioning  $\mathcal{H}$  into two parts with a load ratio of  $\lfloor K/2 \rfloor$  to  $(K - \lfloor K/2 \rfloor)$ , and then recursively partitioning those parts into  $\lfloor K/2 \rfloor$  and  $(K - \lfloor K/2 \rfloor)$  parts, respectively, using the same approach.

A pseudo-code of the multilevel hypergraph bisection algorithm used in PaToH is displayed in Algorithm 1. Mainly, the algorithm has three phases: *coarsening*, *initial partitioning*, and *uncoarsening*. In the first phase, a bottom-up multilevel clustering is successively applied starting from the original hypergraph until either the number of vertices in the coarsened hypergraph reduces below a predetermined threshold value or clustering fails to reduce the size of the hypergraph significantly. In the second phase, the coarsest



PaToH (Partitioning Tool for Hypergraphs). Fig. 2 A sample hypergraph and its representation

```
% base:(0/1) #cells #nets #pins
0 12 11 31
% pins of each net in the hypergraph
2 3 5 6 9
0 1
0 1 2 3
1 3
4 5
4 5 6 7
6 7
8 9 10 11
8 10
8 11
a 2 5
```



PaToH (Partitioning Tool for Hypergraphs). Fig. 3 Text file representation of the sample hypergraph in Fig. 2 and illustration of a partition found by PaToH

hypergraph is bipartitioned using one of the 12 initial partitioning techniques. In the third phase, the partition found in the second phase is successively projected back towards the original hypergraph while it is being improved by one of the iterative refinement heuristics. These three phases are summarized below.

**1. Coarsening Phase:** In this phase, the given hypergraph  $\mathcal{H} = \mathcal{H}_0 = (\mathcal{V}_0, \mathcal{N}_0)$  is coarsened into a sequence of smaller hypergraphs  $\mathcal{H}_1 = (\mathcal{V}_1, \mathcal{N}_1)$ ,  $\mathcal{H}_2 = (\mathcal{V}_2, \mathcal{N}_2)$ , ...,  $\mathcal{H}_\ell = (\mathcal{V}_\ell, \mathcal{N}_\ell)$  satisfying  $|\mathcal{V}_0| > |\mathcal{V}_1| > |\mathcal{V}_2| > \dots > |\mathcal{V}_\ell|$ . This

coarsening is achieved by coalescing disjoint subsets of vertices of hypergraph  $\mathcal{H}_i$  into *clusters* such that each cluster in  $\mathcal{H}_i$  forms a single vertex of  $\mathcal{H}_{i+1}$ . The weight of each vertex of  $\mathcal{H}_{i+1}$  becomes equal to the sum of its constituent vertices of the respective cluster in  $\mathcal{H}_i$ . The net set of each vertex of  $\mathcal{H}_{i+1}$  becomes equal to the union of the net sets of the constituent vertices of the respective cluster in  $\mathcal{H}_i$ . Here, multiple pins of a net  $n \in \mathcal{N}_i$  in a cluster of  $\mathcal{H}_i$  are contracted to a single pin of the respective net  $n' \in \mathcal{N}_{i+1}$  of  $\mathcal{H}_{i+1}$ . Furthermore, the single-pin

**Algorithm 1** Multilevel Bisection.

---

```

function PAToHMLLEVELPARTITION($\mathcal{H} = (\mathcal{V}, \mathcal{N})$)
 $\mathcal{H}_0 \leftarrow \mathcal{H}$
 $\ell \leftarrow 0$
 /* Coarsening Phase: */
 while $|V_\ell| > CoarseTo$ do
 find a clustering, \mathcal{C}_ℓ , using one of the coarsening
 algorithms
 construct $H_{\ell+1}$ using \mathcal{C}_ℓ
 if $(|V_{\ell+1}| - |V_\ell|)/|V_\ell| < CoarsePercent$ then
 break
 else
 $\ell \leftarrow \ell + 1$
 end if
 end while
 /* Initial Partitioning Phase: */
 find an initial partitioning Π_ℓ of \mathcal{H}_ℓ
 /* Uncoarsening Phase: */
 while $\ell > 0$ do
 refine Π_ℓ using one of the refinement algorithms
 if $\ell > 0$ then
 project Π_ℓ to $\Pi_{\ell-1}$
 end if
 $\ell \leftarrow \ell - 1$
 end while
 return Π_0
end function

```

---

nets obtained during this contraction are discarded. The coarsening phase terminates when the number of vertices in the coarsened hypergraph reduces below the predetermined number or clustering fails to reduce the size of the hypergraph significantly.

In PaToH, two types of clusterings are implemented, *matching-based*, where each cluster contains at most of two vertices; and *agglomerative-based*, where clusters can have more than two vertices. The former is simply called *matching* in PaToH, and the latter is called *clustering*.

The matching-based clustering works as follows. Vertices of  $\mathcal{H}_i$  are visited in a user-specified order (could be random, degree sorted, etc.). If a vertex  $u \in \mathcal{V}_i$  has not been matched yet, one of its unmatched *adjacent* vertices is selected according to a criterion. If such a vertex  $v$  exists, the matched pair  $u$  and  $v$  are merged into a cluster. If there is no unmatched adjacent vertex of

$u$ , then vertex  $u$  remains unmatched, that is,  $u$  remains as a singleton cluster. Here, two vertices  $u$  and  $v$  are said to be adjacent if they share at least one net, that is,  $nets[u] \cap nets[v] \neq \emptyset$ .

In the agglomerative clustering schemes, each vertex  $u$  is assumed to constitute a singleton cluster  $C_u = \{u\}$  at the beginning of each coarsening level. Then, vertices are again visited in a user specified order. If a vertex  $u$  has already been clustered (i.e.,  $|C_u| > 1$ ) it is not considered for being the source of a new clustering. However, an unclustered vertex  $u$  can choose to join a multi-vertex cluster as well as a singleton cluster. That is, all adjacent vertices of an unclustered vertex  $u$  are considered for selection according to a criterion. The selection of a vertex  $v$  adjacent to  $u$  corresponds to including vertex  $u$  to cluster  $C_v$  to grow a new multi-vertex cluster  $C_u = C_v = C_v \cup \{u\}$ .

PaToH includes a total of 17 coarsening algorithms: eight matchings and nine clustering algorithms, and the default method is a clustering algorithm that uses *absorption* metric. In this method, when selecting the adjacent vertex  $v$  to cluster with vertex  $u$ , vertex  $v$  is selected to maximize  $\sum_{n \in nets[u] \cap nets[C_v]} \frac{|C_v \cap n|}{s[n]-1}$ , where  $nets[C_v] = \bigcup_{w \in C_v} nets[w]$ .

**2. Initial Partitioning Phase:** The goal in this phase is to find a bipartition on the coarsest hypergraph  $\mathcal{H}_\ell$ . PaToH includes various random partitioning methods as well as variations of *Greedy Hypergraph Growing* (GHG) algorithm for bisecting  $\mathcal{H}_\ell$ . In GHG, a cluster is grown around a randomly selected vertex. During the coarse of the algorithm, the selected and unselected vertices induce a bipartition on  $\mathcal{H}_\ell$ . The unselected vertices connected to the growing cluster are inserted into a priority queue according to their *move-gain* [15], where the gain of an unselected vertex corresponds to the decrease in the cutsize of the current bipartition if the vertex moves to the growing cluster. Then, a vertex with the highest gain is selected from the priority queue. After a vertex moves to the growing cluster, the gains of its unselected adjacent vertices that are currently in the priority queue are updated and those not in the priority queue are inserted. This cluster growing operation continues until a predetermined bipartition balance criterion is reached. The quality of this algorithm is sensitive to the choice of the initial random vertex. Since the coarsest hypergraph  $\mathcal{H}_\ell$  is small, initial partitioning heuristics can be run multiple times and select the best bipartition for refinement during the uncoarsening

phase. By default, PaToH runs 11 different initial partitioning algorithms and selects the bipartition with lowest cost.

**3. Uncoarsening Phase:** At each level  $i$  (for  $i = \ell, \ell-1, \dots, 1$ ), bipartition  $\Pi_i$  found on  $\mathcal{H}_i$  is projected back to a bipartition  $\Pi_{i-1}$  on  $\mathcal{H}_{i-1}$ . The constituent vertices of each cluster in  $\mathcal{H}_{i-1}$  is assigned to the part of the respective vertex in  $\mathcal{H}_i$ . Obviously,  $\Pi_{i-1}$  of  $\mathcal{H}_{i-1}$  has the same cutsize with  $\Pi_i$  of  $\mathcal{H}_i$ . Then, this bipartition is refined by running a KL/FM-based iterative improvement heuristics on  $\mathcal{H}_{i-1}$  starting from initial bipartition  $\Pi_{i-1}$ . PaToH provides 12 refinement algorithms that are based on the well-known Kernighan–Lin (KL) [20] and Fiduccia–Mattheyses (FM) [15] algorithms. These iterative algorithms try to improve the given partition by either swapping vertices between parts or moving vertices from one part to other, while not violating the balance criteria. They also provide heuristic mechanisms to avoid local minima. These algorithms operate on passes. In each pass, a sequence of unmoved/unswapped vertices with the highest *gains* are selected for move/swaps, one by one. At the end of a pass, the maximum prefix subsequence of moves/swaps with the maximum prefix sum that incurs the maximum decrease in the cutsize is constructed, allowing the method to jump over local minima. The permanent realization of the moves/swaps in this maximum prefix subsequence is efficiently achieved by rolling back the remaining moves at the end of the overall sequence. The overall refinement process in a level terminates if the maximum prefix sum of a pass is not positive.

PaToH includes original KL and FM implementations, hybrid versions, like one pass FM followed by one pass KL, as well as improvements like *multilevel-gain* concept of Krishnamurthy [21] that adds a lookahead ability, or *dynamic locking* of Hoffman [17], and Dasdan and Aykanat [14] that relaxes vertex moves allowing a vertex to be moved multiple times in the same pass. PaToH also provides heuristic trade-offs, like *early-termination* in a pass of KL/FM algorithms, or *boundary KL/FM*, which only considers vertices that are in the boundary, to speed up the refinement. The default refinement scheme is boundary FM+KL.

## Related Entries

► [Chaco](#)

► [Data Distribution](#)

► [Graph Algorithms](#)

► [Graph Partitioning](#)

► [Hypergraph Partitioning](#)

► [Linear Algebra, Numerical](#)

► [Preconditioners for Sparse Iterative Methods](#)

## Bibliographic Notes and Further Reading

Latest PaToH binary distributions, including recently developed MATLAB interface [26], and related papers can be found on the Web site listed in [9]. The “Hypergraph Partitioning” entry contains some use cases of hypergraph partitioning.

## Bibliography

1. Alpert CJ, Kahng AB (1995) Recent directions in netlist partitioning: a survey. *VLSI J* 19(1–2):1–81
2. Aykanat C, Cambazoglu BB, Uçar B (May 2008) Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices. *J Parallel Distrib Comput* 68(5):609–625
3. Aykanat C, Pinar A, Çatalyürek UV (2004) Permuting sparse rectangular matrices into block-diagonal form. *SIAM J Sci Comput* 26(6):1860–1879
4. Bui TN, Jones C (1993) A heuristic for reducing fill-in sparse matrix factorization. In: Proceedings of the 6th SIAM conference on parallel processing for scientific computing, Norfolk, Virginia, pp 445–452
5. Catalyürek U, Boman E, Devine K, Bozdag D, Heaphy R, Riesen L (Aug 2009) A repartitioning hypergraph model for dynamic load balancing. *J Parallel Distrib Comput* 69(8):711–724
6. Çatalyürek UV (1999) Hypergraph models for sparse matrix partitioning and reordering. Ph.D. thesis, Bilkent University, Computer Engineering and Information Science, Nov 1999. <http://www.cs.bilkent.edu.tr/tech-reports/1999/ABSTRACTS.1999.html>.
7. Çatalyürek UV, Aykanat C (Dec 1995) A hypergraph model for mapping repeated sparse matrixvector product computations onto multicomputers. In: Proceedings of international conference on high performance computing
8. Çatalyürek UV, Aykanat C (1999) Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans Parallel Distrib Syst* 10(7):673–693
9. Çatalyürek UV, Aykanat C (1999) PaToH: a multilevel hypergraph partitioning tool, version 3.0. Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. PaToH. <http://bmi.osu.edu/~umit/software.html>, 1999 (accessed on November 26, 2010)
10. Çatalyürek UV, Aykanat C (2001) A hypergraph-partitioning approach for coarse-grain decomposition. In: ACM/EEE SC2001, Denver, CO, November 2001
11. Çatalyürek UV, Aykanat C, Kayaaslan E (2009) Hypergraph partitioning-based\_ll-reducing ordering. Technical Report

OSUBMI-TR-2009-n02 and BU-CE-0904, The Ohio State University, Department of Biomedical Informatics and Bilkent University, Computer Engineering Department, 2009. submitted for publication

12. Çatalyürek UV, Aykanat C, Ucar B (2010) On two-dimensional sparse matrix partitioning: models, methods, and a recipe. *SIAM J Sci Comput* 32(2):656–683
13. Cheng C-K, Wei Y-C (1991) An improved two-way partitioning algorithm with stable performance. *IEEE Trans Comput Aided Des* 10(12):1502–1511
14. Dasdan A, Aykanat C (February 1997) Two novel multiway circuit partitioning algorithms using relaxed locking. *IEEE Trans Comput Aided Des* 16(2):169–178
15. Fiduccia CM, Mattheyses RM (1982) A linear-time heuristic for improving network partitions. In: Proceedings of the 19th ACM/IEEE design automation conference, pp 175–181
16. Hendrickson B, Leland R (1993) A multilevel algorithm for partitioning graphs. Technical reports, Sandia National Laboratories
17. Hoffmann A (1994) Dynamic locking heuristic – a new graph partitioning algorithm. In: Proceedings of IEEE international symposium on circuits and systems, pp 173–176
18. Karypis G, Kumar V (1998) A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J Sci Comput* 20(1):359–392
19. Karypis G, Kumar V (1998) Multilevel algorithms for multi-constraint graph partitioning. Technical Report 98-019, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, MN 55455, May 1998
20. Kernighan BW, Lin S (Feb 1970) An efficient heuristic procedure for partitioning graphs. *Bell Syst Tech J* 49(2):291–307
21. Krishnamurthy B (May 1984) An improved min-cut algorithm for partitioning VLSI networks. *IEEE Trans Comput* 33(5):438–446
22. Lengauer T (1990) Combinatorial algorithms for integrated circuit layout. Wiley-Teubner, Chichester, UK
23. Sanchis LA (Jan 1989) Multiple-way network partitioning. *IEEE Trans Comput* 38(1):62–81
24. Schloegl K, Karypis G, Kumar V (2000) Parallel multilevel algorithms for multi-constraint graph partitioning. In: Euro-Par, pp 296–310
25. Schweikert DG, Kernighan BW (1972) A proper model for the partitioning of electrical circuits. In: Proceedings of the 9th ACM/IEEE design automation conference, pp 57–62
26. Uçar B, Çatalyürek ÜV, Aykanat C (2010) A matrix partitioning interface to PaToH in MATLAB. *Parallel Comput* 36(5–6):254–272
27. Wei Y-C, Cheng C-K (July 1991) Ratio cut partitioning for hierarchical designs. *IEEE Trans Comput Aided Des* 10(7):91–921

## PC Clusters

### ►Clusters

## PCI Express

JASMIN AJANOVIC  
Intel Corporation, Portland, OR, USA

### Synonyms

3GIO; PCI-Express; PCIe; PCI-E

### Definition

PCI (Peripheral Component Interconnect) Express is a highly scalable interconnect technology that is the most widely adopted IO interface standard used in the computer and communication industry [2]. By providing scalable speed/width, extendable protocol capabilities, a common configuration/software model, and various mechanical form-factors, PCI Express supports a broad range of applications. It allows implementation of flexible connectivity between a processor/memory complex and an IO subsystems, including peripheral controllers, such as graphics, networking, storage, etc. PCI Express technology development is managed by PCI-SIG (PCI Special Interest Group), an industry association comprising of over 800 member companies.

### Discussion

#### Introduction – A Brief History of PCIe

PCI Express has its roots in Peripheral Component Interconnect (PCI), an open standard specification that was developed by the computing industry in 1992. PCI was a replacement for the ISA bus which was a mainstream PC architecture IO expansion standard at the time. Although there were several alternative solutions, such as MicroChannel, EISA, and VL-bus, that were aiming to replace/supplement ISA, none of them fully addressed the needs of an evolving PC industry. The PCI specification covered both the hardware and software interfaces between PC's CPU/memory complex and add-in cards, such as graphics, network, and disk controllers. One of the most important aspects of PCI was support for the so called “plug-and-play” mechanisms

## Partitioning Tool for Hypergraphs (PaToH)

### ►PaToH (Partitioning Tool for Hypergraphs)

which allowed operating system software to detect installed hardware components, including both add-in cards and motherboard-down devices, configure system resources required for their operations, including memory address ranges and interrupts, and install appropriate software device drivers. From a hardware perspective, PCI was initially defined as a 32-bit multiplexed address/data bus that was shared among multiple devices attached to it, operating at 5 Volt and at speed of 33 MHz. Over time, faster variants of PCI evolved to support ever increasing performance requirements of CPUs which rose operational frequencies from 66 MHz in 1993 to over 3 GHz by 2003. These variants include:

- 66 MHz 32- and 64-bit PCI operating at 5 V or 3 V.
- AGP (Accelerated Graphic Port), a graphics-optimized interface operating at a common clock speed of 66 MHz and carrying data transfers at 2x, 4x, and 8x of that nominal speed resulting in maximum bandwidth of 0.5, 1, and 2 GB/s over the 32-bit interface.
- PCI-X, a server-optimized solution operating at clock speeds of 66, 100, and 133 MHz and resulting in maximum bandwidth of 0.5, 0.8, and 1 GB/s over the 64-bit interface.

Towards the end of the twentieth century, neither of these solutions proved to be an adequate answer to emerging applications that required higher scalability (performance and protocol capabilities) and a more efficient interface (pins, bandwidth, and power). In 2001, Intel Corporation, together with several other industry leaders, spearheaded the development of the next generation IO architecture. Initially called 3GIO [1], this architecture was later endorsed by the PCI-SIG as PCI Express (PCIe).

Appearing in systems starting in 2003, PCI Express was rapidly adopted by the PC/PCI ecosystem replacing entirely AGP and PCI-X, and in some instances PCI, within 1 or 2 product generations. The key to the PCI Express success was in leveraging of PCI base architecture by supporting full backwards compatibility at the software level while providing a more optimized interconnect solution. The PCI Express physical interface is based on width- and speed-scalable, point-to-point,

differential serial signaling technology operating initially at 2.5 GT/s (Giga Transfers/second). As fast as the new PCI Express standard was, processor architectures have continued to accelerate to meet the demands of emerging applications, and PCI Express needed to keep up. For an example, increasing bandwidth was needed to improve the performance of data-intensive graphics workloads as well as server-class storage and network solutions. PCI Express continued to improve by scaling-up speed as well adding architectural/protocol extensions. [Figure 1](#) shows the evolution of PCI Express technology.

## Technology Overview

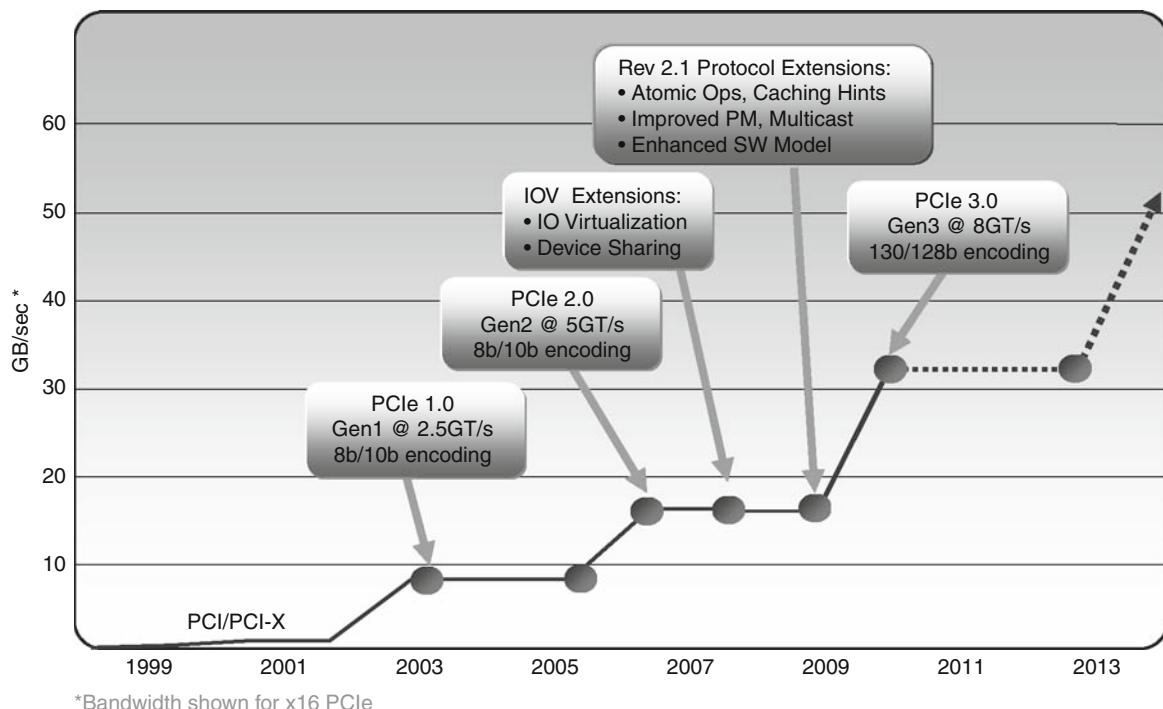
### Basic Elements and Concepts

#### Link and Lane

PCI Express is a point-to-point interconnect technology where Link represents a fundamental element of PCIe-based interconnect fabric. A basic Link is a dual-simplex communications channel between two components that represents a single Lane. Lane consists of two low-voltage, differentially driven signal pairs: a Transmit pair and a Receive pair as shown in [Fig. 2](#). To aggregate the bandwidth of a PCIe Link, multiple Lanes can be grouped together to provide a “wider” Link. In PCIe terminology, Link width is expressed using xN (“by N”) denotation, where N is the number of Lanes that form the Link. PCIe architecture specification defines operations for x1, x2, x4, x8, x12, x16, and x32 Link widths. Companion form-factor specifications (add-in card and connector) define operations for a subset of operational widths (x1, x4, x8, and x16).

#### Signaling, Speed, and Bandwidth

To carry communication over the Link, PCIe uses Low-Voltage Differential Signaling (LVDS) with embedded clocking. Embedded clocking is a mechanism where clock information is embedded within the transmitted data by providing a guaranteed number of transitions between “1’s” and “0’s” that are required to correctly extract the clock on the receiver side. Revision 1.0 (Gen1) and Revision 2.0 (Gen2) of PCIe Specification use standard 8b/10b encoding scheme [3] which is a common mechanism for a number of industry standard interfaces based on serial signaling technology (e.g., Infiniband, Fibre Channel, SATA, etc.). This scheme



**PCI Express. Fig. 1** PCI express technology roadmap

comes with an overhead of 20% because it uses 10bits to carry 8bits of actual information. Since it is relatively simple and very robust, 8b/10b was used by PCIe as a reasonable tradeoff between efficiency and complexity allowing PCIe Gen1 and Gen2 to hit target effective bandwidths while operating at moderately high speeds of 2.5 GT/s and 5 GT/s respectively. However, continuing with 2x increase of operational speed proved to be difficult from a technology enabling/adoptability and ecosystem perspective. For PCIe 3.0 (Gen 3), the PCI-SIG defined a new more efficient 128b/130b encoding scheme, which with ~1% overhead effectively doubles the bandwidth of 5 GT/s Gen2 while operating only at 8 GT/s instead of 10 GT/s (Note that 20% overhead of 8b/10b brings effective data transfers of Gen2 from 5 GT/s down to 4 GT/s.) The following Table 1 shows raw and effective speeds of all three generations of PCIe, including a total bandwidth based on an example of x16 PCIe link.

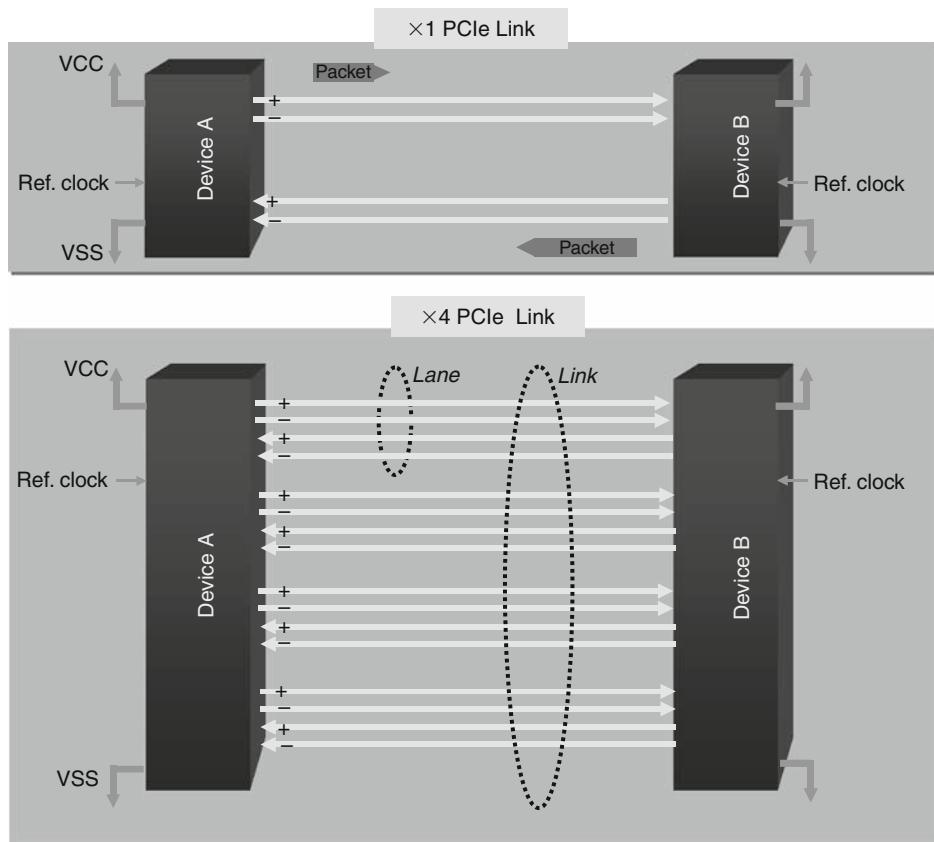
To support backwards compatibility at the system and component/add-in card level, PCI Express Specification requires newer generation devices that operate at higher speeds to support operations at

lower speed. Examples: Gen2 PCIe device operating at nominal 5 GT/s must support operations at 2.5 GT/s, Gen3 device must support operations at both 5 GT/s and 2.5 GT/s.

P

### Link Configuration

PCI Express architecture allows devices that support different Link widths and speeds to be configured for proper operation. The negotiation of width and speed is done as a part of Link initialization process where two devices exchange information about their capabilities using a lowest common denominator method to determine the operational width and speed. For an example, if Device A that supports x8 width at 5.0 GT/s is connected to Device B that supports x4 width at 2.5 GT/s, the Link that connects them will be initialized for x4 width operating at 2.5 GT/s. Note that PCIe allows only configurations of the devices with symmetric Link width. Link width/speed configuration occurs at the Link hardware level without any involvement of software.



**PCI Express. Fig. 2** PCI express link examples - x1 and x4

**PCI Express. Table 1** PCI express speeds and bandwidths

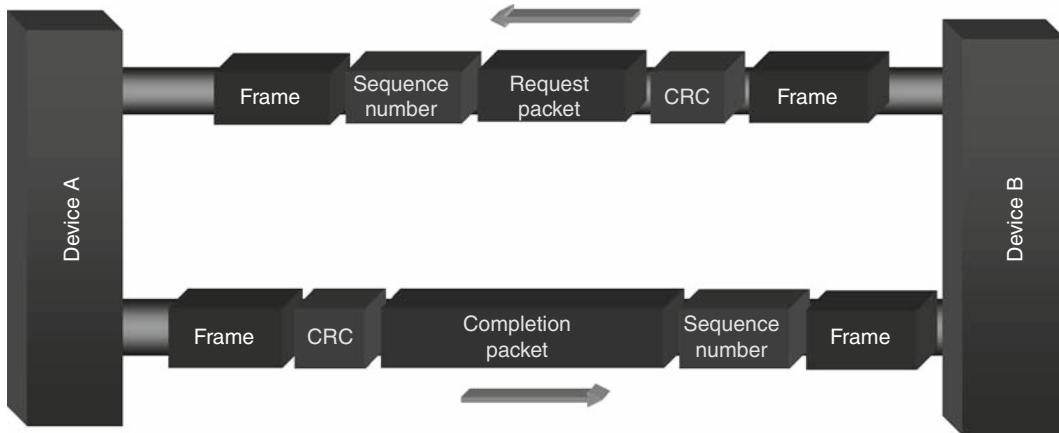
| PCIe Generation | Raw bit rate | Effective bit rate | Bandwidth per lane<br>Per direction | Total bandwidth*<br>For x16 link |
|-----------------|--------------|--------------------|-------------------------------------|----------------------------------|
| PCIe 1.x        | 2.5 GT/s     | 2 Gb/s             | ~250 MB/s                           | ~8 GB/s                          |
| PCIe 2.0        | 5.0 GT/s     | 4 Gb/s             | ~500 MB/s                           | ~16 GB/s                         |
| PCIe 3.0        | 8.0 GT/s     | 8 Gb/s             | ~1 GB/s                             | ~32 GB/s                         |

\*Total bandwidth represents the aggregate interconnect bandwidth in both directions

Once Link is initialized and configured for proper width/speed operations, it may be re-configured during the run-time as a result of a RAS (Reliability, Serviceability, Availability) event due to, for example, data integrity problems. Speed and/or width can be reduced in an attempt to correct the problem and keep the system running with reduced performance/functionalities. Note that the PCIe Specification defines a mechanism where software, through access to configuration/-control registers, can override the established hardware configuration of the Link and force Link to operate at lower speed than nominally capable.

#### Packet-Based Protocol

Instead of using dedicated signals for address, control, and data (such as its predecessor PCI), PCIe uses packets to communicate information between components connected to the PCIe fabric. Packets contain all of the information related to transactions such as: source and target identification/address, type (e.g., Memory Read, Memory Write, Message), attributes (e.g., Isochronous, No Snoop), and data. In addition to this information, Link hardware logic inserts additional information required for correct packet transfer across the Link such as framing, sequencing and packet



**PCI Express. Fig. 3** Packet-based protocol

integrity protection (CRC). Packets can be differentiated as Request and Response packets. To complete the entire transaction (e.g., Memory Read), a single Request packet may cause one or multiple Response packets. Note that some request transaction types (e.g., Message) do not require responses (Fig. 3).

### PCI Express Layering Overview

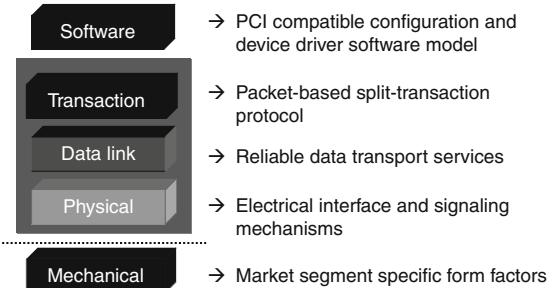
The PCIe interface stack is defined using a layered approach with PCIe Transaction, Data Link, and Physical Layers being formally defined in PCIe Base Specification and the rest of the stack related to Software and Mechanical being covered in companion documents. The following simplified Fig. 4 shows PCIe interface stack and highlights major aspects of each layer.

#### Transaction Layer

The Transaction Layer is responsible for assembly (for transmit) and disassembly (upon receive) of Transaction Layer Packets (TLPs). PCIe supports load-store as well as message-based transactions, where TLPs are used to communicate transactions such as reads, writes, messages, and events using different type of semantics (Memory, I/O, Configuration, and Message), address types/formats (32-bit/64-bit, device ID), and attributes (No Snoop, Relaxed Ordering, and ID-Based Ordering).

The Transaction Layer manages flow of packets between transmitting and receiving devices using credit-based flow control scheme.

Instead of using physical pins/wires to support side-band signals, such as interrupts, power-management



**PCI Express. Fig. 4** PCI express interface stack

requests, etc., PCIe uses Messages as “virtual wires” that carry information in-band. This improves overall efficiency of the interface by eliminating large number of pins/wires.

#### Data Link Layer

This layer provides Link management and data integrity, including error detection and error correction. On the transmit side the Data Link Layer calculates and applies a CRC (Cyclic Redundancy Check) data protection code on the TLP submitted by Transaction Layer. It also adds a TLP sequence number, and passes entire packet to Physical Layer for transmission across the Link. On the receive side the Data Link Layer checks the integrity of received TLPs before passing them to the Transaction Layer for further processing. In the case if an error is detected, this Layer requests retransmission of TLPs until information is correctly received, or the Link is determined to have failed.

### Physical Layer

The Physical Layer converts information received from the Data Link Layer into an appropriate serialized format and transmits it across the Link. This Layer consists of Electrical and Logical functional blocks. Electrical block includes all circuitry required for interface operation: output and input buffers, parallel-to-serial and serial-to-parallel conversion, PLL(s), and transmitter/receiver signal conditioning circuitry. Logical block supports functions required for interface initialization and maintenance, including configuration of Link speed and width.

### Packet Flow Through the Layers

The TLPs are formed in the Transaction Layer to carry the information from the transmitting component to the receiving component. As the transmitted packets flow through the other layers, they are extended with additional information necessary to carry out transfers. At the receiving side the reverse process occurs and packets get transformed from their Physical Layer representation to the Data Link Layer representation and finally to the form that can be processed by the Transaction Layer of the receiving device. [Figure 5](#) shows the conceptual flow of transaction -level packet information through the layers.

Note that for the purpose of Link management, a simpler form of packet communication is supported between two Data Link Layers that are connected to the same Link. These packets are referred to as Data Link Layer Packets (DLLP).

### PCI Express Platform Examples

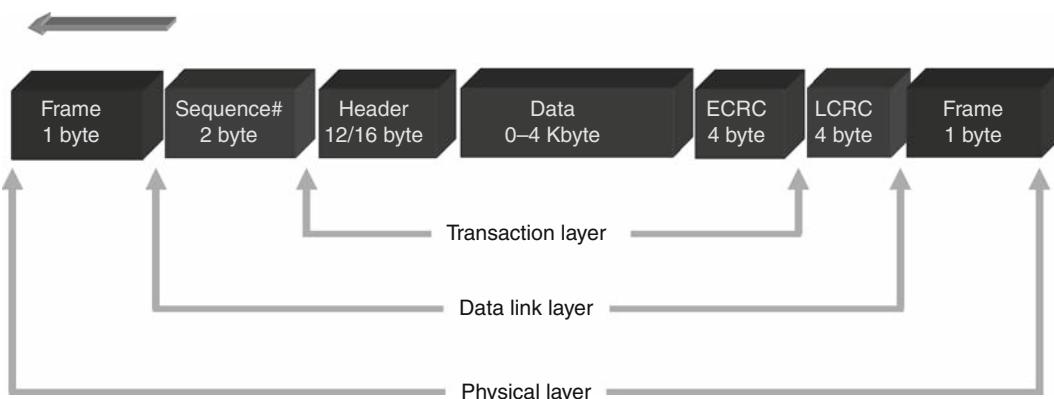
The PCIe architecture supports a variety of platform configurations by allowing the mixing and matching of PCIe link speeds and widths to support different applications. [Figure 6](#) shows examples of PCIe -based client and server computer platforms.

To support high-performance graphics applications, client/workstation computer platforms typically utilize x16 PCIe. Lower bandwidth functions such as network adapters typically use x1 PCIe. Various additional functions (e.g., enhanced audio) can be provided using add-in card slots. Note that an IO Bridge shown on a client platform in [Fig. 6](#) also provides support for some legacy functions, such as old 5 V/33 MHz PCI bus that is required until the technology transition completes.

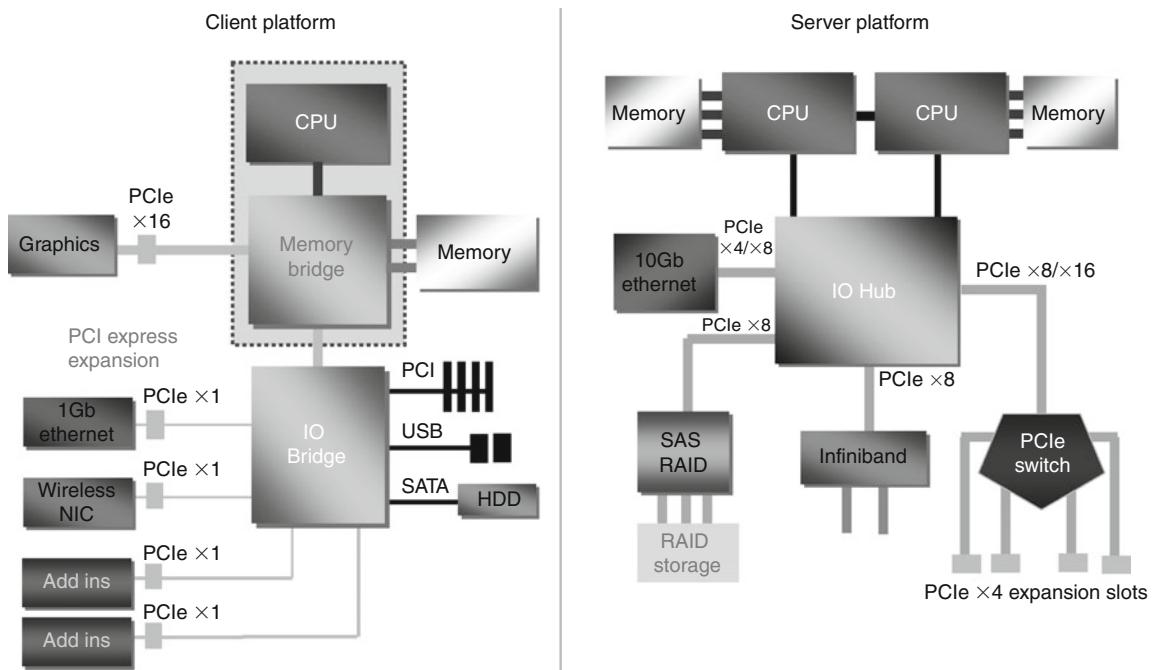
Server platforms typically use x4 and x8 PCIe ports that are required by the high-bandwidth application such as high-performance network (10GigE) and storage (SAS) adapters, Infiniband, Fibre Channel, etc. An important building block for servers is PCIe Switch which is architecturally supported by the PCIe Specification. PCIe Switch is typically used for expansion of IO subsystem by providing additional PCIe interface ports.

### Architecture Features

PCI Express is a scalable architecture that provides a highly flexible and expandable set of capabilities. This section highlights essential features of PCIe.



PCI Express. [Fig. 5](#) Packet formation



PCI Express. Fig. 6 Examples of PCIe based platforms

### Scalable Protocol

PCIe uses a fully packetized split-transaction protocol as described under the Technology Overview section. In addition to load-store semantics, it provides messaging support that is used by the Virtual Wire mechanism to eliminate side-band signals by converting them into in-band messages. PCIe uses a credit-based scheme to control the flow of packets within the PCIe fabric. This scheme applies per Link, and it is used to prevent buffer overflow. To minimize the dependency between different traffic flows in a system, PCIe provides Virtual Channel (VC) mechanism and transaction ordering relaxations. These mechanisms mitigate the overhead of a strict producer-consumer ordering model (that may create head-of-line blocking conditions) and allow support for differentiated Quality of Service at the platform level.

For future scalability of supported packet formats, PCIe defines Transaction Layer Packet Prefix, a mechanism for extending TLP headers.

### PCI Compatible Software Model

PCIe leverages standard mechanisms defined in the PCI Plug-and-Play specification for device initialization, enumeration, and configuration. This allows legacy

software operating systems to boot without modifications on a PCIe-based system. It also preserves compatibility with device driver software model, enabling existing API, and application software to execute unchanged. To remove platform overhead and limitations associated with legacy PCI software configuration, PCIe provides Enhanced Configuration mechanism. This mechanism needs to be supported by newer operating systems to fully take the advantage of PCIe advanced capabilities (e.g., Advanced Error Logging/Reporting, Virtual Channel Mechanism). Enhanced Configuration also allows new capabilities to be added in the future.

### Scalable Performance

Primary factor in PCIe performance scaling as measured by the interface bandwidth is function of interface width and operational frequency and can be expressed using the following formula:

$$\begin{aligned}
 & \text{Total Link Bandwidth [GB/s]} \\
 & = \text{Link_Width[Number of lanes]} \\
 & \quad \times \text{Effective_Signaling_Rate[Gb/s]} \\
 & \quad \times 2[\text{directions}]/8[\text{bits}]
 \end{aligned}$$

GB/s = Giga Byte per second, Gb/s = Giga bit per second, GT/s = Giga Transfer per second

for Gen1 and Gen2: Effective\_Signaling\_Rate[Gb/s] = Raw\_Signaling\_Rate[GT/s] × 0.8

for Gen3: Effective\_Signaling\_Rate [Gb/s] ≈ Raw\_Signaling\_Rate [GT/s]

Secondary factor of PCIe performance is related to the fact that PCIe is point-to-point interconnect with two unidirectional signaling paths which are contention-free i.e., do not require arbitration. This means that in systems that contain multiple PCIe Links, traffic on each Link can occur independently and simultaneously which improves total system throughput. An additional contributor to performance is related to PCIe -pin/bandwidth efficiency. By improving this aspect, PCIe allows a tighter IO integration within the platform, e.g., direct PCIe attach to high-integration CPUs. This can result in lower IO system latencies in an optimized system.

Note that the above formula shows only theoretical bandwidths, but that actual performance as seen at the application level depends on number of factors. These factors include PCIe -interface and system implementation aspects such as: supported data payload size, interface-level data buffering/queuing and effectiveness of flow-control, arbitration mechanisms throughout the platform (e.g., competing for host memory access), power -management mechanisms and policies, software device driver and API overheads, etc.

## Advanced Power Management

PCIe defines Link level power -management scheme including the active-state power -management (ASPM) protocol. The ASPM manages power -state transitions on the Link transparently to run-time software by detecting idle conditions on the Link, i.e., when no actual data is being communicated over the Link. Note that in signaling schemes that use embedded clocking, data needs to be transmitted continuously to maintain synchronization between transmitter and receiver. To reduce the power consumption during “idle,” PCIe defines low-power link states. These states save power but transitions into and out of them require recovery time to resynchronize the transmitter and receiver which potentially may impact the Link latency and affect the overall system performance. In addition to

Link level, power-management PCIe defines related system level power-management mechanisms such as:

- Latency Tolerance Reporting – reduces platform power based on PCIe device service requirements.
- Opportunistic Buffer Flush and Fill – aligns PCIe -device activity with platform power-management events to further reduce platform power.
- Dynamic Power Allocation–dynamic control of power/thermal budget per PCIe device.

## Reliability, Availability, Serviceability (RAS) Support

RAS capabilities are defined to: ensure data integrity, provide ability to identify/manage errors, and allow installation/removal of components in the running system without requiring operating system shutdown.

- Data Integrity – As a basic/required mechanism, PCIe defines a data integrity scheme where 32-bit CRC (Cyclic Redundancy Check) is used to protect packets transmitted over the single Link. For platforms that use more complex topologies (i.e., with PCIe Switches) and require end-to-end data integrity, PCIe defines an optional end-to-end 32-bit CRC. This CRC code is used in addition to Link local 32-bit CRC to protect the data in high -reliability server applications.
- Hot Plug and Hot Swap – PCIe defines native support for hot plugging or hot swapping of IO cards/modules. Solutions based on PCIe hot plug/ swap address requirements of both server and portable computer platforms and support industry standard software stacks for platform management and configuration.
- Advanced Error Reporting/Handling – PCIe defines support for error logging/reporting to improve system -fault isolation and enable recovery solutions.

## Differentiated Quality of Service (QoS) Support

Using Virtual Channel (VC) mechanism as a foundation, PCIe supports eight levels of QoS differentiated traffic including isochronous traffic type. PCIe specification defines VC-system configuration and programmable-VC arbitration mechanism necessary to support an end-to-end solution designed for applications, such as isochronous that require real-time delivery of voice and video related data.

## IO Virtualization and Device Sharing Support

PCIe defines an IO interface-level mechanism for support of platform virtualization. In virtualized platforms, a single instance of platform hardware is mapped in to a multiple independent Virtual Machines, each running their own operating system and application software stacks. In addition to PCI Express Base Specification, the PCI-SIG maintains a set of specifications that cover PCIe support for:

- Address Translation Services: allow PCIe devices to obtain copies of translated system addresses to mitigate address translation latencies in IO Virtualized platform.
- Single-Root IO Virtualization and Device Sharing: improves system performance scaling by allowing a single PCIe device to be presented to the software as multiple independent/pseudo-independent hardware devices.
- Multi-Root IO Virtualization: allows multiple independent PCIe-based platforms (such as in blade-server systems) to overlap/share system resources.

Note that IO Virtualization capabilities require new software, both at the PCIe fabric configuration/management level as well as at the system level.

## Support for Heterogeneous Processing and Application Acceleration

There is a set of capabilities that are provided to support high-performance applications which require efficient co-processing and data sharing, such as GP-GPU, computational accelerators, and network/storage accelerators. These include:

- Transaction Layer Packet Processing Hints – Request hints from PCIe device to enable optimized processing within host memory/cache hierarchy.
- Atomic Read-Modify-Write Transactions – Reduce synchronization overhead for shared data structures.
- Address Based Multicast – Allows significant gains in efficiency compared to multiple unicast transactions by carrying transaction between the single source and multiple destinations.

## Form-Factors

In addition to chip-chip connectivity usage, PCI Express supports a variety of system board and add-in

card form-factors. To address diversity of applications across many segments (desktop and mobile PCs, servers, embedded/communications, etc.), the following add-in form-factors are defined:

- PCIe Card Electromechanical (CEM) used for desktops, workstations, and servers.
- PCIe Mini Card used for portable computers.
- PCIe Module also known as Server IO Module (SIOM) optimized for server/communication systems.
- ExpressCard used for portable computers and small form-factor desktops.
- PCIe External Cabling Spec used for embedded/communication.

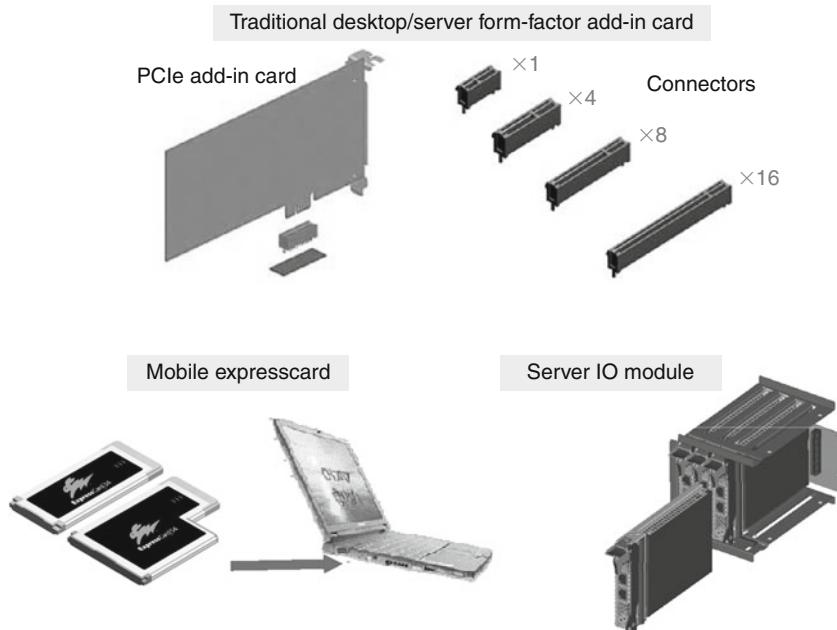
[Figure 7](#) shows a sample of form-factors including the most common PCIe CEM add-in card specified by the PCI-SIG. This card is defined to fit into the traditional desktop PC and server systems and comes with different connector widths (x1, x4, x8, and x16). It supports power usage from baseline 25 W up to 300 W in x16 variant used for high-end graphics and similar applications.

In addition to the PCI-SIG, there are several other industry associations, such as PCMCIA and PICMG that have been involved in development of PCI Express based form-factors.

## PCI Express Today

Products based on first generation (2.5 GT/s) and second generation (5 GT/s) of PCIe technologies are deployed broadly across computer and communication/embedded industries. When you consider this in light of the PCI-SIG's 800 members, many of them very active, the PCIe product development and users community represents a large ecosystem. According to a report from an industry analyst organization, In-Stat [2], the size of market of components that include PCIe interface will exceed 440 million units/year by 2010.

PCI-SIG is marching towards the milestone of releasing a third generation of PCI Express spec, i.e., PCIe Rev 3.0 in second half of 2010. In addition to 2.5 and 5 GT/s, PCIe 3.0 supports 8 GT/s operational speed by using a new, more efficient signaling encoding scheme which will allow doubling of bandwidth compared to the prior generation. In addition, the new generation of PCIe products based on Rev 3.0 will support enhanced power management as well as features for



PCI Express. Fig. 7 Example of PCIe form-factors

application performance acceleration through: atomic read-modify-write, data reuse/caching hints, multicast operations, etc.

### Future Directions

There are two very significant trends within the mainstream computer and communication industry that will influence the evolution of PCI Express technology:

- Increasing level of integrated functionality resulting in very high-integration single die devices, known as SoC (System on a Chip), or high-integration multi-die components/modules, known as SiP (System in a Package).
- Evolution of Internet with ever increasing need for improved computational, storage, and communication performance of the main building blocks that provide the service.

First trend is mainly driven by the applications that dictate small form-factors, low-power, and low-cost. They range from smart-phones and entertainment devices to the components used in embedded control applications such as home appliances, cars, printers, etc. For some of these products, levels of integration will result in removal of external interfaces such as PCIe and for

the others, external interfaces will be still needed, but with a requirement for significantly improved power efficiency. This may dictate evolution of PCIe technology in two directions:

- On-die interconnects that are architecturally equivalent to PCIe and allow integration of PCIe devices in the form of IP (Intellectual Property) building blocks.
- Lower power and lower speed variants of PCIe.

Note that significant value of PCIe ecosystem comes from the ability to support standard operating system software, device drivers, and applications. On-die interconnects that are an architectural equivalent (not necessarily physical/electrical equivalent) of PCIe, will enable integration of hardware functionality without the requirement to change the software stack. This may result in huge R&D savings and faster time-to-market for SoC/SiP designs that use PCIe IP building blocks.

Second trend, evolution and expansion of Internet, will translate into a requirement to the computer industry to provide new generations of communication

servers and data centers that are capable of handling increasing and diverse workloads. This will drive a need for hardware and software optimizations for virtualization and cloud computing. Evolution of traditional rack-servers and blade-servers may open room for a new interconnect backbone/fabric based on PCIe technology. This will drive the following enhancements to the PCIe technology:

- Enhanced routing and configuration to support more elaborate topologies (besides PCI/PCIe hierarchical tree) to allow system scalability through aggregation of large number of discrete devices as well as high-integration devices (that consist of multiple logical devices).
- Capability for tunneling and mapping other protocols (e.g., for storage, networking, system management, etc.) which will allow consolidation of interconnect technologies within the rack/blade and perhaps even a data center.
- Doubling/multiplying bandwidth needed to support next generation of Internet, HPC (High-Performance Computing), and cloud computing. This will eventually result in emergence of PCIe Optical connectivity starting with discrete solutions at speeds ranging from 16GT/s to 25 GT/s and evolving to 50 GT/s silicon-photonics-based integrated solution.

Evolution of PCI Express will likely track the future path of the computer industry. New IO technologies will continue to emerge in the future, using some elements of PCI Express as its foundation.

## Related Entries

- [Bandwidth-Latency Models \(BSP, LoGP\)](#)
- [Benchmarks](#)
- [Buses and Crossbars](#)
- [Collective Communication](#)
- [Computer Graphics](#)
- [Data Centers](#)
- [Deadlocks](#)
- [Ethernet](#)
- [Flow Control](#)
- [InfiniBand](#)
- [Interconnection Networks](#)
- [Network Interfaces](#)

- [Routing \(Including Deadlock Avoidance\)](#)
- [Switch Architecture](#)
- [Switching Techniques](#)
- [Synchronization](#)

## Bibliographic Notes and Further Reading

As mentioned in the introduction, development of PCI Express technology as an industry standard is being coordinated through PCI Special Interest Group (PCI-SIG). This organization maintains the website with officially published specification documents and other collaterals that are part of PCI Express standard. For detailed information go to:

- PCI-SIG: [www.pcisig.com](http://www.pcisig.com).

Note that, although some documents are available to a general audience, membership with the PCI-SIG is required for non-restricted access to all documents and specifications managed by the SIG.

In addition to PCI-SIG, there are several other industry organizations, such as:

- PCMCIA: [www.pcmcia.org](http://www.pcmcia.org)
- PICMG: [www.picmg.org](http://www.picmg.org)

that manage development of complementary industry standards based on PCIe Base Specification. These organizations enhance PCI Express technology by defining additional form-factors and design collaterals that allow even broader adoption of PCI Express component-level hardware products as well as companion software.

Among the component and platform vendors that provide significant additional support for PCI Express technology is Intel Corporation with: Intel® Developer Network for PCI Express\* Architecture. This forum provides members with technical data, marketing support and industry connections needed to accelerate the innovation and marketing of PCI Express based solutions. For more details see:

- PCIe DevNet: [http://www.intel.com/technology/pci\\_express/devnet/](http://www.intel.com/technology/pci_express/devnet/)

There are several technical books that provide overviews of PCI Express technology as well as detailed implementation guidelines. See references [4–6].

## Bibliography

1. Intel white paper. Creating a Third Generation I/O Interconnect. [www.intel.com/technology/pciexpress/devnet/docs/WhatisPCIExpress.pdf](http://www.intel.com/technology/pciexpress/devnet/docs/WhatisPCIExpress.pdf)
2. In-Stat – In Depth Analysis ([www.in-stat.com](http://www.in-stat.com)): I/O, I/O, Changing the Status Quo: Chip-to-Chip Interconnects, February 2, 2007, <http://www.instat.com/newmk.asp?ID=1909>
3. Widmer AX, Franaszek PA (1983) A DC-balanced, partitioned-block 8B/10B transmission code. IBM J Res Dev 27(5):440–451
4. Budruk R, Anderson D, Shanley T (2003) PCI express system architecture. Mindshare, Colorado
5. Wilen A, Schade JP, Thornburg R (2003) Introduction to PCI express: a hardware and software developer's guide Intel Hillsboro
6. Solari Ed, Congdon B, Clark D (2003) The complete PCI express reference: design implications for hardware and software developers (Engineer to Engineer series). Intel, Hillsboro
7. OpenSystems Media – Articles: PCI Express. <http://www.opensystems-publishing.com/articles/search/?topic=PCI+Express>
8. Compact PCI Systems. <http://www.compactpci-systems.com>
9. DSP-FPGA.com – Articles, Videos and White Papers: PCI Express. <http://www.dsp-fpga.com/articles/search/index.php?mag=&max=10&op=ew&q=pcie&skip=10>
10. Embedded Systems. [www.embedded.com](http://www.embedded.com)
11. PCI Express Architecture Frequently Asked Questions, PCI-SIG. [http://www.pcisig.com/news\\_room/faqs/faq\\_express/](http://www.pcisig.com/news_room/faqs/faq_express/)
12. PCI Express External Cabling 1.0 Specification. [http://www.pcisig.com/specifications/pcieexpress/pcie\\_cabling1.0/](http://www.pcisig.com/specifications/pcieexpress/pcie_cabling1.0/)
13. PCI-SIG Announces PCI Express 3.0 Bit Rate For Products In 2011 And Beyond. 8 Aug 2007. [http://www.pcisig.com/news\\_room/08\\_08\\_07/](http://www.pcisig.com/news_room/08_08_07/)
14. PHY Interface for the PCI Express Architecture, version 2.00 (PDF). [http://download.intel.com/technology/pciexpress/devnet/docs/pipe2\\_00.pdf](http://download.intel.com/technology/pciexpress/devnet/docs/pipe2_00.pdf)
15. PCI Express 3.0 Frequently Asked Questions, PCI-SIG. [http://www.pcisig.com/news\\_room/faqs/pcie3.0\\_faq/](http://www.pcisig.com/news_room/faqs/pcie3.0_faq/)

## PCIe

► [PCI Express](#)

## PCI-E

► [PCI Express](#)

## PCI-Express

► [PCI Express](#)

## Peer-to-Peer

STEFAN SCHMID<sup>1</sup>, ROGER WATTENHOFER<sup>2</sup>

<sup>1</sup>Telekom Laboratories/TU Berlin, Berlin, Germany

<sup>2</sup>ETH Zürich, Zurich, Switzerland

## Synonyms

Distributed hash table (DHT); Overlay network; Decentralization; Open distributed systems; Consistent hashing

## Definition

The term *peer-to-peer* (p2p) is ambiguous, and is used in a variety of different contexts, such as:

- In popular media coverage, p2p is often synonymous to software or protocols that allow users to “share” files (music, software, books, movies, etc.). p2p file sharing is very popular and a large fraction of the total Internet traffic is due to p2p.
- In academia, the term p2p is used mostly in two ways. A narrow view essentially defines p2p as the “theory behind file-sharing protocols.” In other words, how do Internet hosts need to be organized in order to deliver a search engine to find (share) content (files) efficiently? A popular term is “distributed hash table” (DHT), a distributed data structure that implements such a content search engine. A DHT should support at least a search (for a key) and an insert(key, object) operation. A DHT has many applications beyond file sharing, e.g., the Internet domain name system (DNS).
- A broader view generalizes p2p beyond file sharing: Indeed, there is a growing number of applications operating outside the juridical gray area, e.g., p2p Internet telephony à la Skype, p2p mass player games, p2p live audio and video streaming as in PPLive, StreamForge or Zattoo, or p2p social storage and cloud computing systems such as Wuula. Trying to account for the new applications beyond file sharing, one might define p2p as a large-scale distributed system that operates without a central server bottleneck. However, with this definition almost “everything decentralized” is p2p!
- From a different viewpoint, the term p2p may also be synonymous for privacy protection, as various

p2p systems such as Freenet allow publishers of information to remain anonymous and uncensored.

In other words, there is no single well-fitting definition of p2p, as some definitions in use today are even contradictory. In the following, an academic viewpoint is assumed (second and third definition above).

## Discussion

### The Paradigm

At the heart of p2p computing lies the idea that each network participant serves both as a producer (“server”) and consumer (“client”) of services. Depending on the application, the shared resources can be data (files), CPU power, disk storage, or network bandwidth. Often p2p systems have an open clientele, and do not rely on the availability of specific individual machines; rather they can deal with dynamic resources and do not exhibit single points of failure or bottlenecks.

Compared to centralized solutions, the p2p paradigm features a better scalability because the amount of resources grows with the network size, availability (avoiding a single point of failure), reliability, fairness, cooperation incentives, privacy, and security – just about everything researchers expect from a future Internet architecture. As such, it is not surprising that new “clean slate” Internet architecture proposals often revolve around p2p concepts.

One might naively assume that for instance scalability is not an issue in today’s Internet, as even most popular web pages are generally highly available. However, this is not necessarily due to our well-designed Internet architecture, but rather due to the help of so-called overlay networks: The Google Web site for instance manages to respond so reliably and quickly because Google maintains a large distributed infrastructure, essentially a p2p system. Similarly, companies like Akamai sell “p2p functionality” to their customers to make today’s user experience possible in the first place. Quite possibly today’s p2p applications are just testbeds for tomorrow’s Internet architecture.

### Implications

p2p networks are often highly dynamic in nature. While traditional computer systems are typically based on fixed infrastructures and are under a single administrative domain (e.g., owned and maintained by a single

company or corporation), the participating machines in p2p networks are under the control of individual (and to some extent: anonymous) users who can join and leave at any time and concurrently. In p2p parlance, such membership changes are called *churn*.

A second implication of the autonomy of the machines in p2p networks is that the network consists of different stakeholders. Users can have various reasons for joining the network. For instance, an (anonymous) user may not voluntarily contribute his or her bandwidth, disk space, or CPU cycles to the system, but prefer to *free ride*. This adds a socioeconomic aspect to p2p computing. As the p2p paradigm relies on the contributions of the participating machines, effective incentive mechanisms have to be designed, which foster cooperation and punish free riders.

Another source of inequality in p2p systems apart from selfishness is *heterogeneity*: Due to the open membership, different machines run different operating systems, have different Internet connections, and so on.

### Applications

The best-known representatives of p2p technology are probably the numerous file-sharing applications such as Napster, Gnutella, KaZaA, eMule, or BitTorrent. Also, the Internet telephony tool Skype is very popular and used by millions everyday. Zattoo, PPLive, and StreamForge, among many others, use p2p principles to stream video or audio content. The cloud computing service Wuala offers free online storage by exploiting the participants’ disks and Internet connections to improve performance. Recently, the power and anonymity of decentralized Internet working has gained the attention of operators of botnets in order to attack certain infrastructure components by a denial-of-service attack. Finally, p2p technology is used for large-scale computer games.

### Architecture Variants

Several p2p architectures are known:

- Client/Server goes p2p: Even though Napster is known to be the first p2p system (1999), by today’s standards its architecture would not deserve the label p2p anymore. Napster clients accessed a central server that managed all the information of the shared files, i.e., which file was to be found on

- which client. Only the downloading process itself was between clients (“peers”) directly, hence p2p. In the early days of Napster the load of the server was relatively small, so the simple Napster architecture was sufficient. Over time, it turned out that the server may become a bottleneck – and an attractive target for an attack. Indeed, eventually a judge ruled the server to be shut down (a “juridical denial of service attack”). However, it remains to note that many popular P2P networks today still include centralized components, e.g., KaZaA or the eDonkey network accessed by the eMule client. Also, the peer swarms downloading the same file in the BitTorrent network are organized by a so-called tracker whose functionality today is still centralized (although initiatives exist to build distributed trackers).
- **Unstructured p2p:** The Gnutella protocol is the antithesis of Napster, as it is a fully decentralized system, with no single entity having a global picture. Instead each peer connects to a random sample of other peers, constantly changing the neighbors of this virtual overlay network by exchanging neighbors with neighbors of neighbors. (Any unstructured system also needs to solve the so-called *bootstrap problem*, namely how to discover a first neighbor in a decentralized manner. A popular solution is the use of well-known peer lists.) The fact that users often turn off their clients once they downloaded their content implies high levels of churn (peers joining and leaving at high rates), and hence selecting the right “random” neighbors is an interesting research problem. The Achilles’ heel of unstructured p2p architectures such as Gnutella is the cost of searching. A search request is typically flooded in the network and each search operation will cost  $m$  messages,  $m$  being the number of virtual edges in the architecture. In other words, such an unstructured p2p architecture will not scale. Indeed, when Napster was unplugged, Gnutella broke down as well soon afterward due to the inrush of former Napster users.
  - **Hybrid p2p:** The synthesis of client/server architectures such as Napster and unstructured architectures such as Gnutella are hybrid architectures. Some powerful peers are promoted to so-called superpeers (or, similarly, trackers). The set of superpeers may change over time, and taking down

a fraction of superpeers will not harm the system. Search requests are handled on the superpeer level, resulting in much less messages than in flat/homogeneous unstructured systems. Essentially, the superpeers together provide a more fault-tolerant version of the Napster server, as all regular peers connect to a superpeer. As of today, almost all popular p2p systems have such a hybrid architecture, carefully trading off reliability and efficiency.

- **Structured p2p:** Inspired by the early success of Napster, the academic world started to look into the question of efficient file sharing. Indeed, even earlier, in 1997, Plaxton et al. [34] proposed a hypercubic architecture for p2p systems. This was a blueprint for many so-called structured p2p architecture proposals, such as Chord [46], CAN [36], Pastry [37], Tapestry [50], Viceroy [26], Kademia [27], Koord [15], SkipGraph [3], and SkipNet [11]. Maybe surprisingly, in practice, structured p2p architectures did not take off yet, apart from certain exceptions such as the Kad architecture (from Kademia [27]), which is accessible with the eMule client.

## Scientific Origins

The scientific foundations of p2p computing were laid many years before the most simple “real” p2p systems like Napster emerged. As already mentioned, in 1997, a blueprint for structured systems has been proposed in [34]. Indeed, also the [34] paper was standing on the shoulders of giants. Some of its eminent precursors are the following:

- Research on linear and consistent hashing, e.g., [16].
- Research on locating shared objects, e.g., [4] or [5].
- Research on so-called compact routing: The idea is to construct routing tables such that there is a trade-off between memory (size of routing tables) and stretch (quality of routes), e.g., [31] or [49].
- Even earlier, hypercubic networks, see below.

## Hypercubic Overlays and Consistent Hashing

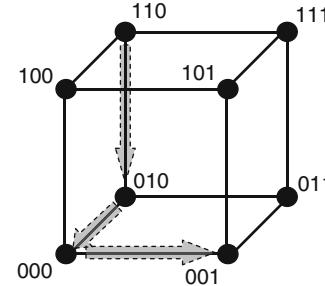
Every application run on multiple machines needs a mechanism that allows the machines to exchange information. A naive solution is to store at each machine the domain name or IP address of every other machine.

While this may work well for a small number of machines, large-scale distributed applications such as file sharing, grid computing, cloud computing, or data center networking systems need a different, more scalable approach: instead of forming a clique (where everybody knows everybody else), each machine should only be required to know some small subset of other machines. This graph of knowledge can be seen as a logical network interconnecting the machines; it is also known as an *overlay network*. A prerequisite for an overlay network to be useful is that it has good topological properties. Among the most important are small peer degree, small network diameter, robustness to churn, or absence of congestion bottlenecks.

The most basic network topologies used in practice are trees, rings, grids, or tori. Many other suggested networks are simply combinations or derivatives of these. The advantage of trees is that the routing is very easy: for every source-destination pair there is only one possible path. However, the root of a tree can be a severe bottleneck. An exception is a p2p streaming system where the single content provider forms the network root. However, trees are also highly vulnerable, e.g., with respect to membership changes.

Essentially all state-of-the-art p2p networks today have some kind of hypercubic topology (e.g., Chord, Pastry, Kademlia). Hypercube graphs have many interesting properties, e.g., they allow for efficient routing: although each peer only needs to store a logarithmic number of other peers in the system (the peers' neighbors), by a simple routing scheme, a peer can reach each other peer in a logarithmic number of steps (or "hops"). In a nutshell, this is achieved by assigning each peer a unique  $d$ -bit identifier. A peer is connected to all  $d$  peers that differ from its identifier at exactly one bit position. In the resulting hypercube network, routing is done by adjusting the bits in which the source and the destination peers differ – one at a time (at most  $d$  many). Thus, if the source and the destination differ by  $k$  bits, there are  $k!$  routes with  $k$  hops. [Figure 1](#) gives an example.

Given a hypercubic topology, it is then simple to construct a distributed hash table (DHT): Assume there are  $n = 2^d$  peers that are connected in a hypercube topology as described above. Now a globally known hash function  $f$  is used, mapping file names to long bit strings. Let  $f_d$  denote the first  $d$  bits (prefix) of



**Peer-to-Peer. Fig. 1** A simplified p2p topology: a three-dimensional hypercube. Each peer has a three-bit identifier. For example, peer 110 is connected to the three peers 010, 100, 111 whose identifiers differ at exactly one position. In order to route a message from peer 110 to say peer 001, one bit is fixed after the other. One possible routing path is depicted in the figure: 110 → 010 → 000 → 001. An alternative path could be 110 → 111 → 101 → 001

the bitstring produced by  $f$ . If a peer is searching for file name  $X$ , it routes a request message  $f(X)$  to peer  $f_d(X)$ . Clearly, peer  $f_d(X)$  can only answer this request if all files with hash prefix  $f_d(X)$  have been previously registered at peer  $f_d(X)$ .

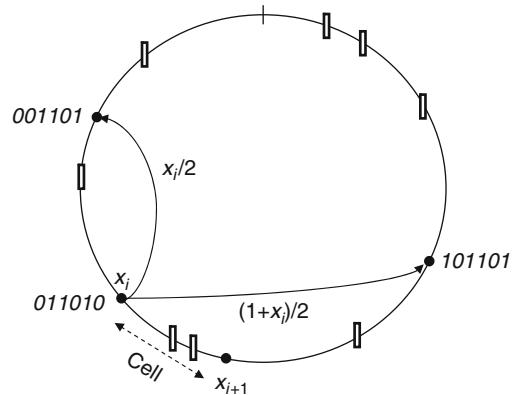
There are some additional issues to be addressed in order to design a DHT from a hypercubic topology, in particular how to allow peers to join and leave without notice. To deal with churn the system needs some level of replication, i.e., a number of peers, which are responsible for each prefix such that failure of some peers will not compromise the system. In addition, there are security and efficiency issues that can be addressed to improve the system.

There are many hypercubic networks that are derived from the hypercube: among these are the butterfly, the cube-connected-cycles, the shuffle-exchange, and the de Bruijn graph. For example, the butterfly graph is basically a "rolled out" hypercube (hence directly providing replication!) of constant degree. Another important class of hypercubic topologies are skip graphs [3, 11].

A simple, interesting way to design dynamic p2p systems is the *continuous-discrete approach* described by Naor and Wieder [29]. This approach is based on a "think continuously, act discretely" strategy, and can be used to design a variety of hypercubic topologies. The continuous-discrete approach gives a unified method

for performing join/leave operations and for dealing with the scalability issue, thus separating it from the actual network. The idea is as follows: Let  $I$  be a Euclidean space, e.g., a (cyclic) one-dimensional space. Let  $G_c$  be a graph where the vertex set is the continuous set  $I$ . Each point in  $I$  is connected to some other points. The actual network then is a discretization of this continuous graph based on a dynamic decomposition of the underlying space  $I$  into cells where each “server” is responsible for a cell. Two cells are connected if they contain adjacent points in the continuous graph. Clearly, the partition of the space into cells should be maintained in a distributed manner. When a join operation is performed, an existing cell splits, when a leave operation is performed two cells are merged into one. The task of designing a dynamic and scalable network follows these design rules: (1) Choose a proper continuous graph  $G_c$  over the continuous space  $I$ . Design the algorithms in the continuous setting, which is often simpler (also in terms of analysis) than in the discrete case. (2) Find an efficient way to discretize the continuous graph in a distributed manner, such that the algorithms designed for the continuous graph would perform well in the discrete graph. The discretization is done via a decomposition of  $I$  into the cells. If the cells that compose  $I$  are allowed to overlap, then the resulting graph would be fault tolerant.

To give an example, in order to build a dynamic *de Bruijn* network (a so-called Distance Halving DHT), a peer at position  $x \in [0,1)$  (in binary form  $b_1 b_2 \dots$  such that  $x = \sum_{i=1}^{\infty} 2^{-b_i}$ ) connects to positions  $l(x) := x/2 \in [0,1)$  and  $r(x) := (1+x)/2 \in [0,1)$  in  $G_c$  (out-degree two per peer). Observe that if position  $x$  is written in binary form, then  $l(x)$  effectively shifts in a “0” from the left and  $r(x)$  shifts in a “1” from the left. Thus, routing is straightforward: based solely on the current position and the destination (without the overhead of maintaining routing tables), a message can be forwarded by a peer by fixing one bit per hop. The set of peers in the cyclic  $[0,1)$  space then define the p2p network: Let  $x_i$  denote the position of the  $i^{th}$  peer (ordered in increasing order with respect to position). Peer  $i$  is responsible for the cell  $[x_i, x_{i+1})$ , computed in a modulo manner, i.e., this peer is responsible to store the data mapped to this cell plus for the establishment of the corresponding connections defined in  $G_c$ . Figure 2 gives an example.



**Peer-to-Peer. Fig. 2** The continuous–discrete approach for the dynamic de Bruijn graph. Peers are indicated using circles, files using rectangles. In the continuous setting, the peer at position  $x_i = 0.011010$  (in binary notation) is connected to positions  $x_i/2$  and  $(1+x_i)/2$ . In the discrete setting, it is responsible for the cell (i.e., the connections and files that are mapped there) between positions  $x_i$  and  $x_{i+1}$

## Dealing with Churn

A distinguishing property of p2p systems are the frequent membership changes. Measuring the churn levels of existing p2p systems is challenging and one has to be careful when generalizing a given measurement to entire application classes (e.g., [10]). Nevertheless, several insightful measurement studies have been conducted. For instance, [9, 38] reported on the dynamic nature of early p2p networks such as Napster and Gnutella, and [41] analyzed low-level data of a large Internet Service Provider (ISP) to estimate churn. Also the Kad DHT has been subject to measurement studies, and the reader is referred to the results in [47] and [43].

It is widely believed that hypercubic structures are a good basis for churn-resilient p2p systems. As written earlier, a DHT is essentially a hypercubic structure with peers having identifiers such that they span the ID space of the objects to be stored. A simple approach to map the ID space onto the peers has already been described for the hypercube. To give another example, in the butterfly network, we may use its layers for replication, i.e., all peers with the same ID redundantly store the data of the same hash prefix. Other hypercubic DHTs can be more difficult to design, e.g., networks based on the pancake graph [19].

For many well-known systems, theoretic analyses exist showing that the networks remain well-structured after some joins, leaves, or failures occur. In order to evaluate the robustness formally, metrics such as the network *expansion* (for deterministic failures) or the *span* [6] (for randomized failures) are used. Unfortunately, the span is difficult to compute, and the span value is known only for the most simple topologies.

The continuous-discrete approach [29] already mentioned constitutes the basis of several dynamic systems. For example, the SHELL system [40] is robust to certain attacks by connecting older or more reliable peers in a core network where access can be controlled; SHELL also allows to organize heterogeneous peers in an efficient topology.

Many systems proposed in the literature offer a high robustness in the average case, i.e., they provide probabilistic guarantees that hold with high probability. Robustness under attacks or worst-case dynamics is less well understood. In [19], a system is developed that achieves an optimal worst-case robustness in the sense that there is no alternative system that can tolerate higher churn rates without disconnecting. The basic idea is to simulate a hypercube: each peer is part of a distinct hypercube *node*; each hypercube node consists of a logarithmic number of peers. Peers have connections to other peers of their hypercube node and to peers of the neighboring hypercube nodes. After a number of joins and leaves, some peers may have to change to another hypercube node such that up to constant factors, all hypercube nodes have the same cardinality at all times. If the total number of peers grows or shrinks above or below a certain threshold, the dimension of the hypercube is increased or decreased by one, respectively. The balancing of peers among the nodes can be seen as a *dynamic token distribution problem* on the hypercube: Each node of a graph (hypercube) has a certain number of tokens, and the goal is to distribute the tokens along the edges of the graph such that all nodes end up with the same or almost the same number of tokens. Thus, the system builds on two basic components: (1) an algorithm, which performs the described dynamic token distribution and (2) an information aggregation algorithm, which is used to estimate the number of peers in the system and to adapt the hypercube's dimension accordingly. These techniques also work for alternative graphs, like pancake graphs [19].

An appealing notion of robustness is *topological self-stabilization*: A p2p topology is called self-stabilizing if it is guaranteed that from any weakly connected initial state (e.g., after an attack), it will quickly converge to a desirable network in the absence of further membership changes. In contrast to the worst-case churn considered in [19], self-stabilization focuses on the convergence time in periods without membership changes, but allows for general initial system states. While until recently, self-stabilizing algorithms with guaranteed runtime have only been known for simple one-dimensional or two-dimensional linearization problems [14], recently a construction for a variation of skip graphs, namely SKIP+ graphs [13], has been proposed. Single joins and leaves in SKIP+ can be handled locally, and require logarithmic time and polylogarithmic work only. However, there remains the important open question of how to provide degree guarantees during convergence from arbitrary states.

## Fostering Cooperation

The appeal of p2p computing arises from the collaboration of the system's constituent parts, the peers. If all the participating peers contribute some of their resources, highly scalable decentralized systems can be built. However, in reality, peers may act selfishly and strive for maximizing their own utility by benefitting from the system without contributing much themselves [42]. Hence, the performance of a p2p system crucially depends on its capability of dealing with selfishness.

Already in 2000, Adar and Huberman [1] noticed that there exists a large fraction of free riders in the file-sharing network Gnutella. The problem of selfish behavior in p2p systems has been a hot topic in p2p research ever since, and many mechanisms to encourage cooperation have been proposed [30]. Perhaps the simplest fairness mechanism is to directly incorporate contribution monitoring into the client software. For instance, in the file-sharing system KaZaA, the client records the contribution of its user. However, such a solution can simply be bypassed by implementing a different client that hard-wires the contribution level to the maximum, as it was the case with *KaZaA Lite*. Inspired by real economies, some researchers have also proposed the introduction of some form of virtual money, which is used for the transactions.

BitTorrent has incorporated a fairness mechanism from the beginning and has hence been subject to intensive research (e.g., [21, 22, 35]). Although this mechanism has similarities to the well-known tit-for-tat scheme, the strategy employed in BitTorrent distinguishes itself from the classic mechanism in many respects. For instance, it is possible for peers to obtain parts of a file “for free,” i.e., without reciprocating. While this may be a useful property for bootstrapping newly joined peers, it has been shown that the BitTorrent mechanism can be exploited: the *BitThief* BitTorrent client [24] allows to download entire files fast without uploading any data. It has also been demonstrated in [24] that sharing communities are particularly vulnerable to such exploits. BitThief is not the only client cheating BitTorrent. Piatek et al. [32] presented *BitTyrant*. BitTyrant’s strategy is to exploit the BitTorrent protocol in order to maximize download rates. For instance, BitTyrant uses a smart neighbor selection strategy and connects to those peers with the best reciprocation ratios. In contrast to BitThief, BitTyrant does not free ride. BitTyrant seeks to provide the minimal necessary contribution, and also increases the active neighbor set if this is beneficial to the download rate. The authors claim that their client provides a median 70% performance gain in certain environments.

There can be many other forms of strategic behavior in open distributed systems. One subject that has recently gained attention, especially by the game-theoretic research community, is *neighbor selection* in unstructured p2p networks (e.g., [28]). There may be several reasons for a peer to prefer connecting to some peers rather than others. For instance, a peer may want to connect to peers with high bandwidths, peers storing many interesting files, or peers having large degrees and hence provide quick access to many other peers. At the same time, a selfish peer itself may not be eager to store and maintain too many neighbors itself.

### Current Trends and Outlook

One can argue that today, p2p computing is already a relatively mature (research) field; nevertheless, there are still many active discussions and developments, also in the context of the future Internet design. Moreover, there exists a discrepancy between the technology of the systems in use and what is actually known in theory.

For example, the Kad network is still vulnerable to quite simple attacks [44].

If employed by the wrong people, the flexibility and robustness of p2p technology also constitutes a threat. Denial-of-service attacks are arguably one of the most cumbersome problems in today’s Internet, and it is appealing to coordinate botnets in a p2p fashion. A DHT can be used by the bots, e.g., to download new instructions. For instance, it was estimated that in 2007, the DHT-based *Storm botnet* [20] ran on several million computers. Apart from mechanisms to detect or prevent attacks even before they take place, a smart redundancy management may improve availability during the attack itself (see, e.g., the Chameleon system [7]).

In terms of cooperation, there is a tension between the goal of providing incentive compatible mechanisms that exclude free riders and the goal of designing heterogeneous p2p systems that also tolerate (and make use of!) weak participants. Moreover, in addition to design mechanisms dealing with pure selfishness, there is a trend toward p2p systems that are also resilient to malicious behavior (see, e.g., [23] or [39]).

Another active discussion regards the interface between p2p systems and ISPs. The large amount of p2p traffic raises the question of how ISPs should deal with p2p, e.g., by caching contents. p2p networks often employ inefficient overlay-to-ISP mappings as the logical overlay network is typically not aware of the underlying “real” networks and constraints, and much overhead can be avoided by improving the interface between p2p networks and ISPs, e.g., by an oracle [2]. For a critical point of view on the subject, the reader is referred to [33].

It seems that while a few years ago the lion’s share of Internet traffic was due to p2p, the proportion seems to be declining [12] now. Especially web services and server-based solutions such as the popular YouTube and RapidShare are catching up. The measured data traces should be interpreted with care however, as they do not take into account what happens behind the scenes of big corporations. Indeed, it is believed that there is a paradigm shift in p2p computing: While p2p retreats (relatively to other applications) from public Internet traffic, today p2p technology plays a crucial role in the coordination and management of large data centers and server farms of corporations such as Akamai or Google.

## Related Entries

► [Hypercubes and Meshes](#)

## Bibliographic Notes and Further Reading

Beyond the specific literature pointed to directly in the text, there are several recommendable introductory books on p2p computing. In particular, the reader is referred to the classic books [8, 45, 48] and two more recent issues [8, 17]. The theoretically more inclined reader may also be interested in [25], which provides an overview of compact routing solutions, and [18] which discusses trade-offs in local algorithms that achieve global goals based on local information only and without centralized entities whatsoever. Regarding the challenges of distributed cooperation, the recent book [30] gives a thorough and up-to-date survey of current (game-theoretic) trends, and also includes a chapter on p2p specific questions.

## Bibliography

1. Adar E, Huberman B (2000) Free riding on gnutella. *First Monday* 5(10):1–22
2. Aggarwal V, Feldmann A, Scheideler C (2007) Can ISPs and p2p users cooperate for improved performance? *ACM Comput Commun Rev* 37(3):29–40
3. Aspnes J, Shah G (2003) Skip graphs. In: Proceedings of the 14th annual ACM-SIAM symposium on discrete algorithms (SODA), Baltimore, 2003
4. Awerbuch B, Peleg D (1990) Sparse partitions. In: Proceedings of the 31st annual symposium on foundations of computer science (SFCS), vol 2, pp 503–513, Washington, 1990
5. Awerbuch B, Peleg D (1995) Online tracking of mobile users. *J ACM* 42(5):1021–1058
6. Bagchi A, Bhargava A, Chaudhary A, Eppstein D, Scheideler C (2004) The effect of faults on network expansion. In: Proceedings of the 16th annual ACM symposium on parallelism in algorithms and architectures (SPAA), Barcelona, 2004
7. Baumgart M, Scheideler C, Schmid S. A DoSresilient information system for dynamic data management. In: Proceedings of the 21st ACM symposium on parallelism in algorithms and architectures (SPAA), Calgary, Alberta, 2009
8. Buford J, Yu H, Lua EK (2008) P2P networking and applications. Morgan Kaufmann, San Francisco, 2008
9. Gummadi K, Dunn R, Saroiu S, Gribble SD, Levy HM, Zahorjan J (2003) Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In: Proceedings of the 19th ACM symposium on operating systems principles (SOSP), Bolton Landing, 2003
10. Haeberlen A, Mislove A, Post A, Druschel P (2006) Fallacies in evaluating decentralized systems. In: Proceedings of the 5th international workshop on peer-to-peer systems (IPTPS), Santa Barbara, 2006
11. Harvey NJA, Jones MB, Saroiu S, Theimer M, Wolman A. Skipnet: a scalable overlay network with practical locality properties. In: Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS), Seattle, 2003
12. IPOQUE (2009) Internet study 2008/2009. <http://www.ipoque.com/resources/internet-studies/internet-study-2008-2009>, pp 704–713 (accessed on October 31, 2010)
13. Jacob R, Richa A, Scheideler C, Schmid S, Täubig H (2009) A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In: Proceedings of the ACM symposium on principles of distributed computing (PODC), New York, 2009
14. Jacob R, Ritscher S, Scheideler C, Schmid S (2009) A self-stabilizing and local delaunay graph construction. In: Proceedings of the 20th international symposium on algorithms and computation (ISAAC), Hawaii, 2009
15. Kaashoek F, Karger DR (2003) Koord: a simple degree-optimal distributed hash table. In: Proceedings of the international workshop on peer-to-peer systems (IPTPS), Berkeley, 2003
16. Karger D, Lehman E, Leighton T, Panigrahy R, Levine M, Lewin D (1997) Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In: Proceedings of the 29th ACM symposium on theory of computing (STOC), New York, pp 654–663, 1997
17. Khan J, Wierzbicki A (2008) Foundation of peer-to-peer computing. Elsevier Computer Communication, 2008
18. Kuhn F, Moscibroda T, Wattenhofer R (2006) The price of being near-sighted. In: Proceedings of the 17th ACM-SIAM symposium on discrete algorithms (SODA), Miami, 2006
19. Kuhn F, Schmid S, Wattenhofer R (2010) Towards worstcase churn resistant peer-to-peer systems. *J Distrib Comput (DIST)* 22(4):249–267
20. Larkin E (2007) Storm worm's virulence may change tactics. British Computer Society (accessed on August 03, 2007)
21. Legout A, Urvoy-Keller G, Michiardi P (2006) Rarest first and choke algorithms are enough. In: Proceedings of the 6th ACM SIGCOMM conference on internet measurement (IMC), pp 203–216, Rio de Janeiro, 2006
22. Levin D, LaCurts K, Spring N, Bhattacharjee B (2008) BitTorrent is an auction: analyzing and improving BitTorrent's incentives. *SIGCOMM Comput Commun Rev* 38(4):243–254
23. Li H, Clement A, Marchetti M, Kapritos M, Robinson L, Alvisi L, Dahlin M (2008) Flighthpath: obedience vs choice in cooperative services. In: Proceedings of the symposium on operating systems design and implementation (OSDI), San Diego, 2008
24. Locher T, Moor P, Schmid S, Wattenhofer R (2006) Free riding in BitTorrent is cheap. In: Proceedings of the 5th Workshop on Hot Topics in Networks (HotNets), Irvine, 2006
25. Malkhi D (2004) Locality-aware network solutions. Technical Report, The Hebrew University of Jerusalem, HUJI-CSE-LTR-2004-6

26. Malkhi D, Naor M, Ratajczak D (2002) Viceroy: a scalable and dynamic emulation of the butterfly. In: Proceedings of the 21st annual symposium on principles of distributed computing (PODC), Monterey, 2002
27. Maymounkov P, Mazières D (2002) Kademia: a peer-to-peer information system based on the xor metric. In: Proceedings of the 1st international workshop on peer-to-peer systems (IPTPS), Cambridge, 2002
28. Moscibroda T, Schmid S, Wattenhofer R (2006) On the topologies formed by selfish peers. In: Proceedings of the 25th annual symposium on principles of distributed computing (PODC), Denver, 2006
29. Naor M, Wieder U (2003) Novel architectures for p2p applications: the continuous-discrete approach. In: Proceedings of the 15th annual ACM symposium on parallel algorithms and architectures (SPAA), pp 50–59, San Diego, 2003
30. Nisan N, Roughgarden T, Tardos E, Vazirani VV (2007) Algorithmic game theory. Cambridge University Press, Cambridge
31. Peleg D, Upfal E (1988) A tradeoff between space and efficiency for routing tables. In: Proceedings of the 20th annual ACM symposium on theory of computing (STOC), pp 43–52, Chicago, 1988
32. Piatek M, Isdal T, Anderson T, Krishnamurthy A, Venkataramani A (2007) Do incentives build robustness in bittorrent? In: Proceedings of the 4th USENIX symposium on networked systems design and implementation, Cambridge, 2007
33. Piatek M, Madhyastha HV, John JP, Krishnamurthy A, Anderson T (2009) Pitfalls for ISP-friendly p2p design. In: Proceedings of the hotnets, New York, 2009
34. Plaxton C, Rajaraman R, Richa AW (1997) Accessing nearby copies of replicated objects in a distributed environment. In: Proceedings of the 9th ACM symposium on parallel algorithms and architectures (SPAA), pp 311, 320, Newport, 1997
35. Qiu D, Srikant R (2004) Modeling and performance analysis of bittorrent-like peer-to-peer networks. In: Proceedings of the conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM), pp 367–378, New York, 2004
36. Ratnasamy S, Francis P, Handley M, Karp R, Schenker S (2001) A scalable content-addressable network. In: Proceedings of the ACM SIG-COMM conference on applications, technologies, architectures, and protocols for computer communications, pp 161–172, New York, 2001
37. Rowstron AIT, Druschel P (2001) Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Proceedings of the IFIP/ACM international conference on distributed systems platforms (middleware), pp 329–350, Heidelberg, 2001
38. Saroiu S, Gummadi PK, Gribble SD (2002) A measurement study of peer-to-peer file sharing systems. In: Proceedings of the multimedia computing and networking (MMCN), San Jose, 2002
39. Scheideler C (2005) How to spread adversarial nodes?: rotate! In: Proceedings of the 37th Annual ACM symposium on theory of computing (STOC), pp 704–713, Baltimore, 2005
40. Scheideler C, Schmid S (2009) A distributed and oblivious heap. In: Proceedings of the 36th international colloquium on automata, languages and programming (ICALP), Rhodes, 2009
41. Sen S, Wang J (2004) Analyzing peer-to-peer traffic across large networks. IEEE/ACM Trans Netw 12(2):219–232
42. Shneidman J, Parkes DC (2003) Rationality and self-interest in peer to peer networks. In: Proceedings of the 2nd international workshop on peer-to-peer systems (IPTPS), Berkeley, 2003
43. Steiner M, Biersack EW, Ennajary T (2007) Actively monitoring peers in KAD. In: Proceedings of the 6th international workshop on peer-to-peer systems (IPTPS), Bellevue, 2007
44. Steiner M, En-Najary T, Biersack EW (2007) Exploiting KAD: possible uses and misuses. Comput Commu Rev 37(5):65–69
45. Steinmetz R, Wehrle K (2005) Peer-to-peer systems and applications. Springer, Heidelberg
46. Stoica I, Morris R, Karger D, Kaashoek F, Balakrishnan H (2001) Chord: a scalable peer-to-peer lookup service for internet applications. In: Proceedings of the ACM SIGCOMM conference on applications, technologies, architectures, and protocols for computer communications, San Diego, 2001
47. Stutzbach D, Rejaie R (2006) Understanding churn in peer-to-peer networks. In: Proceedings of the 6th internet measurement conference (IMC), New York, 2006
48. Subramanian R, Goodman B (2005) Peer-to-peer computing: the evolution of a disruptive technology. IGI, Hershey
49. Thorup M, Zwick U (2001) Compact routing schemes. In: Proceedings of the annual ACM symposium on parallel algorithms and architectures (SPAA), pp 1–10, Crete, Greece, 2001
50. Zhao BY, Huang L, Stribling J, Rhea SC, Joseph AD, Kubiatowicz J (2004) Tapestry: a resilient global-scale overlay for Service deployment. IEEE journal on selected areas in communuincations vol 22, No. 1

## Pentium

► [Intel Core Microarchitecture, x86 Processor Family](#)

## PERCS System Architecture

E. N. (MOOTAZ) ELNOZAHY<sup>1</sup>, EVAN W. SPEIGHT<sup>1</sup>, JIAN LI<sup>1</sup>, RAM RAJAMONY<sup>1</sup>, LIXIN ZHANG<sup>1</sup>, BABA ARIMILLI<sup>2</sup>

<sup>1</sup>IBM Research, Austin, TX, USA

<sup>2</sup>IBM Systems and Technology Group, Austin, TX, USA

## Definition

In 2002, IBM started to develop a pioneering supercomputer, codenamed PERCS (Productive, Easy-to-use, Reliable Computing System), with support from the

Defense Advanced Research Project Agency (DARPA). The project was part of DARPA's High Productivity Computing Systems (HPCS)[3] initiative, a 10-year research program that sought to change the landscape of high-end computing by shifting the focus away from just floating point performance toward overall system productivity. Enhancing system productivity required unprecedented innovation in the system design to simplify system usage and programming tasks, all while maintaining the system cost within the requirements of a realistic commercial offering. The system's productivity is orders of magnitude better than previous supercomputers as expressed by the High Productivity Challenge benchmark suite [7].

## Discussion

### Introduction

High-end computing went through a generational transformation during the 1990s, which saw emerging clusters of commodity processors and networks gradually replace traditional supercomputer systems based on vector and large shared-memory systems. Commodity clusters offered advantages in cost and scalability compared with the technologies they replaced. The simultaneous emergence of the Message Passing Interface (MPI) provided programmers with a portable, standard programming environment that exploited these clusters well. Interest in high-performance computing gained momentum, and a semiannual "top 500 supercomputers" contest was created based on the Linpack benchmark [4] to assess the available supercomputers in the market and to provide an arena for vendors to compete.

With the Linpack benchmark's emphasis on floating-point performance, designers focused on improving processor performance, and the resulting systems were showing great leaps in Linpack performance. These improvements however did not benefit mainstream applications that required a system design that balances floating-point operations with commensurate memory and communication bandwidths. For example, streaming data applications, message-intensive applications exemplified by the GUPS benchmark [7], and digital-signal processing applications based on Fast Fourier Transform [5] are sensitive to network performance and memory bandwidth. Programmers were forced to adapt

their algorithms to reduce data transfers and harness the increased computational capability of commodity clusters. The approach proved futile as programmers generally failed to achieve performance gains to show for the added code complexity and programming efforts. A programmer and system productivity crises were clearly at hand.

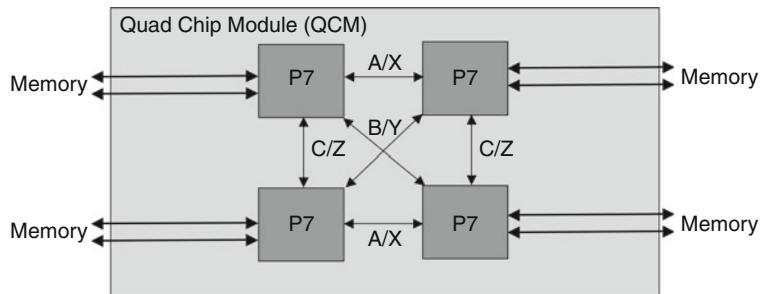
In 2001, visionaries at DARPA and various U.S. defense agencies took the initiative to confront the situation and started the High Productivity Computing System (HPCS) program. HPCS took the form of three competitive stages, the first two being devoted to pure research with each round culminating in the elimination of proposals that were deemed noncompetitive or noninnovative. The third stage was devoted to the commercialization of the research results of the first two stages into a mainstream product. Five companies competed in the initial stage, namely Cray, HP, IBM, SGI, and Sun, and by 2007, only Cray and IBM were supported to proceed with the third stage of commercialization. The HPCS vision called for balanced system attributes and high-level computing abstractions that alleviate programmers from the chores of adapting code to improve performance. The program aimed to foster research that would result in systems that are amenable to productive use without too much effort on the part of the programmer. This article provides a short summary of IBM's PERCS project. Compared to state-of-the-art high-performance computing (HPC) systems, PERCS achieves high performance and productivity goals through tight integration of computing, networking, storage, and software. PERCS focuses on scaling all parts of the system while easing the programming complexity.

## Key Elements of the PERCS Design

### Compute Node Design

The building block for the PERCS design is the compute node as shown in Fig. 1.

There are four POWER7 chips [6] in a node with a single operating system image that controls resource allocation. Each chip features eight cores, four threads per core, and two memory controllers that can connect to 128GB of DRAM memory. Applications executing on a single compute node can thus utilize 32 cores, 128 SMT threads, eight memory controllers, up



**PERCS System Architecture. Fig. 1** PERCS compute node architecture

to 512 GB of memory capacity, one teraflop of compute power, and over 512 GB/s of memory bandwidth. The four POWER7 chips are cache coherent and are tightly coupled using three pairs of buses. Each processor has six fabric bus interfaces to connect to three other processors in a multi-chip module (MCM). These 1-teraflop tightly coupled shared-memory nodes deliver 192 GB/sec of bandwidth (0.2 Byte/flop) to a specialized hub chip used for I/O, messaging, and switching. The POWER7 chip represents a large leap forward in single-chip performance, available both in terms of computational ability and achievable memory bandwidth over other processor chip offerings.

### PERCS Interconnect

The challenge of building a highly productive system required the entire system infrastructure to scale along with the microprocessor's capabilities. Previous solutions such as fat-tree Interconnect suffered from substantial communication latency due to message copying, protocol processing, multiple data conversions between optical and electrical signaling, and switching overheads. Other approaches such as toroidal networks mitigated some of these issues at the expense of handing the programmer very complex communication topologies.

Such limitations have forced programmers traditionally to consolidate data communications into large messages that amortize the communications overhead. Applications that required frequent exchange of short messages [7] either had to settle for poor performance or had to be modified in nontrivial ways to exploit large messages. To solve this problem, we had to design an

Interconnect that obeyed the following design principles in a cost-effective product:

- (a) Minimum data copying
- (b) Only one conversion from electrical to optical domain was allowed for any message
- (c) Simple communications protocol with limited processing overhead
- (d) Maintain a simple topology that simplifies programming

We approached the problem in an unorthodox way by building on the shared-memory protocol that IBM uses to build highly-scalable shared-memory machines. The communication is treated at three levels:

- (a) Within a node, communications take place over the shared-memory bus.
- (b) A second level consists of a *supernode*, which is a group of 32 nodes connected via noncoherent shared-memory buses. Physically, two supernodes fit within a single rack.
- (c) A third level consists of communications between supernodes. Supernodes are connected using a fully connected graph of optical cables. Thus, each supernode has an equal distance to other supernodes, simplifying the topology and the programming task.

A communication coprocessor [1] resides on the memory bus within each node, and thus data can be streamed from source to destination using the shared-memory protocol, using a 128-byte payload. This eliminates the data copying common in previous solutions where data had to be copied from the memory to the I/O bus.

The communication coprocessor, also called the hub-chip [1], is directly connected to the POWER7 MCM at 192 GB/s and provides 336 GB/s to seven other nodes in the same drawer on copper connections; 240 GB/s to 24 nodes in the same supernode (composed of four drawers, or 32 compute nodes) on optical connections; 320 GB/s to other supernodes on optical connections; and 40 GB/s for general I/O, for a total of 1,128 GB/s peak bandwidth per hub chip (see Fig. 2).

Two key design goals for PERCS were to dramatically improve bisection bandwidth (over other topologies such as fat-tree interconnects) and to eliminate the need for external switches. With these goals in mind, the hub chip was designed to support a large number of links that connect it to other hub chips. These links are classified into two categories “L,” and “D,” which permit the system to be organized into a two-level direct-connect topology. Every hub chip has thirty-one L links that connect to 31 other hub chips. Within this group of thirty-two hub chips, every chip has a direct communication link to every other chip. The hub chip implementation further divides the L links into two categories: seven electrical LL links with a combined bandwidth of 336 GB/s and 24 optical LR links with a combined bandwidth of 240 GB/s. The L links bind thirty-two compute nodes into a supernode.

Every hub chip also has 16 D links that are used to connect to other supernodes with a combined bandwidth of 320 GB/s. The topology maintains at least one D link between every pair of supernodes in the system, although smaller systems can employ multiple D links between supernode pairs.

The hub chip is connected to the POWER7 chips in the compute node at a bandwidth of 192 GB/s and has 40 GB/s of bandwidth for general I/O. The peak

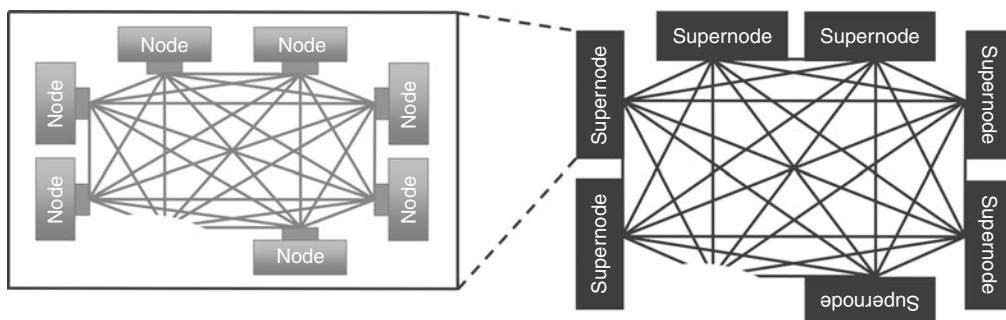
switching bandwidth of the hub chip exceeds 1.1 GB/s. When all links are populated and operate at peak, the injection bandwidth to network bandwidth ratio is 1:4.6. Note though that by performing the dual roles of data routing and interconnect gateway, the majority of traffic through the hub chip will typically be destined for other compute nodes. The low injection to network bandwidth ratio means that plenty of bandwidth is available for that other traffic.

The topology used by PERCS permits routes to be made up of very small numbers of hops. Within a supernode, any compute node can communicate with any other compute node using a distinct L link. Across supernodes, a compute node has to employ at most one L hop to get to the “right” compute node within its supernode that is connected to the destination supernode (recall that every supernode pair is connected by at least one D link). At the destination supernode, at most one L hop is again sufficient to reach the destination compute node.

## Routing Between Nodes

The above-described principles form the basis for direct routing in the PERCS system. A direct route employs a shortest path between any two compute nodes in the system. Since a pair of supernodes can be connected together by more than one D link, there can be multiple shortest paths between a given set of compute nodes. With only two levels in the topology, the longest direct route L-D-L can have at most three hops made up of no more than two L hops and at most one D hop.

PERCS also supports indirect routes to guard against potential interconnect hot spots. An indirect route is one that has an intermediate compute node in



PERCS System Architecture. Fig. 2 Diagram of PERCS interconnect

the route that resides on a different supernode from that of the source and destination compute nodes. An indirect route must employ a shortest path from the source compute node to the intermediate one, and a shortest path from the intermediate compute node to the destination compute node. The longest indirect route L-D-L-D-L can have at most five hops made up of no more than three L hops and at most two D hops. [Figure 3](#) illustrates direct and indirect routing within the PERCS system.

A specific route can be selected in three ways when multiple routes exist between a source-destination pair. First, software can specify the intermediate supernode but let the hardware determine how to route to and then from the intermediate supernode. Second, hardware can select amongst the multiple routes in a round robin manner for both direct and indirect routes. Finally, the hub chip also provides support for route randomization, whereby the hardware can pseudo-randomly pick one of the many possible routes between a source-destination pair.

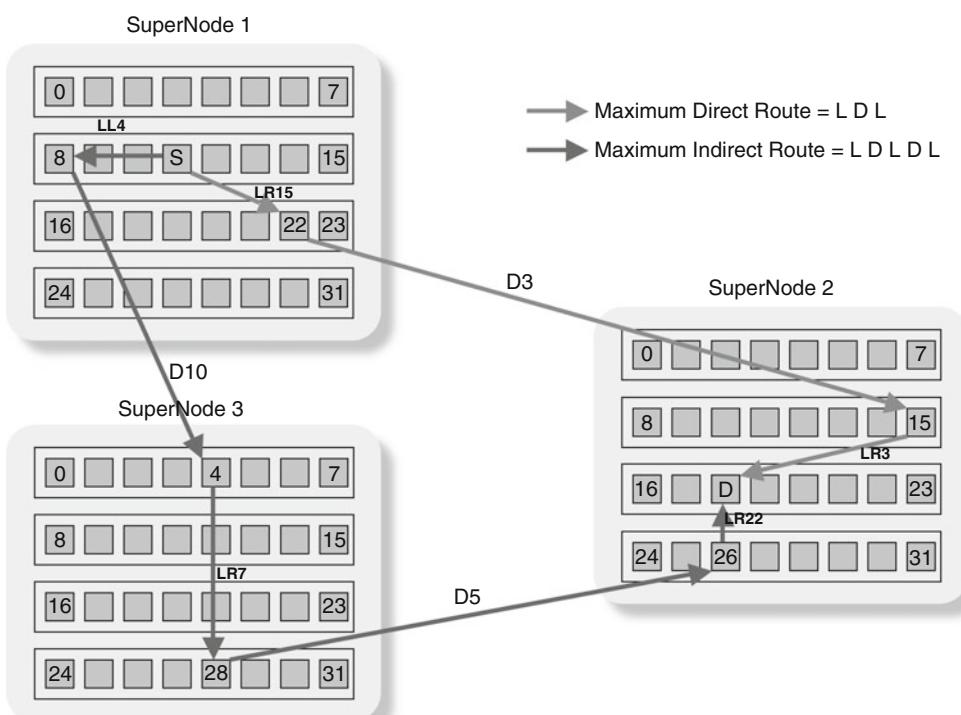
The minimum direct-routed message latency in the system is approximately  $1 \mu$  second. The peak unidirectional link bandwidths are as follows: 24 GB/s from each POWER7 chip in an SMP to its hub/switch, 24 GB/s from a given hub/switch to each of the other seven hub/switches in the same drawer (336 GB/s total), 5 GB/s from a given hub/switch to each of the other 24 hub/switches in different drawers in the same supernode (240 GB/s total), and 10 GB/s from a given hub/switch in one supernode to a hub/switch in another supernode to which that particular hub/switch is directly connected (320 GB/s total).

### Novel Features of the PERCS Hub Chip

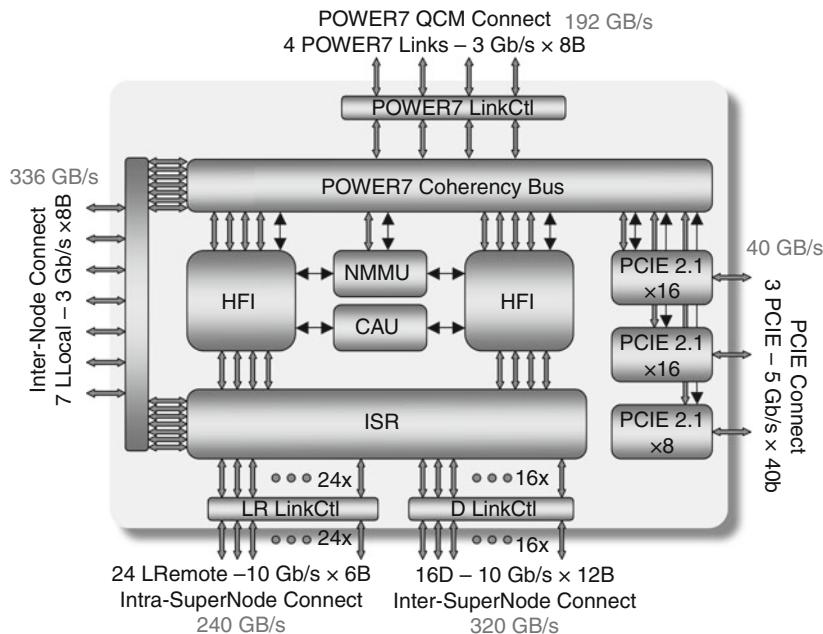
The PERCS Hub Chip, detailed in [Fig. 4](#), incorporates many features to enable the high-performance targets set for the system on HPC applications.

### Collective Accelerator Unit (CAU)

Many HPC applications perform collective operations that involve all or a large subset of the nodes



**PERCS System Architecture. Fig. 3** Direct and indirect routing in PERCS



**PERCS System Architecture. Fig. 4** Details of PERCS Hub chip

participating in the computation. Forward progress can only be realized when each node has completed the collective operation and results have been returned to all participating nodes. The PERCS hub chip provides specialized hardware to accelerate frequently used collective operations such as multicast, barriers, and reduction operations. For reductions, a dedicated arithmetic logic unit (ALU) within the CAU supports the following operations and data types:

- Fixed point: NOP, SUM, MIN, MAX, OR, AND, XOR (signed and unsigned)
- Floating point: MIN, MAX, SUM, PROD (single and double precision)

Software organizes the CAUs in the system into collective trees, firing when data on all of its inputs are available with the result being fed to the next “upstream” CAU in a data-flow manner. Each hub chip contains a single CAU. A multiple-entry content addressable memory (CAM) structure per CAU supports multiple independent trees that can be concurrently used by different applications, for different collective patterns within the same application, or some combination thereof.

### Power Bus Interface

The on-chip interconnect for the POWER7 processor is the newly-designed PowerBus architecture. Each hub chip contains a PowerBus interface that enables it to participate in the coherency operations taking place between the four POWER7 chips in the compute node, providing the hub chip visibility to coherence transactions taking place in the node.

### Host Fabric Interface

The two Host Fabric Interface (HFI) units in the hub chip manage communication to and from the PERCS interconnect. The HFI was designed to provide user-level access to applications. The basic construct provided by the HFI to applications for delineating different communication contexts is the “window.” The HFI supports many hundreds of such windows each with its associated hardware state.

An application invokes the operating system to reserve a window for its use. The reservation procedure maps certain structures of the HFI into the application’s address space with window control being possible from that point onward through user-level reads and write to the HFI-mapped structures.

The HFI supports three APIs for communication: a general packet transfer mechanism that can be used for composing either reliable or unreliable protocols upon which to build MPI or other messaging layers; a protocol for global address space operations that allow user-level codes to directly manipulate memory of a task residing on a different compute node; and direct internet protocol transfer ability.

The HFI can extract data that needs to be communicated over the interconnect from either the POWER7 memory or directly from the POWER7 caches. The choice of source is transparent, and data is automatically sourced from the faster location (caches can typically source data faster than memory). In addition to writing network data to memory, the HFI can also inject network data directly into a processor's L3 cache, lowering the data access latency for code executing on that processor [8].

### Integrated Switch Router (ISR)

The ISR implements the two-tiered full-graph network utilized in the PERCS system. It is organized as a  $56 \times 56$  full crossbar that operates at up to 3 GHz. In addition to the 47 L and D ports described previously, the ISR also has eight ports to the two local Host Fabric Interfaces, and one service port.

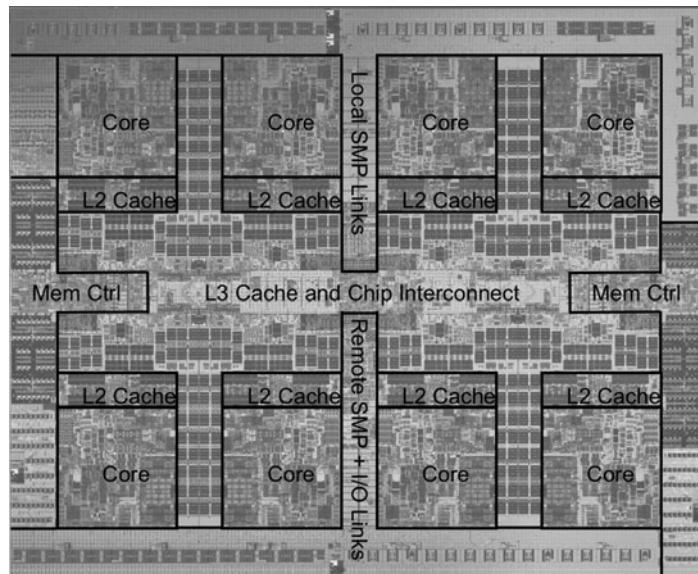
The ISR uses both input and output buffering with a packet replay mechanism to tolerate transient link

errors. This feature is especially important since the D links can be several tens of meters in length. The ISR operates in units of 128-byte FLITs with a maximum packet size of 2,048 bytes. Messages are composed of multiple packets with the packets making up a message being potentially delivered out of order.

High-performance computing applications benefit from having access to a single global clock across the entire system. The ISR implements a global clock feature, whereby a clock onboard is globally distributed across the interconnect and kept consistent with the clocks on other Hub chips. Deadlock prevention is achieved through virtual channels, each corresponding to a hop in the L-D-L-D-L worst case route.

### POWER7 Processor Overview

PERCS systems use a version of the new POWER7 processor chip fabricated in IBM's 45nm SOI CMOS technology, a die photo of which is shown in Fig. 5. POWER7 incorporates eight high-performance processor cores, L1, L2, and L3 caches per core, on-chip interconnect, multiple I/O controllers, memory controllers, and the SMP fabric controller. The processor design is closely linked with the technology node development, which is co-developed with the high-performance server processors to produce a high-performance server



PERCS System Architecture. Fig. 5 Die photo of the POWER7 processor chip

processor as well as a world class CMOS technology node.

The version of the POWER7 processor chip used in PERCS is fabricated utilizing IBM's 45nm technology at frequencies ranging up to 4 GHz. The chip size is 567 mm<sup>2</sup> and contains eight multithreaded cores. The processor design offers an excellent balance between floating and fixed-point operations, and between computation and memory access. Each core has four double precision floating-point units (*FPU*) implemented as two 2-way SIMD engines (scalar floating point instructions can only use two FPUs), two fixed-point units (*FXU*), two load-store units (*LSU*), a vector (*VMX*) unit, a decimal floating point unit, a branch unit, and a condition register. Each FPU is capable of performing one multiply-add instruction (*FMA*) in one cycle, and each LSU can execute simple integer operations. Thus the chip is capable of 256GF/s and 128GOp/s at 4GHz. The chip also features two memory controllers serving up to eight memory channels, with an aggregate bandwidth of 128GB/s. This offers about 0.5 B/F or 1B/Op greater memory bandwidth balance.

## Processor Core

POWER7 implements the 64-bit Power PC-AS architecture and is backward compatible with previous POWER chips, ensuring that existing binaries will run on the new architecture without the need to recompile. IBM's continued focus on single-thread performance in POWER7 remains an important component in its viability in the HPC marketplace. The core uses an extremely power-efficient, high-frequency design with a superscalar, out-of-order execution engine with highly-accurate branch prediction mechanism. Up to eight instructions may be executed in a given cycle.

POWER7 also provides excellent performance for throughput-oriented workloads. Simultaneous multithreading consisting of four hardware contexts per core provides up to 32 different instruction streams per chip for use by software. The multithreaded design ensures that a thread does not block the flow of instructions for other threads through the instruction pipeline, the cache or memory hierarchy, while keeping the area and power overhead low.

## Processor Cache Hierarchy

The POWER7 processor chip has three levels of cache, all sharing a common line size of 128 bytes. ECC is used extensively to ensure reliability and data integrity across all cache levels. While the Level 2 (L2) and Level 3 (L3) caches contain both instruction and data, the Level 1 (L1) cache is split into separate instruction and data caches. The L1 instruction cache is 32KB, 4-way set-associative and provides a maximum bandwidth of 64B per processor cycle. The L1 data cache is 32KB, 8-way set-associative and can be accessed in every processor cycle to deliver two 16B load operations, and one 16B store operation.

Each core has a private, 8-way set-associative, 256KB L2 cache. The L2 cache access latency is about seven processor cycles, and in every processor cycle, the processor core can load 32B of data or instructions from L2 and store 16B of data.

Each core has a private 16-way set-associative L3 cache of size 4MB constructed from on-chip embedded DRAM (eDRAM), which offers low power, high density, and excellent access times. The L3 cache access latency is about 22 processor cycles, and in every processor cycle, it can supply 16B of data to the L2 cache and receive 8B of data from the L2 cache. The L3 cache serves as a victim cache of the L2 cache. All L3 caches on a chip may be virtually aggregated into a quasi-shared cache by causing L3s to write back dirty lines to each other before writing them off-chip to memory. A sophisticated multi-tier LRU algorithm maintains balance among this confederation of L3 caches. The L3 cache arrays can also be reconfigured so that two adjacent banks are combined into a larger L3 cache of size 8MB shared between the two corresponding cores, which is useful for commercial workloads. This is another instance of the configurability of the architecture, and how this configurability is used to meet different workload characteristics and enhance the commercial viability of the system as outlined in the vision set forth by DARPA for the HPCS program.

## On-chip Integrated Fabric and Chip Interconnect

Another innovative aspect of the POWER7 design is the use of existing system buses to both implement shared-memory traffic and integrate the network switching functionality. Effectively, the POWER7 implements a

distributed switch function within the computer rack. The POWER7 Fabric Bus Controller (FBC) is the building block that implements this distributed switch functionality.

The FBC is integrated on the POWER7 chip and acts as the central point of cache-coherent data traffic. It is responsible for implementing cache-to-cache data transfers, bus arbitration, address routing, etc. It maintains simple and configurable routing tables and implements all necessary flow control to ensure freedom from deadlocks and livelocks. The FBC acts as the hub of all coherent and noncoherent communications among all internal chip units (eight processor cores, two memory controllers, L2 and L3 caches, Non-Cacheable Unit, Gigabit Ethernet, and PCI-Express controller) to other internal units on or outside of the chip. The FBC provides all of the interfaces, buffering, and sequencing of address and data operations within the coherent memory subsystem.

Physically, the Fabric bus is an 8-Byte wide, split-transaction, multiplexed address and data bus. Data packets are four beats each carrying 32 bytes of data. It takes four data packets to transfer the entire 128 Byte cache line. Configuration switches assign node identification for each chip and also for data routing tables. Data transactions within the node are always sent along a unique point-to-point path. A tag travels with the data to help make routing decisions along the way. The Fabric busses provide fully connected topology to reduce latency.

## Memory Subsystem

Special care has been taken in the design of the POWER7 chipmemory subsystem to ensure the maximum bandwidth within reasonable constraints of cost and power. The design is flexible and configurable, offering different memory access modes, each optimized to a certain application profile, ranging from sequential access of memory to random access of memory. The memory subsystem also can perform partial cache line loads, improving the efficiency and bandwidth for applications with poor spatial locality such as sparse matrix computations.

The PERCS memory subsystem has two memory controllers with support for up to eight DIMM sockets and uses custom DDR3 DRAM chips running at 1.333 GHz. These DIMMs include specialized buffer chips

to increase the amount of memory and bandwidth to memory per chip.

Peak memory bandwidth (read + write) of a single channel is rated at 16GB/s, two thirds of which is for read bandwidth and the remaining one third for write bandwidth. The address space is split across the multiple fully buffered DIMM arrays in a manner that allows for an evenly distributed access pattern across them. This access pattern results in the highest bandwidth available at the processor interface and an evenly distributed thermal pattern.

## Blue Waters: The First PERCS Installation

The PERCS system design will first be implemented in the Blue Waters supercomputer to be placed at the National Center for Supercomputing Applications (NCSA) at the University of Illinois [2]. Blue Waters will be comprised of 304 supernodes with nearly 312,000 POWER7 cores, more than 1 PB memory, more than 10 PB disk storage, more than 0.5 EB archival storage, and will achieve close to 10 PF/s peak performance. Acquisition of the Blue Waters system is supported by the National Science Foundation and leverages the investment made by the Defense Advanced Research Projects Agency's High Productivity Computing Systems (HPCS) program.

IBM, the University of Illinois, and the NCSA will work together throughout Blue Waters' lifespan to enhance IBM's high-performance computing environment, ensuring that applications can take full advantage of Blue Waters and achieve performance on a variety of real-world applications. The enhanced high-performance computing environment will also increase the productivity of applications developers, system administrators, and researchers by providing an integrated toolkit for using, analyzing, monitoring, and controlling Blue Waters.

Blue Waters is expected to be one of the most powerful supercomputers in the world when it comes online. It will have a peak performance close to 10 petaflops (10 quadrillion calculations every second) and will achieve sustained performance of 1 petaflop per second running a range of science and engineering codes. Scientists will create breakthroughs in nearly all fields of science using Blue Waters, including predicting the behavior of complex biological systems; understanding how the cosmos

evolved after the Big Bang; designing new materials at the atomic level; predicting the behavior of hurricanes and tornadoes; and simulating complex engineered systems like the power distribution system, airplanes, and automobiles.

## Bibliography

1. Arimilli B, Arimilli R, Chung V, Clark S, Denzel W, Drerup B, Hoefer T, Joyner J, Lewis J, Li J, Ni N, Rajamony R (2010) The PERCS high-performance interconnect. Proceedings of Hot Interconnects, Mountain View, CA, August 2010
2. Blue waters project at the national center for supercomputing applications. <http://www.ncsa.illinois.edu/BlueWaters/>. Accessed January 2010
3. DARPA high productivity computer systems. <http://www.highproductivity.org/>
4. Dongarra J, Luszczek P, Petitet A (2003) The LINPACK benchmark: past, present, and future. *J Concur Comput Pract Exp* 15:803–820
5. Duhamel P, Vetterli M (1990) Fast Fourier transforms: a tutorial review and a state of the art. *Signal Process* 19:259–299
6. Kalla R, Sinharoy B (2009) POWER7: IBM's next generation server processor. *IEEE symposium on high-performance chips (Hot chips 21)*, Stanford, August 2009
7. Luszczek P, Dongarra J, Koester D, Rabenseifner R, Lucas B, Kepner J, McCalpin J, Bailey D, Takahashi D (2005) Introduction to the HPC challenge Benchmark Suite. <http://icl.cs.utk.edu/projectsfiles/hpcc/pubs/hpcc-challenge-benchmark05.pdf>. March 2005
8. Milenkovic A, Milutinovic V (2000) Cache injection: a novel technique for tolerating memory latency in bus-based SMPs. *EUROPAR 2000 parallel processing. Lecture notes in computer science, vol 1900/2000*, Munich, pp 558–566

ARC2D, a finite difference fluid dynamics code developed at NASA Ames; BDNA, a molecular dynamics code for nucleic acid simulation contributed by IBM Kingston; DYFESM, a structural dynamics finite element code contributed by NASA Langley Research Center; FLO52Q, a computation fluid dynamics code contributed by Princeton University; MDG, which uses molecular dynamics to simulate liquid water; MG3D, a signal processing code contributed by Tel Aviv University; OCEAN, a computational fluid dynamics code contributed by Princeton University; QCD, a quantum chromodynamics code contributed by Caltech; SPEC77, a weather simulation code contributed by CSRD; SPICE, a circuit simulator contributed by UC Berkeley; TRACK, which determines the course of a collection of targets from observations taken at regular intervals, contributed by Caltech; and TRFD, a quantum mechanics computation code contributed by IBM Kingston.

## Bibliography

1. Berry M, Chen D, Koss P, Kuck D, Lo S, Pang Y, Pointer L, Roloff R, Sameh A, Clementi E, Chin S, Schneider D, Fox G, Messina P, Walker D, Hsiung C, Schwarzmeier J, Lue K, Orszag S, Seidl F, Johnson O, Goodrum R, Martin J (1989) The perfect club Benchmarks: effective performance evaluation of supercomputers. *International Journal of Supercomputing Applications* 4(3):9–40

## Perfect Benchmarks

The Perfect Benchmarks [1] were created in the late 1980s under the leadership of the University of Illinois' Center for Supercomputing Research and Development (CSR) with the participation of several other institutions (the Perfect Club). The name Perfect is an acronym for PERformance Evaluation by Cost-effective Transformations. The Benchmarks came from real applications in contrast to the Livermore Loops and other simple codes used in the 1980s to evaluate machines. Thirteen Benchmarks were included: ADM, solves the hydrodynamics equations to simulate air pollution, contributed by IBM Kingston;

## Performance Analysis Tools

MICHAEL GERNDT

Technische Universität München, München, Germany

## Synonyms

[Performance measurement](#); [Profiling](#); [Tracing](#)

## Definition

Performance analysis tools support the application developer in tuning the application's performance for a given architecture. They measure performance data during the execution of the application and provide means to analyze and interpret the provided data and to detect performance bottlenecks.

## Discussion

### Introduction

The development of high-performance applications requires a careful adaptation of the program to the underlying parallel architecture. Due to the manifold interrelations of the parallel program and the architecture, designing an application with optimal performance on parallel systems is almost impossible. Therefore, the application goes through a tuning cycle which consists of measuring performance, detecting performance bottlenecks, and applying program transformations. The assumption for this tuning approach is that the performance will be the same for different runs with the same resources and the same input data.

Performance analysis tools support the programmer in the first two tasks of the tuning cycle. Performance data are gathered during program execution by monitoring the application's execution. Performance data are either summarized and stored as profile data or all the details are stored in so-called trace files.

In addition to application monitoring, performance analysis tools also provide the means to analyze and interpret the provided performance data and thus to detect performance problems. The following sections present the basis of performance analysis, i.e., the execution event model, the different monitoring techniques, and the most important analysis techniques.

### Event Model

The abstraction of the execution used in performance analysis is an event model. Events happen at a specific point in time in a process or thread. Events belong to event classes, such as enter and exit events of a user-level function, start and finish events of a send operation in MPI programs, iteration assignment in work-sharing constructs in OpenMP, and cache misses in sequential execution.

In profiling, information about events of the same class is aggregated during runtime. For example, cache misses are counted, the time spent between the start event and finish event of a send operation is accumulated as the communication time of the call site, and the time for taking a scheduling decision in work-sharing loops is accumulated as parallelization overhead.

In tracing, specific information is recorded for each event in a trace file. The event record written to the file

at least contains a time stamp and an identification of the executing process or thread. Frequently, additional information is recorded, such as the message receiver and the message length or the scheduling time for an iteration assignment.

### Monitoring

The general technique for collecting information about events is called *Monitoring*. Three different monitoring techniques can be distinguished: hardware monitoring, sampling, and instrumentation. These techniques are explained in the next three paragraphs.

### Hardware Monitoring

The monitoring of applications should not influence the program's execution. Therefore, hardware monitoring is required for frequent events, such as events in the processor or the caches. If cache misses, e.g., were to be counted via hardware interrupts, the intrusion would be immense. Therefore, current microprocessors provide event counters. A small number of hardware counters can be used to count a large number of different event types. Careful selection of the right event type is thus required.

The low-level APIs for accessing the system's hardware counters are quite different on different architectures. Therefore, standard APIs have been developed. The most notable one is the *Performance Application Programming Interface* (PAPI) [4, 8].

### Sampling

Many profiling tools such as the Unix *prof* utility collect statistical performance information via sampling. The processor is interrupted with a certain frequency, known as the sampling rate. The interrupt routine determines the address of the current instruction from the program counter and adds, e.g., the length of the sampling interval to the accumulated execution time of the currently executed source location. Instead of the execution time, other metrics such as the number of cache misses or of floating point operations can be used.

The most important advantage of this technique is that its intrusion is usually low and can be controlled by setting the sampling rate appropriately. The major drawback is that only statistical information is gathered. More precise information can be obtained via instrumentation.

A similar technique is called *Event-Based Sampling*. Each time a hardware counter passes a threshold, an interrupt is generated and information is accumulated for the current source line. The user can control the intrusion of this method by specifying the overflow threshold. A larger threshold will give less accurate information, but is less intrusive. This technique is used, e.g., in the Digital Continuous Profiling Infrastructure (DCPI) [1].

## Instrumentation

In contrast to sampling, precise information about the dynamic application behavior can be obtained via program instrumentation. Here the application is modified to gather information at specific events. For example, a call to a monitoring routine is inserted at the beginning and end of a user function to measure the execution time of the function.

Three important instrumentation techniques can be distinguished: source code instrumentation, object code instrumentation, and library interposition. In *source code instrumentation*, the compiler or a source-to-source transformation tool inserts the monitor calls into the program before compilation. In object code instrumentation, the instrumenter patches the machine code.

While the first approach is very portable, *object code instrumentation* is extremely machine specific. It depends not only on the machine's instruction set but also on the compiler, the OS, and the object format. On the other hand, with object code instrumentation, the source language is unimportant and programs can even be instrumented for which no source code is available.

Object code instrumentation can also be applied at runtime. While the program is already executing, instrumentation is inserted only at the required places. This technique was developed within the Paradyn environment and is supported in DYNINST [3, 5] and DPCL [2, 7].

While object code instrumentation is limited to code regions that can be deduced from the binary, it is typically limited to functions or basic blocks. Source code instrumentation can be used to gather information for arbitrary program regions.

Instrumentation of functions can also be done based on a technique called *library interposition*. This

technique is used to instrument MPI functions via the MPI profiling interface (PMPI). It determines for all MPI functions a second name via the PMPI prefix. The implementor of an MPI monitoring library can write wrappers for MPI functions and call the PMPI version inside. Within the wrapper, code can be inserted to collect performance information. The wrapper library is then linked to the application before the original MPI library so that the wrappers are called instead of the original functions.

## Analysis

The techniques and tools for performance analysis presented in this entry all assume that the performance behavior of the application is the same over multiple experiments. Based on this assumption, multiple experiments can be performed with different tools or different tool configurations to identify and rank performance problems.

The following aspects are relevant for performance analysis tools:

1. Level of detail
2. Performance aspects
3. Application perturbation
4. Level of automation
5. Scalability

## Level of Detail

Three classes of performance analysis tools can be distinguished depending on how detailed the raw performance data are gathered and analyzed, i.e., profiles, profile time series, and traces.

*Profiling tools* aggregate the raw performance data over possibly multiple dimensions. Most common is the aggregation over time, e.g., the number of cache misses for individual functions in each process. Some tools also aggregate the data over processes and threads or even into a single value for the entire execution, e.g., the number of floating point instructions executed by an application. Furthermore, tools apply aggregation with respect to the program regions, i.e., functions and loops. They either compute a flat profile, in which the data are aggregated for each region, or a call path profile, where the data are aggregated for each call path. Call path profiles are very useful if the behavior of functions is different for individual invocations. This is frequently

the case for library routines such as basic mathematical or MPI routines.

Some profiling tools provide not only aggregated data over the entire execution but also time series of snapshots of profile information. Such time series of profile data allow the analysis of time-dependent variations of the performance behavior.

The most detailed analysis is performed by *tracing tools* that store information about individual events in so-called traces. For each event a trace record with a time stamp is generated. A trace record includes additional data, such as the sender and receiver, the amount of data, and an identification of the message sent by an MPI send operation. The trace records are typically stored in trace files that are subsequently analyzed for performance problems.

While profiling tools have the advantage that the amount of performance data is not proportional to the execution time of the application, tracing tools generate performance data proportional to the execution time and the number of processes and threads. On the other hand, profiling tools are limited in the analyses they can perform.

## Performance Aspects

Most performance analysis tools are specialized for specific performance aspects. Tuning an application with respect to all possible performance problems probably requires the use of multiple performance analysis tools. The performance tools are typically specialized in one or multiple of the following aspects:

1. Execution time
2. Instruction execution
3. Memory access behavior
4. Memory usage
5. IO
6. Parallel execution

The most basic and thus the most frequently used performance tools provide the user with a time profile. This allows the user to detect the hot code regions, i.e., those regions where most of the execution time is spent and thus have the highest potential for performance improvement.

The next class of tools supports the programmer in the inspection of performance problems related to the utilization of the resources in an individual core. These

tools typically provide information about the instruction mix, highlight expensive operations such as divides or type conversions, point out exception handling, or identify pipeline stalls, e.g., due to branch mispredictions. They are based on measurements performed with the processor's hardware performance counters.

Very critical for the performance of applications is the memory access behavior. The long latency of memory accesses and the limited bandwidth require that most of the data accesses are served from the on-chip caches. Programs must be carefully tuned for data locality so that the caches are most effectively used. Performance tools support the analysis of the access behavior with information about the number of cache hits and misses as well as the number of stall cycles for memory references. More precise information can be obtained from cache simulators which are fed with memory reference traces of the application. Cache simulation tools, e.g., can determine the distribution of misses across miss classes, i.e., compulsory, conflict, capacity, and invalidation misses. They can identify false sharing as well as compute higher-level metrics, such as the reuse distance.

Another memory-related analysis supported by performance analysis tools is the inspection of memory usage. Performance problems might arise from allocating too much memory may be due to the size of data structures or to memory leakages.

High-performance applications tend to work on huge data sets. Thus performance tools need to help in the identification of performance problems with respect to file IO. Information can be obtained from runtime libraries implementing IO operations as well as from the OS.

Many performance analysis tools support the detection of performance problems with respect to the parallel execution. Certainly, the two standard programming interfaces MPI and OpenMP have the best support.

Profiling and tracing tools are available for message passing programs that support the analysis of communication among the processes. More elementary tools provide measurements on the time spent in certain MPI functions and the amount of data transferred between communication partners, while advanced tools provide information on different waiting times that indicate certain performance problems, e.g., the late sender problem where the sender of a message arrives later at the

send than the receiver at the receive statement. Tools for message passing also indicate load balancing problems based on the execution time of collective operations.

Analysis tools for shared memory programming, i.e., OpenMP, focus on load imbalances, synchronization, and management overhead. Currently available tools do not yet fully support the extensions of OpenMP 3.0, most notably the tasking concept.

## Perturbation

Performance analysis tools are based on measurements. If those measurements are not fully done by additional hardware, the tool will influence or change the codes performance behavior. The intrusion of tools has multiple sources. First of all the time spent in the monitoring library delays the execution. Another important overhead is flushing performance data to external files which needs to be done by tracing tools since the trace records cannot be stored in main memory. These flushes can either be done at global synchronization points and thus delay the execution of all processes, or they can be done asynchronously in the processes which might lead to artificial load imbalance. The intrusion can also be more subtle because memory accesses of the performance analysis tool might change the cache status, or execution delays might even influence the algorithmic behavior in the case of nondeterministic algorithms.

Performance analysis tools try to work around that problem by either correcting the obtained measurements or by reducing the amount of perturbation. Since the first approach is extremely difficult many current tools focus on the second approach.

While the overhead induced by the measurements can be controlled in the sampling approach by selecting an appropriate sampling rate, instrumentation-based measurements can be optimized by reducing the amount of instrumentation and by tuning the instrumentation functions. Instrumentation of program regions that are frequently called but have only little execution time suffer most from the instrumentation overhead. Performance analysis tools provide the means to guide the instrumentation, such as to instrument only certain region types or to exclude individual program regions.

Of course, perturbation not only results from the instrumentation but also from other time-consuming

activities of the analysis tools during the program's execution. Examples are the management of trace records and the computation of call path profiles. Various techniques have been developed to tune those operations.

## Automation

The ultimate goal of performance analysis tools is to indicate performance problems and thus potential for performance improvement. Ideally, the tools should perform that task in a fully automatic way. Full automation requires the formalization of the performance analysis specialist's and the application specialist's knowledge. Until now, this has only partially been achieved.

One area of automation is program instrumentation. The techniques here range from manual instrumentation to fully automatic instrumentation. Some tools require the user to insert instrumentation into the program. Other tools provide semiautomatic instrumentation via the compiler or a source-to-source instrumenter. The instrumentation can be configured to control the amount of program perturbation. Some tools assist the user in the configuration by pointing out regions that need not be or should not be instrumented based on a previous analysis run. In the fully automatic approach, the instrumentation is selected by the performance analysis tool. Based on the tool's knowledge of which instrumentation is required to obtain the required performance data, the instrumentation is automatically configured.

Automation is also required in the actual measurement of performance data via the monitoring. The analysis tool decides automatically which data will be measured. However for some tools, especially if they access the hardware performance counters, the measurement configuration is done manually. Depending on the type of performance problem the user is interested in, he or she can configure the analysis to measure certain events with the hardware counters. It might even be necessary to perform multiple runs of the application to gather all the required information.

In addition, the last step of the analysis process can be automated. A performance analysis tool should automatically extract and rank the performance problems such that the user can start by tuning the application to solve the most severe performance problem.

In most tools, this step is not automated. The tools allow the user to manually inspect the performance data either via textual or graphical displays. Textual information, i.e., a sorted list of hot regions, is provided with references to the source code or the source code is actually annotated with information such as the number of cache misses. Graphical diagrams are typically used in tracing tools. The dynamic behavior is visualized via timeline diagrams, statistics are shown as bar charts, and many other chart types are utilized to visually represent the huge data sets. It is the task of the user of those tools to select the right displays to identify performance problems.

Only a few tools automate the last step and automatically identify performance problems and their severity. The tools use a formal description of potential performance problems and verify whether those known potential problems can be found based on the measured data.

Fully automatic tools automate the whole process, covering instrumentation, measurement, and analysis. The search for performance problems is driven based on formalized potential performance problems. Following a certain search strategy, they deduce from the performance problem specifications which information is required, measure the data, verify whether a problem exists, and report the found problems to the tool user.

Fully automatic tools can perform the analysis online. Online tools have also been developed for semiautomatic tools that visualize performance data, but those require that the user performs the analysis while the application is executing. This is not appropriate especially since large-scale application runs are only possible in batch jobs. Therefore, most of those tools are offline tools. They produce the data by monitoring an actual run of the application but the inspection and analysis of the data is done independently of the execution.

## Scalability

Especially in the field of high-performance computing, performance analysis tools have to be scalable. Applications are run on thousands of processors and the performance behavior cannot be studied for scaled down runs. Tools have to be scalable with respect to the number of processors as well as to the duration of the application's execution. Typical limitations of the tools are the

time spent in the analysis as well as the space requirements to store and process performance data. Scalability is obviously a big issue for tracing tools, but profiling-based tools can also easily run into scalability problems if hundreds of thousands of processors are used as in the current BlueGene systems. This problem can be alleviated by processing the data in a tree analysis network (MRNet [15]). The leaves measure the performance data and while the data travel up the tree, they are aggregated to coarsen the information.

Several techniques for tracing tools have been developed to increase scalability. Most important is to reduce the size of the traces. Therefore, compression techniques as well as "on the fly" detection of recurring patterns in traces are applied. A few tools also parallelize the analysis step. SCALASCA post-processes the trace in parallel on the processors of the application after this has terminated. Vampir uses a parallel analysis server that processes the trace files while the user is working with the analysis tool to inspect the measured performance data. Periscope processes the performance data within a hierarchy of analysis agents while the application is executing.

## Representative Tools

This section identifies a few performance analysis tools and gives a short characterization. The mentioned tools are only a very small set of examples from the numerous tools developed.

*Gprof* is the GNU Profiler tool. It provides a flat profile and a callpath profile for the program's functions.

The measurements are done by instrumentation.

*Vtune* is a performance analysis tool from Intel that provides flat and callpath profiles based on sampling and instrumentation, as well as measurements based on the hardware performance counters. The measurements can be inspected on source and assembler level.

*pfmon* is a tool for accessing the hardware performance counters developed by HP. It can be used to collect certain events specified via command line arguments. Application-specific as well as system-wide measurements can be performed.

*ompP* is a profiling tool for OpenMP developed at Technische Universität München and University of Tennessee [9]. It is based on instrumentation with Opari

[13] and determines certain overhead categories of parallel regions.

*HPC Toolkit* from Rice University uses statistical sampling to collect a performance profile [17]. It relates the measurements back to the original source code based on a static analysis of the binary and presents the data by annotating the sources.

*Tau* is a flexible performance environment from the University of Oregon [16]. It can generate application profiles as well as traces. While it provides tools to analyze the profiles, traces can be analyzed with Vampir.

*Vampir* is a commercial trace-based performance analysis tool from the Technische Universität Dresden [14]. It provides a powerful visualization of traces and scales to thousands of processors based on a parallel visualization server.

*Paraver* is a trace visualization and analysis environment from the Barcelona Supercomputing Center. It provides powerful analysis services such as automatic phase detection [6] and clustering.

*Paradyn* from the University of Wisconsin was the first automatic online analysis tool [12]. Its performance consultant guided the search for performance bottlenecks while the application was executing.

*SCALASCA* is an automatic performance analysis tool developed at the Forschungszentrum Jülich [10]. It is based on performance profiles as well as on traces. The automatic trace analysis determines MPI wait time via a parallel trace replay after the application execution on the application's processors.

*Periscope* currently under development at the Technische Universität München follows the approach of Paradyn and performs an online search for performance problems [11]. It is based on a hierarchy of analysis agents to fulfill scalability requirements of current and future HPC systems.

## Related Entries

- [Intel® Thread Profiler](#)
- [OpenMP Profiling with OmpP](#)
- [Parallel Tools Platform](#)
- [Periscope](#)
- [PMPI Tools](#)
- [Scalasca](#)
- [Tau](#)
- [Vampir](#)

## Bibliography

1. Digital continuous profiling infrastructure. [www.unix.digital.com/dcpi](http://www.unix.digital.com/dcpi)
2. Dynamic probe class library. [oss.software.ibm.com/developerworksopensource/dpcl/](http://oss.software.ibm.com/developerworksopensource/dpcl/)
3. Dyninst api. [www.dyninst.org](http://www.dyninst.org)
4. Performance application programming interface. [icl.cs.utk.edu/papi](http://icl.cs.utk.edu/papi)
5. Buck B, Hollingsworth JK (2000) An API for runtime code patching. *Int J High Perform Comput Appl* 14(4):317–329
6. Casas M, Badia RM, Labarta J (2007) Automatic phase detection of MPI applications. In: Bischof C et al (eds) *Proceedings of the international conference on parallel computing (ParCo '07)*, Jülich/Aachen. *Advances in Parallel Computing*, vol 15, IOS, Amsterdam, pp 129–136
7. DeRose L, Hoover T, Hollingsworth JK (2001) The dynamic probe class library – an infrastructure for developing instrumentation for performance tools. IBM, 2001
8. Dongarra J, London K, Moore S, Mucci P, Terpstra D, You H, Zhou M (2003) Experiences and lessons learned with a portable interface to hardware performance counters. In: IPDPS '03: *Proceedings of the 17th international symposium on parallel and distributed processing*, Washington, DC, IEEE Computer Society, p 289.2
9. Fürlinger K, Gerndt M, Dongarra J (2007) Scalability analysis of the SPEC OpenMP benchmarks on large-scale shared memory multiprocessors. In: Shi Y, van Albada GD, Dongarra J, Sloot PMA (eds) *Computational science – ICCS 2007*, Beijing. Lecture notes in computer science, vol 4488. Springer, Berlin, pp 815–822
10. Geimer M, Wolf F, Wylie BJR, Mohr B (2006) Scalable parallel trace-based performance analysis. In: *Proceedings of the 13th European PVM/MPI users' group meeting on recent advances in parallel virtual machine and message passing interface (EuroPVM/MPI 2006)*, Bonn, pp 303–312
11. Gerndt M, Ott M (2010) Automatic performance analysis with Periscope. *Concurr Comput Pract Exp* 22(6):736–748
12. Miller BP, Callaghan MD, Cargille JM, Hollingsworth JK, Irvin RB, Karavanic KL, Kunchithapadam K, Newhall T (1995) The Paradyn parallel performance measurement tool. *IEEE Comput* 28(11):37–46
13. Mohr B, Malony AD, Shende SS, Wolf F (2001) Towards a performance tool interface for OpenMP: An approach based on directive rewriting. In: *Proceedings of the third workshop on OpenMP (EWOMP'01)*, Barcelona, September 2001
14. Müller MS, Knüpfer A, Jurenz M, Lieber M, Brunst H, Mix H, Nagel WE (2007) Developing scalable applications with Vampir, VampirServer and VampirTrace. In: Bischof C et al (eds) *Proceedings of the international conference on parallel computing (ParCo '07)*, Jülich/Aachen. *Advances in Parallel Computing*, vol 15, IOS, Amsterdam, pp 113–120
15. Roth PC, Arnold DC, Miller BP (2003) MRNet: A software-based multicast/reduction network for scalable tools. In: *Proceedings of the 2003 conference on supercomputing (SC 2003)*, Phoenix, November 2003

16. Shende SS, Malony AD (2006) The TAU parallel performance system. *Int J High Perform Comput Appl*, ACTS Collection Special Issue, SAGE, 20(2):287–311
17. Tallent NR, Mellor-Crummey JM, Adhianto L, Fagan MW, Krentel M (2009) Diagnosing performance bottlenecks in emerging petascale applications. In: SC '09: Proceedings of the conference on high performance computing networking, storage and analysis, Portland, ACM, New York, pp 1–11

## Performance Measurement

- Performance Analysis Tools

## Performance Metrics

- Metrics

## Periscope

MICHAEL GERNDT

Technische Universität München, München, Germany

### Definition

Periscope is an automatic performance analysis tool for highly parallel applications written in MPI or OpenMP. It performs an online search for performance properties in a distributed fashion. The properties found by Periscope point to areas of parallel programs that might benefit from further tuning.

### Discussion

#### Introduction

Performance analysis is an important step in the development of parallel applications for high-performance architectures. Due to the complexity of the architecture of these machines, applications can typically not be designed to be most efficient. Experiments are required to understand the performance obtained and to identify possible areas in the code that might profit from further tuning.

Performance analysis tools help the developer in analyzing the application's performance. During an execution of the application with a typical input data set

and a typical number of processors, performance data are measured. Afterwards those data are analyzed to detect performance problems of the application. Certain tuning actions can then be selected, e.g., selection of different compiler optimization switches or modifications of the source code, based on the detected performance problems.

Periscope is a performance analysis tool developed at Technische Universität München. It is a representative for a class of automatic performance analysis tools automating the whole analysis procedure. Specific for Periscope is that it is an online tool and it works in a distributed fashion. This means that the analysis is done while the application is executing (online) and by a set of analysis agents that are searching for performance problems in a subset of the application's processes (distributed).

Periscope was inspired by the work of the European American APART working group and especially by Paradyn developed at University of Wisconsin, Madison. Paradyn uses a performance consultant that does dynamic instrumentation and searches for bottlenecks based on summary information during the program's execution.

Automation in Periscope is based on formalized performance properties, e.g., inefficient cache usage or load imbalance. Those properties are formalized in the APART Specification Language (ASL). Based on a repository of performance properties the analysis agents can search for those properties in the program execution under investigation. They automatically determine which properties to search for, which measurements are required, which properties were found, and which are more specific properties to look for in the next step. The low-level performance data are analyzed in a distributed fashion and only high-level performance properties are finally reported to the user, significantly reducing the amount of data to be inspected by the user.

#### Performance Properties

The basis for the analysis is the formalization of performance properties. Each *performance property* consists of a condition, a severity, and a confidence. The *condition* uses measured performance data and static information to decide whether the property holds and thus is a *performance problem*. The *confidence* defines whether

the property is only a hint for a performance problem or a proof. For example, properties based on static information, e.g., the number of prefetch operations, are typically only hints since the actual execution of those operations depends on runtime values. The *severity* states the importance of the performance problem in comparison to the other found performance problems. Typically, the severity is calculated as the amount of time lost due to the problem.

## Search Strategies

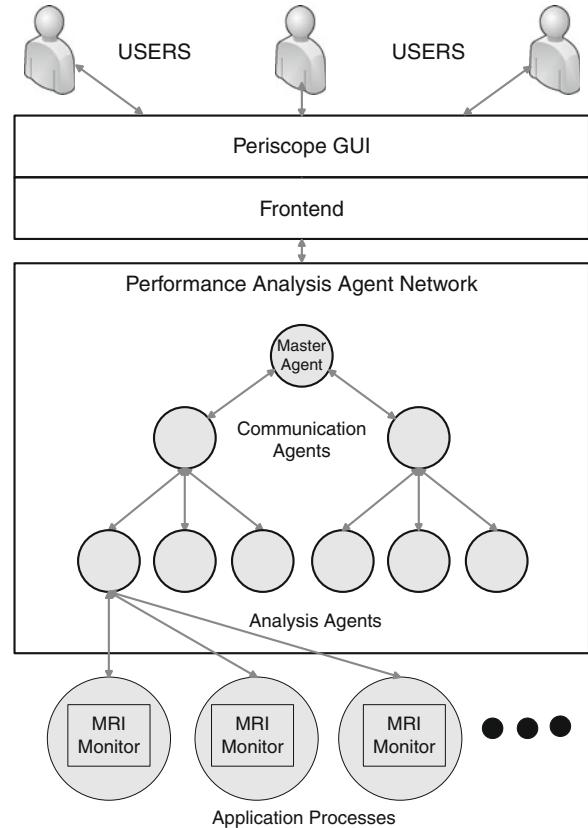
The overall search for performance problems is determined by search strategies. A *search strategy* defines in which order an analysis agent investigates the multidimensional search space of properties, program regions, and processes. Many of Periscope's search strategies are multistep strategies, i.e., they consist of multiple search steps. A very important concept for multistep strategies is a *program phase* which is determined by a certain repetitive program region. A good example for such a phase region is the body of the time loop in scientific simulations where each iteration simulates one time step.

In a first search step, the strategy determines a set of candidate properties, e.g., load imbalance at a barrier in a certain process. The agent then requests certain measurements from the monitor linked into the application. Afterwards the application is released for one execution of the phase. At the end of the phase the execution is stopped again and the measured performance data are returned to the agent. Based on the measurements the agent evaluates the candidate properties and determines the set of found properties. If a refinement is possible, i.e., for the found properties more precise properties are available, a next analysis step is started with a new candidate property set.

Periscope provides search strategies for single-node performance, e.g., searching for inefficient use of the memory hierarchy, MPI, and OpenMP.

## Architecture

Periscope consists of an agent hierarchy shown in Fig. 1. The leaves in the hierarchy are the *analysis agents* that perform the actual performance analysis. Each analysis agent is responsible for a subset of the application's processes. It communicates with the monitor linked with the application processes. The MRI monitor serves



**Periscope. Fig. 1** Architecture of periscope

two purposes: It performs the measurements of performance data requested by the analysis agent and it controls the execution of the process which means, it takes commands from the agent to release or stop the execution.

The root of the hierarchy is the master agent which takes commands from the frontend, propagates the commands to the analysis agents, and provides the list of found performance properties to the frontend. The agents in the middle of the hierarchy are responsible for forwarding information among the analysis agents and the master agent.

The search is controlled by the frontend. It invokes the application and creates the agent hierarchy. How many agents are created depends on the number of application processes and of additional processors provided in the batch job. The processes and agents are mapped to the available processors in a way that communication is local with respect to the physical interconnection network topology.

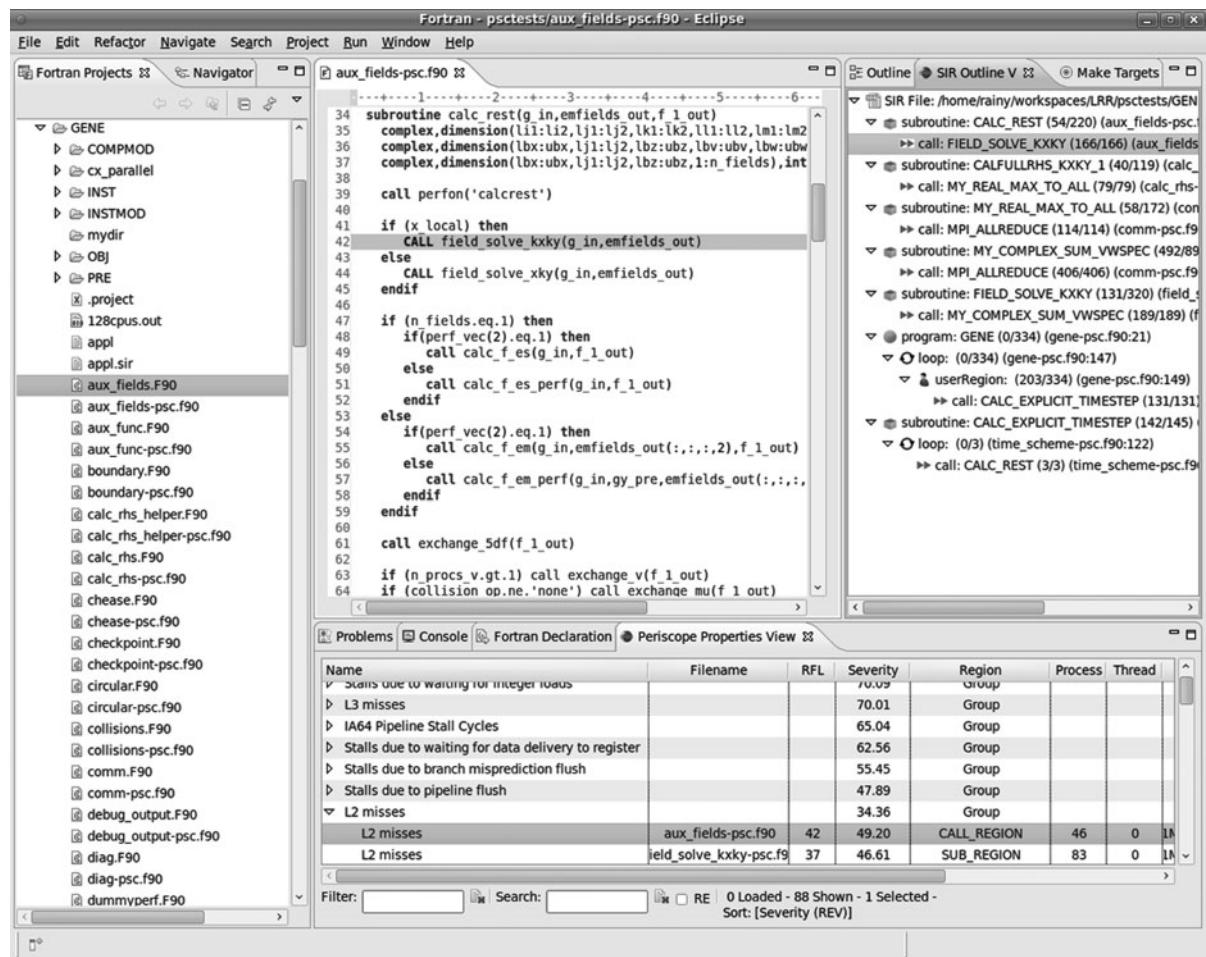
After the application and the agent hierarchy have been started, the frontend starts the search by propagating a command to all the analysis agents. If an agent requests a new experiment, i.e., an execution of the program phase with measurements to evaluate performance properties, the master agent starts the next experiment via another command. The reason for the global synchronization is that Periscope also supports *automatic restart*. If another experiment is requested but the application terminated, the application can automatically be restarted by the frontend.

On top of the frontend, Periscope provides a graphical user interface based on Eclipse and the Parallel Tools Platform (PTP). The GUI allows the programmer to define a project with all the source files, start a

performance analysis via the frontend and, most important, to investigate the performance properties found by Periscope.

Figure 2 illustrates Periscope's GUI. In the lower right a table view visualizes all the properties found in the application. It provides multi-criteria sorting to identify the most severe performance problems, filtering, regular expressions searching, multi-level grouping, as well as clustering. The clustering can be used to identify groups of processes with the same performance problems.

Double clicking on one of the properties will focus on the source code of the respective region in the central source editor. On the right, an outline view of the entire program is provided. It presents an overview of



**Periscope. Fig. 2** Inspection of performance properties with the graphical user interface of Periscope

the regions, the nesting of regions, and the number of properties found for a region. Selecting a region in the outline view can be used to enforce that only those properties found for that region are shown in the region view below.

## Summary

Periscope is an automatic performance analysis tool for large-scale parallel systems. It performs a distributed online search for performance properties. The properties are formalized and the set of properties can be easily extended. Specialized search strategies determine how the analysis agents walk the multidimensional search space. The result of a search is a set of high-level performance properties that can be inspected via a graphical user interface implemented as an Eclipse plugin. Therefore, the user can follow an integrated approach of program development in Eclipse and performance analysis with Periscope.

## Related Entries

- ▶ [Parallel Tools Platform](#)
- ▶ [Performance Analysis Tools](#)
- ▶ [Scalasca](#)
- ▶ [Tracing](#)

## Bibliographic Notes and Further Reading

A recent overview of Periscope can be found in [1]. The steps in applying Periscope to parallel applications as well as results obtained with Periscope are reported in [2]. Details on the graphical user interface can be found in [3]. More information on the formalization of performance properties with the APART Specification Language (ASL) can be found in [4].

## Bibliography

1. Gerndt M, Ott M (2010) Automatic performance analysis with Periscope. *Concurr Comput Pract Exp* 22(6):736–748
2. Benedict S, Petkov V, Gerndt M (2009) PERISCOPE: an online-based distributed performance analysis tool. In: 3rd parallel tools workshop, Dresden. Springer, Berlin, pp 1–16, ISBN 978-3-642-11261-4
3. Petkov V, Gerndt M (2010) Integrating parallel application development with performance analysis in periscope. In: 15th international workshop on high-level programming models and supportive environments (HIPS 2010), Atlanta, GA, April 19–23. Springer, Berlin, pp 1–8, ISBN 978-1-4244-6533-0
4. Fahringer T, Gerndt M, Riley G, Träff JL (2000) Specification of performance problems in MPI-programs with ASL. In: International conference on parallel processing (ICPP'00), Toronto. IEEE Computer Society, Washington, DC, pp 51–58

## Personalized All-to-All Exchange

- ▶ [All-to-All](#)

## Petaflop Barrier

- ▶ [Roadrunner Project, Los Alamos](#)

## Petascale Computer

Petascale computers are those capable of executing at least  $10^{15}$  floating point operations per second. This number of operations per second is known as a petaflop.

## Related Entries

- ▶ [Roadrunner Project, Los Alamos](#)
- ▶ [Top500](#)

## Petri Nets

JACK B. DENNIS  
Massachusetts Institute of Technology, Cambridge,  
MA, USA

## Synonyms

- [Place-transition nets](#)

## Definition

A *Petri Net* is a graph model for the control behavior of systems exhibiting concurrency in their operation. The graph is bipartite, the two node types being *places* drawn as circles, and *transitions* drawn as bars. The arcs of the graph are directed and run from places to transitions or vice versa. Each place may be empty, or hold a finite number of *tokens*. The state of a Petri net is the distribution of tokens on its places, called a *marking* of the net.

A transition is *enabled* if each of its input places holds at least one token. *Firing* a transition means removing one token from each input place and adding one token to each output place. A *run* of a Petri net is any sequence of firings of enabled transitions; a run defines a sequence of markings. Because many transitions may be enabled in a state, there are often many possible distinct runs of a Petri net. Hence, a Petri net represents a kind of nondeterministic state machine, but in a convenient form for modeling and analyzing concurrent systems. Various extensions and generalizations of Petri nets have been found useful in applications.

## Discussion

### Introduction

The study of Petri nets began in 1962 with the dissertation of Carl Adam Petri entitled *Communication with Automata*. Petri's work was promoted in the United States by Anatol Holt and Fred Commoner, who introduced the graphical form of Petri nets now widely used. Further development of properties and applications of Petri nets was done at the MIT Laboratory for Computer Science and elsewhere, and promulgated through workshop meetings held in 1970 and 1975. Extensions to Coloured Petri nets, Timed Petri nets, and Continuous Petri nets have been developed for modeling additional properties of concurrent systems. Methods based on Petri nets have become widely used for control aspects of all sorts of concurrent systems, from digital hardware through cooperating computation processes, to diverse applications in areas such as project management and even biological systems.

### Definition and Examples

Figure 1 shows a Petri net with three *places* (the circles) and three *transitions* (the bars) interconnected by directed arcs. The graph is bipartite, the two node types being places and transitions. The places may be empty, or may hold any finite number of *tokens*, represented by black dots. The distribution of tokens is called a *marking* of the Petri net and represents its state. Operation of a Petri net produces a sequence of markings by successive applications of the *firing rule*:

**Firing Rule:** A transition in a Petri net is *enabled* if each of its input places holds at least one token. *Firing* an enabled transition means removing one token from

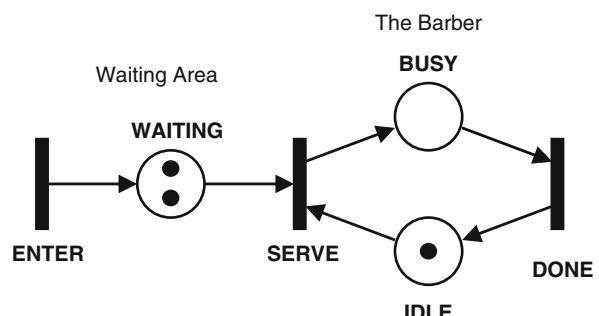
each of its input places and adding one token to each of its output places.

Note that, because more than one transition may be enabled in a marking of a Petri net, the net is, in general, a nondeterministic system, that is, many distinct firing sequences may exist starting from a given initial marking.

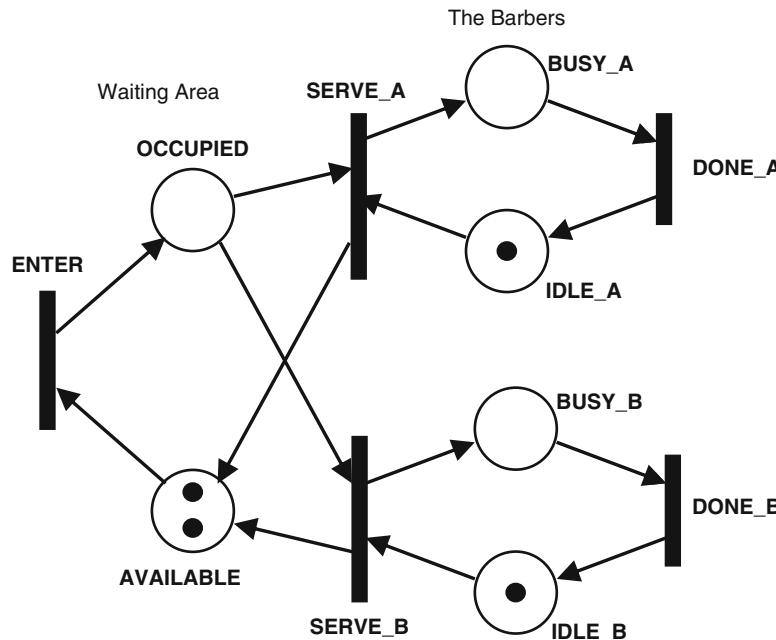
### Example: A Barber Shop

Figure 1 is a Petri net model for a barber shop. The net includes a place **WAITING** that models a waiting area where customers wait for the barber to serve them, and two places that model the barber's status, **BUSY** or **IDLE**. The transitions model events that can occur: A firing of transition **ENTER** corresponds to a customer entering the waiting area of the shop. Transition **SERVE** corresponds to a customer moving to the barber's chair for service. Transition **DONE** corresponds to the customer leaving the barber chair after receiving service, with the barber becoming **IDLE**. This model of the barber shop permits any number of customers to enter the waiting room, but models the constraint that only one customer can occupy the barber chair. This net is an *unbounded* Petri net because the number of tokens in the **WAITING** place can increase indefinitely.

Figure 2 shows a model of a barber shop with two additional features: there are now two barbers and the waiting area has bounded capacity. Place **AVAILABLE** models the number of empty spaces for customers in the waiting area and place **OCCUPIED** models the occupied spaces. This addition turns the model into a *bounded*



**Petri Nets. Fig. 1** The barber shop with one barber. Places are shown as circles; transitions are shown as bars. Two tokens represent customers in the waiting area. A third token represents the waiting state of the barber



Petri Nets. Fig. 2 The barber shop with two barbers. Either barber may serve a waiting customer

Petri net. Note that there are now two transitions that model a customer moving to a barber chair. These transitions share an input place and are said to be in *conflict*. This structure represents the nondeterminacy of deciding which barber serves the next customer in the case that both barbers are idle. A Petri net in which there can be no conflict is *determinate*, that is, its behavior is such that all firing sequences are equivalent, even though operation is nondeterministic.

If one wishes more detail about the discipline of the waiting area, it can be modeled as a FIFO queue, as shown in Fig. 3. This net contains as many stages as desired (the capacity of the queue), each stage modeling a single queued item (waiting customer).

The modeling techniques shown by the barber shop example illustrate application of Petri nets to such systems as pipelined processors and production systems for manufacturing or business workflow. Petri nets lack means to model the timing of actions represented by transition firings, but extensions have been developed to remedy this limitation (See below).

### Petri Nets and Finite State Machines

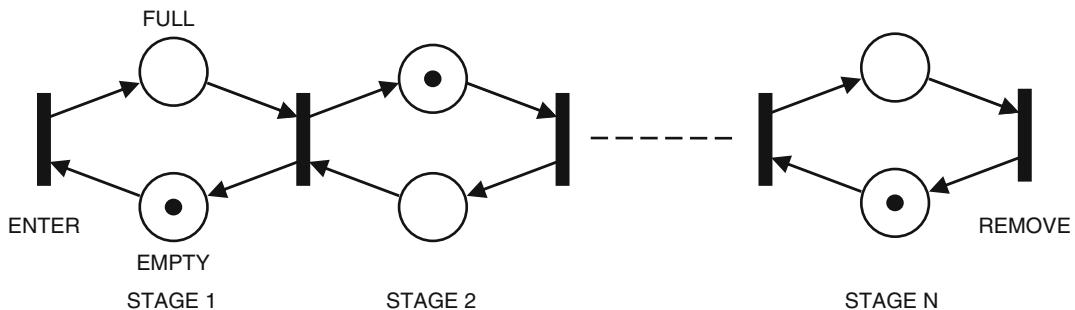
The Finite State Machine (FSM) model, a well-known tool used in the design and analysis of switching circuits and other engineering artifacts, lacks the ability

provided by Petri nets to capture the spatial structure of systems in a way that can aid study and analysis. This is illustrated by the FIFO queue of Fig. 3. By means of  $2N$  places and  $N$  transitions, a queue of length  $N$  may be represented, including the occupancy state of each position in the queue. An equivalent FSM description of the queue would have  $2^N$  states. This difference is a strong argument in favor of using Petri nets in system design and analysis.

Petri nets can exhibit some interesting properties. A Petri net with a marking may have no enabled transitions; no firings are possible and the net is in *deadlock*. Also, as seen in Fig. 1, a Petri net can have firing sequences in which the number of tokens in some places increases without limit. Two definitions restrict Petri nets to subclasses most useful for modeling certain kinds of realistic systems:

**Liveness:** A Petri net with marking  $M$  is *live* if and only if given any marking  $M'$  reachable from the given marking  $M$ , and any transition  $T$  of the net, there exists a firing sequence starting from  $M'$  that fires transition  $T$ .

**Safety:** A Petri net is  $N$ -*safe* for a given marking if and only if no marking reachable from the given marking has more than  $N$  tokens in any place. A net and marking that is *1-safe* is said to be *safe*.



**Petri Nets. Fig. 3** Model of a FIFO queue with stage 2 occupied

Note that the definition of live is formulated to rule out nets containing an initialization sequence that is never executed again in steady-state operation of the net.

A Petri net cannot be both live and  $N$ -safe for a marking unless the net is connected, that is, the net contains a directed path between any pair of nodes. The nets of Fig. 1 and Fig. 2 are both live as marked, but only Fig. 2 is safe because the **WAITING** place in Fig. 1 may accumulate tokens without limit. The class of live and safe Petri nets is especially interesting because these properties are characteristic of real systems that continue in operation indefinitely and are implementable with finite hardware.

### Conflict and Determinacy

*Conflict* in a Petri net occurs if, in some reachable marking, two transitions are both enabled and share an input place. A Petri net and marking for which no conflict is possible is *determinate*. The net in Fig. 1 has no conflict and is determinate; the net of Fig. 2 is not determinate because transitions **SERVE\_A** and **SERVE\_B** may be in conflict, representing the choice of barber in the case that both servers are idle.

### The Petri Net Hierarchy

A convenient hierarchy of classes of Petri nets having increasing modeling power may be specified by syntactic constraints on the structure of a net. The simplest of these subclasses is the *marked graphs*, which are Petri nets in which each place is an input place of exactly one transition and an output place of exactly one transition. These nets describe systems whose behavior is an endless repetition of a pattern of events. The name marked graphs is given to these nets because they may be drawn with each place, together with its input arc and output arc, represented as a simple arc, as shown in Fig. 4b for

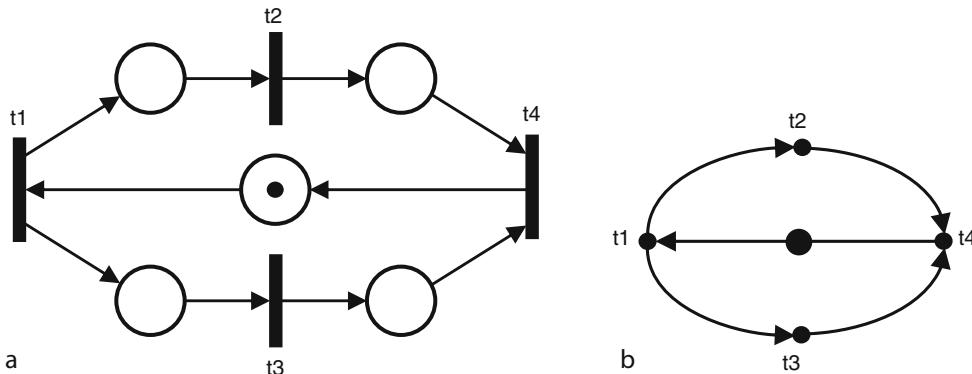
the net of Fig. 4a. Transitions are drawn as dots instead of bars, and a marking is shown by placing tokens on the arcs. A marked graph with an initial marking that is live must be connected, and can be shown to be  $N$ -safe for some  $N$ .

Another simple class of Petri nets is the *state machines*, which are nets in which each transition has exactly one input and one output place. A state machine with a one-token marking corresponds to an FSM with as many states as the net has places. A Petri net can be both a marked graph and a state machine. Such a net is very simple and consists of a set of disconnected cycles of alternating transitions and places.

Both marked graphs and state machines are determinate because conflict is impossible.

*Free choice* Petri nets constitute the next interesting level toward greater expressivity. Free choice nets permit a constrained form of conflict to be represented, corresponding to decisions in a digital system, or to choices based on predicates in a system description. A Petri net is a *free choice* net if and only if for each place  $P$  that is an input place of two transitions  $t_1$  and  $t_2$ ; place  $P$  is the only input place of  $t_1$  and  $t_2$ . This ensures that in any marking in which  $P$  holds at least one token, it is guaranteed that a free choice is available – both  $t_1$  and  $t_2$  are enabled and either may fire, disabling the other. Figure 1b is not a free choice net because the decision is conditioned by a barber being available. Free choice Petri nets that have conflict are not determinate for any live marking.

A Petri net is a *simple Petri net* if and only if it has no transition with more than one shared input place. Simple nets can represent systems in which arbitration occurs among several activities competing for shared resources, behavior not representable with free choice nets.



Petri Nets. Fig. 4 A Petri net (a) and its representation as a marked graph (b)

The subclasses of Petri nets form a hierarchy of strictly increasing expressive power:

**Marked Graphs   State Machines**  
**Free Choice Nets**  
**Simple Nets**  
**All Basic Petri Nets**

### Extensions

The basic Petri nets described above have found uses in describing digital systems, modeling process synchronization schemes using Dijkstra's P and V operations, and monitors, and for the control structures underlying dataflow graphs. Methods have been developed for converting Petri nets into logic designs. Outside the realm of computer science, Petri nets have found application in modeling manufacturing systems, among many other uses.

On the other hand, basic Petri nets are cumbersome and limited for application to many important problems in the analysis of concurrent systems such as performance analysis and job/work flow management. To make the formalism more generally useful, several extensions of basic Petri nets have been developed, and a rich body of formal analysis and application for these extensions has evolved. The most important extensions are Coloured Petri Nets, Timed Petri Nets, and Continuous/Hybrid Petri Nets.

### Coloured Petri Nets

In a Coloured Petri Net, each token may be labeled with a color from an arbitrary set of colors. The use of colored tokens permits more compact representations of large and complex systems. If transitions are permitted

to map the colors of input tokens to new colors of output tokens, a powerful model of arbitrary parallel computations is realized. If the set of colors is finite, then any colored Petri net has an equivalent basic Petri net, albeit possibly very large. Thus, the coloring of tokens does not increase the expressive power of basic nets. However, adding hierarchy to the colored Petri nets permits recursive computation to be directly represented.

### Timed Petri Nets

One important extension is the addition of explicit timing. One form of this extension is to associate a fixed time with each transition. The interpretation is that when an enabled transition fires, tokens are removed from its input places; then, at the specified later time, tokens are added to the output places of the transition. In this interpretation, a transition may have several instances of operation progressing simultaneously. This may be seen as a generalization of the Program Evaluation and Review Technique (PERT) of critical path planning for project scheduling. Association of time specifications with places and with arcs has also been studied. Analysis methods have been developed for determining bounds on the overall execution time of timed Petri nets. Another variation associates a probability distribution of operation times with each transition, leading to a generalization of queuing networks. Timed Petri nets have significant applications in performance analysis of computer systems, workflow analysis and manufacturing systems.

### Continuous and Hybrid Petri Nets

Another direction in extension of Petri nets is motivated by applications to industrial control and manufacturing

systems. In a *continuous Petri net*, tokens carry values that are nonnegative real numbers and are thought of as comprising an infinite number of *marks*. Transitions are permitted to fire continuously so long as each input place holds a positive, nonzero value. In such a net, a place can model, for example, the amount of liquid in a tank, the number parts in a bin, or the rate of traffic flow on a highway. In a hybrid Petri net, two kinds of places are distinguished: *continuous* places that hold real values, and discrete places that hold a nonnegative, integral number of tokens as in basic Petri nets. Transitions are also of two kinds. By means of a transition that has both a continuous place and a discrete place as inputs, turning a flow on and off can be modeled.

Many variations and extensions of the Petri net model have arisen in the years since the early developments. The principal record of these developments has been the series of conference proceedings published by Springer as *Lecture Notes in Computer Science* (LNCS) under the title Applications and Theory of Petri Nets.

## Related Entries

- [CSP \(Communicating Sequential Processes\)](#)
- [Data Flow Graphs](#)
- [Synchronization](#)

## Bibliographic Notes and Further Reading

The study of Petri nets began with the doctoral dissertation of Carl Adam Petri [7]. The graphical expression of Petri nets now widely used was presented by Holt and Commoner [3]. The MIT Project MAC report in which this paper is published includes annotated references to significant prior work in the field. The book by James Peterson [6] was the first exposition of the theory and applications of Petri nets in book form, and includes a good summary of the subject's early development and its literature. The survey paper of Murata [5] is an excellent summary of work on basic Petri nets through the 1980s. Timed Petri nets were first studied by Ramchandani [10] in 1973 and many versions of Petri nets dealing with system timing have been studied since [1]. The three volumes by Jensen provide a comprehensive treatment of "colored" Petri nets [4]. Continuous and hybrid Petri nets are treated by David and Alla [1], where a review of basic net theory and extensive bibliographic notes may be found. The principal current

record of advances and applications in Petri nets is the series of Conferences now titled "Applications and Theory of Petri Nets" [11]. An interesting web site is maintained by Carl Adam Petri and Wolfgang Reisig [8], and these two authors are the contributors of the Petri net entry in scholarpedia [9].

## Bibliography

1. David R, Alla H (2005) Discrete, continuous, and hybrid Petri Nets. Springer, Berlin
2. Girault C, Valk R (2003) Petri Nets for systems engineering. Springer, New York
3. Holt A, Commoner F (1970) Events and conditions. In: Record of the project MAC conference on concurrent systems and parallel computation. ACM, New York, pp 3–52
4. Jensen K (1995, 1996) Coloured Petri Nets: basic concepts, analysis methods and practical use. Monographs in theoretical computer science, vol 1, 2. Springer, Berlin
5. Murata T (Apr 1989) Petri nets: properties, analysis and applications. Proc IEEE 77(4):541–580
6. Peterson JL (1981) Petri Net theory and the modeling of systems. Prentice-Hall, Englewood Cliffs
7. Petri CA (1962) Kommunikation mit automaten. Schriften des Institutes für Instrumentelle mathematik. Ph.D. dissertation, University of Bonn, Germany
8. Petri CA. [http://www.informatik.uni-hamburg.de/TGI/mitarbeiter/profs/petri\\_eng.html](http://www.informatik.uni-hamburg.de/TGI/mitarbeiter/profs/petri_eng.html)
9. Petri CA, Reisig W (2008) Petri net. Scholarpedia 3(4):6477. [http://www.scholarpedia.org/article/Petri\\_net](http://www.scholarpedia.org/article/Petri_net)
10. Ramchandani C (Feb 1974) Analysis of asynchronous concurrent systems by Petri Nets. Technical Report MIT/LCS/TR-120, MIT Laboratory for Computer Science
11. Springer (1980) Application and theory of Petri Nets. In: Informatik fachberichte; (1984–1993) Advances in Petri Nets. Lecture notes in computer science, vol 10; (1992–2009) Applications and theory of Petri Nets. Lecture notes in computer science, vol 18. Springer, Berlin

## PETSc (Portable, Extensible Toolkit for Scientific Computation)

BARRY SMITH

Argonne National Laboratory, Argonne, IL, USA

## Definition

The Portable, Extensible Toolkit for Scientific computation (PETSc, pronounced PET-see) is a suite of

open source software libraries for the parallel solution of linear and nonlinear algebraic equations. PETSc uses the Message Passing Interface (MPI) for all of its parallelism.

## Discussion

The numerical solution of linear systems with sparse matrix representations is at the heart of many numerical simulations, from brain surgery to rocket science. These linear systems arise from the replacement of continuum partial differential equation (PDE) models with suitable discrete models by the use of the finite element, finite volume, finite difference, collocation, or spectral methods and then possibly linearization by a fully implicit strategy such as Newton's method or semi-implicit techniques. The resulting linear systems can range from having a few thousand unknowns to billions of unknowns, thus requiring the largest parallel computers currently available. Large-scale linear systems also arise directly in optimization, economics modeling, and many other non-PDE-based models. The main focus of PETSc is in solving linear systems arising from PDE-based models, though it is applied to other problems as well. PETSc also has limited support for dense matrix computations (through an interface to LAPACK and PLAPACK); but if the computation involves exclusively dense matrices, then PLAPACK or ScaLAPACK are appropriate libraries.

PETSc is a software library intended for use by mathematicians, scientists, and engineers with a solid understanding of programming, some basic understanding of the issues in parallel computing (though they need not have programmed in MPI), a basic understanding of numerical analysis, and an understanding of the basics of linear algebra. It has a higher and deeper learning curve than do software packages such as Matlab. PETSc can be used directly from Fortran 77/90, C/C++, and Python, with bindings that are specialized to each language.

PETSc is used by a variety of parallel PDE solver libraries, including freeCFD, a general-purpose CFD solver; OpenFVM, a finite-volume-based CFD solver; OOFEM, an object-oriented finite element library; libMesh, an adaptive finite element library; and DEAL.II, a sophisticated C++ based finite element simulation package. Magpar is a widely used,

parallel micromagnetics package written using PETSc. For large-scale optimization and the scalable computation of eigenvalues, PETSc has two companion packages, TAO and SLEPc, developed by other groups, that use all of the PETSc parallelism and linear solver infrastructure.

The emphasis of the PETSc solvers is on iterative methods for the solution of linear systems, but it provides its own efficient sequential direct (LU and Cholesky factorization-based) solvers as well as interfaces to several parallel direct solvers; see Table 1. PETSc has a unique configuration system that will automatically download and install the multitude of optional packages that it can use. In addition to the direct solvers, it can use several parallel partitioning packages as well as preconditioners in the hypre and TRILINOS solver packages; see Table 2. A key design feature of PETSc is the compositability of its linear solvers. Two or more solvers may be combined in various ways: by splittings, multigrid, and Schur complementing to produce efficient, problem-specific solvers.

The parallelism in PETSc is usually achieved by domain decomposition. The geometry on which the PDE is being solved is divided among the processes,

**PETSc (Portable, Extensible Toolkit for Scientific) Computation.** Table 1 Partial list of direct solvers available in PETSc

| Factorization | Package      | Complex numbers support | Parallel support |
|---------------|--------------|-------------------------|------------------|
| LU            | PETSc        | x                       |                  |
|               | SuperLU      | x                       |                  |
|               | SuperLU_Dist | x                       | x                |
|               | MUMPS        | x                       | x                |
|               | Spooles      | x                       | x                |
|               | PaStiX       | x                       | x                |
|               | IBM's ESSL   |                         |                  |
|               | UMFPACK      |                         |                  |
|               | LUSOL        |                         |                  |
| Cholesky      | PETSc        | x                       |                  |
|               | Spooles      | x                       | x                |
|               | MUMPS        | x                       | x                |
|               | PaStiX       | x                       | x                |
|               | DSCPACK      | x                       |                  |

**PETSc (Portable, Extensible Toolkit for Scientific)**

**Computation.** **Table 2** Partial list of preconditioners available in PETSc

| Preconditioner      | Package         | Complex numbers support | Parallel support |
|---------------------|-----------------|-------------------------|------------------|
| ICC(k)              | PETSc           | x                       |                  |
| ILU(k)              | PETSc           | x                       |                  |
|                     | Euclid/hypre    |                         | x                |
| ILUdt               | pilut/hypre     |                         | x                |
| Jacobi              | PETSc           | x                       | x                |
| SOR                 | PETSc           | x                       |                  |
| Block Jacobi        | PETSc           | x                       | x                |
| Additive Schwarz    | PETSc           | x                       | x                |
| Geometric multigrid | PETSc           | x                       | x                |
| Algebraic multigrid | BoomerAMG/hypre |                         | x                |
|                     | ML/TRILINOS     |                         | x                |
| Approximate inverse | SPAI            |                         | x                |
|                     | Parasails/hypre |                         | x                |

and each process is assigned the unknowns and matrix elements associated with that domain. The communication required during the solution process is then nearest neighbor ghost (halo) point updates and global reductions (using `MPI_Allreduce()`) over a MPI communicator. PETSc has optimized code based on the inspector-executor model to perform the ghost point updates.

In addition to its broad support for linear solvers, PETSc provides robust implementations of Newton's method for nonlinear systems of equations. These include a variety of line-search and trust-region schemes for globalization. The solvers are extensible, allowing easy provision of user-provided convergence tests, line-search strategies, and damping strategies. Several variants of the Eisenstat–Walker convergence criteria for inexact Newton solves are available. There is also support for grid sequencing to efficiently generate high-quality initial solutions for fine grids. To compute the Jacobians commonly needed for Newton's method, PETSc provides coloring of sparse matrices and efficient computation of the Jacobian entries using the coloring

with finite differencing, ADIC (automatic differentiation for C programs), and ADIFOR (automatic differentiation for Fortran 77 programs). All of these run scalably in parallel.

PETSc also provides a family of implicit and explicit ODE integrators, including an extensive suite of explicit Runge–Kutta methods. The implicit methods support all the functionality of the PETSc nonlinear solvers and use of any of the Krylov methods and preconditioners. The more sophisticated adaptive time-stepping ODE integrators of SUNDIALS can also be used with PETSc and allow use of all PETSc preconditioners.

Provided in PETSc is an infrastructure for profiling the parallel performance of the application and the solvers it uses, including floating-point operations done, messages, and sizes of messages sent and received. It provides the results in a table that indicates the percentage of time spent in the various parts of the solver and application.

Development of PETSc was started in 1995 by Bill Gropp, Lois Curfman McInnes, and Barry Smith at Argonne National Laboratory. They were joined shortly later by Satish Balay. Aside from a small amount of National Science Foundation funding in the mid-1990s, the US Department of Energy has provided the funding for PETSc development and support. Since its origin, PETSc has received software contributions from many of its users.

PETSc was the winner of a 2009 R&D 100 award. It also has formed the basis of three Gordon Bell Prize application codes in 1999, 2003, and 2004 as well as several Gordon Bell finalists.

**Library Design**

PETSc follows the distributed-memory single program multiple data (SPMD) model of MPI, with the flexibility of having different types of computation running on different processes. Specifically PETSc allows users to create their own MPI communicators and designate computations for PETSc to perform on each of these communicators. A typical application code written with PETSc requires very few MPI calls by the developer.

PETSc is written in C using the object-oriented programming techniques of data encapsulation, polymorphism, and inheritance. Opaque objects are defined that contain function tables (using C function pointers) used to call the code appropriate for the underlying

data structures. The six main abstract classes in PETSc are the **Vec** vector class for managing the system solutions, the **Mat** matrix class for managing the sparse matrices, the **KSP** Krylov solver class for managing the iterative accelerators, the **PC** preconditioner class, the **SNES** nonlinear solver class, and the **TS** ordinary differential equations (ODE) integrator class. The **DM** helper class manages transferring information about grids and discretizations into the **Vec** and **Mat** classes. Virtually all of the parallel communication required by PETSc (the MPI message passing and collective calls) takes place within these objects. The constructor for each PETSc object takes an MPI communicator, which determines on what processes the object and its computations will reside. The most common are **MPI\_COMM\_WORLD**, in which the object is distributed across all the user's processes (and computations involving the object will require communication within that communicator), and **MPI\_COMM\_SELF**, in which the object lives on just that process and no communication is ever required for its computations.

A typical application that requires linear solvers has a structure as depicted in Fig. 1. In this example, the **DA** object, which is an implementation of the **DM** class for structured grids, is used to construct the needed sparse matrix and vectors to contain the solution and right-hand side; it serves as a factory for **Vec** and **Mat** objects. Once the numerical values of the matrix are set, in this case by calls to **MatSetValues()**, the matrix is provided to the linear solver via **KSPSetOperators()**. Since **MatSetValues()** may be called with values that belong to any process, the calls to **MatAssemblyBegin/End()** are used to communicate the values to the process where they belong. Values may be set into vectors either by using **VecSetValues()**, with a concluding **VecAssemblyBegin/End()**, as with matrices, or by accessing the array of values using **VecGetArray()**, **VecGetArrayF90()**, or **DAVecGetArray()** and putting values directly into the array. In this latter case, no communication of off-process values is done by PETSc.

A typical application that requires nonlinear solvers has a structure as depicted in Fig. 2. In addition to serving as a factory for the Jacobian sparse matrix and solution vector (as in the linear case), the **DA** object is used as a factory for the ghosted representation of the solution **xlocal** and performs the ghost point updates with **DAGlobalToLocal()** in the routines

**ComputeFunction()** and **ComputeJacobian()**. These call-back routines are registered with the nonlinear solver object **SNES** with the routines **SNESSetFunction()** and **SNESSetJacobian()**. They are called when needed by the solver class.

A typical application that requires ODE integration has a structure as depicted in Fig. 3. This simple example uses the Python interface to **TS** where the entire discretized ODE (in this case using the backward Euler method) is provided directly as the function and Jacobian. It is also possible to provide the function and Jacobian of the right-hand side of the ODE, that is,  $u_t = F(u)$ , and have the **TS** class manage the ODE discretization, with either an explicit or an implicit scheme.

Each PETSc object has a method **XXXSetFromOptions()** that allows runtime control of almost all of the solver options through the PETSc options database. Command-line arguments (as keyword value pairs) are stored in a simple database. The **XXXSetFromOptions()** routines then search the options database, select any appropriate options, and apply them. For example, the option **-ksp\_type gmres** is used by **KSPSetFromOptions()** to call **KSPSetType()** to set the solver type to GMRES. The options database may also be used directly by user code.

Also common to all classes are the **XXXView()** methods. These provide a common interface to printing and saving information about any object to a **PetscView** object, which is an abstract representation of a binary file, a text file (like **stdout**), a graphical window for drawing, or a Unix socket. For example, **MatView(Mat A,PetscViewer v)** will present the matrix in a wide variety of ways depending on the viewer type and its state. Calling the viewer method on a solver class, such as **SNES**, displays the type of solver and all its options; see Fig. 4 for an example. Note that the figure displays both the nonlinear and linear solver options.

For the **Mat** class, PETSc provides several realizations. The most important of these are the following:

- Compressed sparse row (CSR) format
- Point-block version of the CSR where a single index is used for small dense blocks of the matrix
- Symmetric version of the point-block CSR that requires roughly one-half the storage
- User-provided format (via inheritance)

```

program main ! Solves the linear system J x = f
#include "finclude/petscalldef.h"
use petscksp; use petscda
Vec x,f; Mat J; DA da; KSP ksp; PetscErrorCode ierr
call PetscInitialize(PETSC_NULL_CHARACTER,ierr)

call DACreateId(MPI_COMM_WORLD,DA_NONPERIODIC,8,1,1,PETSC_NULL_INTEGER,da,ierr)
call DACreateGlobalVector(da,x,ierr); call VecDuplicate(x,f,ierr)
call DAGetMatrix(da,MATAIJ,J,ierr)

call ComputeRHS(da,f,ierr)
call ComputeMatrix(da,J,ierr)

call KSPCreate(MPI_COMM_WORLD,ksp,ierr)
call KSPSetOperators(ksp,J,J,SAME_NONZERO_PATTERN,ierr)
call KSPSetFromOptions(ksp,ierr)
call KSPSolve(ksp,f,x,ierr)

call MatDestroy(J,ierr); call VecDestroy(x,ierr); call VecDestroy(f,ierr)
call KSPDestroy(ksp,ierr); call DADestroy(da,ierr)
call PetscFinalize(ierr)
end
subroutine ComputeRHS(da,x,ierr)
#include "finclude/petscalldef.h"
use petscda
DA da; Vec x; PetscErrorCode ierr; PetscInt xs,xm,i,mx; PetscScalar hx; PetscScalar, pointer::xx(:)
call DAGetInfo(da,PETSC_NULL_INTEGER,mx,PETSC_NULL_INTEGER,PETSC_NULL_INTEGER,PETSC_NULL_INTEGER,...
hx = 1.d0/(mx-1)
call VecGetArrayF90(x,xx,ierr)
do i=xs,xs+xm-1
 xx(i) = i*hx
enddo
call VecRestoreArrayF90(x,xx,ierr)
return
end
subroutine ComputeMatrix(da,J,ierr)
#include "finclude/petscalldef.h"
use petscda
Mat J; DA da; PetscErrorCode ierr; PetscInt xs,xm,i,mx; PetscScalar hx
call DAGetInfo(da,PETSC_NULL_INTEGER,mx,PETSC_NULL_INTEGER,PETSC_NULL_INTEGER,PETSC_NULL_INTEGER,...
call DAGetCorners(da,xs,PETSC_NULL_INTEGER,PETSC_NULL_INTEGER,xm,PETSC_NULL_INTEGER,...
hx = 1.d0/(mx-1)
do i=xs,xs+xm-1
 if ((i .eq. 0) .or. (i .eq. mx-1)) then
 call MatSetValue(J,i,i,1d0,INSERT_VALUES,ierr)
 else
 call MatSetValue(J,i,i-1,-hx,INSERT_VALUES,ierr)
 call MatSetValue(J,i,i+1,-hx,INSERT_VALUES,ierr)
 call MatSetValue(J,i,i,2*hx,INSERT_VALUES,ierr)
 endif
enddo
call MatAssemblyBegin(J,MAT_FINAL_ASSEMBLY,ierr); call MatAssemblyEnd(J,MAT_FINAL_ASSEMBLY,ierr)
return
end

```

**PETSc (Portable, Extensible Toolkit for Scientific Computation).** Fig. 1 Example of linear solver usage in PETSc in Fortran 90

- “Matrix-free” representations, where the matrix entries are not explicitly stored, but instead matrix-vector products are performed by using one of the following:
  - Finite differencing of the function evaluations
  - Automatic differentiation of the function evaluations using either ADIC, for C language code, or ADIFOR, for Fortran 77 language code
  - User-provided C, C++, Fortran, or Python routine

```

static char help[] = "Solves -Laplacian u - exp(u) = 0, 0 < x < 1\n\n";
#include "petscda.h"
#include "petscsnes.h"

int main(int argc,char **argv) {
 SNES snes; Vec x,f; Mat J; DA da;
 PetscInitialize(&argc,&argv,(char *)0,help);

 DACreateId(PETSC_COMM_WORLD,DA_NONPERIODIC,8,1,1,PETSC_NULL,&da);
 DACreateGlobalVector(da,&x); VecDuplicate(x,&f);
 DAGetMatrix(da,MATAIJ,&J);

 SNESCreate(PETSC_COMM_WORLD,&snes);
 SNESSetFunction(snes,f,ComputeFunction,da);
 SNESSet Jacobian(snes,J,J,Compute Jacobian,da);
 SNESSetFromOptions(snes);
 SNESSolve(snes,PETSC NULL,x);

 MatDestroy(J); VecDestroy(x); VecDestroy(f); SNESDestroy(snes); DADestroy(da);
 PetscFinalize();
 return 0;
}

PetscErrorCode ComputeFunction(SNES snes,Vec x,Vec f,void *ctx) {
 PetscInt i,Mx,xm; PetscScalar *xx,*ff,hx; DA da = (DA) ctx; Vec xlocal;
 DAGetInfo(da,PETSC_IGNORE,&Mx,PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,...);
 hx = 1.0/(PetscReal)(Mx-1);
 DAGetLocalVector(da,&xlocal); DAGlobalToLocalBegin(da,x,INSERT_VALUES,xlocal); DAGlobalToLocalEnd(da,x,...);
 DAVecGetArray(da,xlocal,&xx); DAVecGetArray(da,f,&ff);
 DAGetCorners(da,&xs,PETSC NULL,PETSC NULL,&xm,PETSC NULL,PETSC NULL);

 for (i=xs; i<xs+xm; i++) {
 if (i == 0 || i == Mx-1) ff[i] = xx[i]/hx;
 else ff[i] = (2.0*xx[i] - xx[i-1] - xx[i+1])/hx - hx*PetscExpScalar(xx[i]);
 }
 DAVecRestoreArray(da,xlocal,&xx); DARestoreLocalVector(da,&xlocal); DAVecRestoreArray(da,f,&ff);
 return 0;
}

PetscErrorCode Compute Jacobian(SNES snes,Vec x,Mat *J,MatStructure *flag,void *ctx) {
 DA da = (DA) ctx; PetscInt i,Mx,xm; PetscScalar hx,*xx; Vec xlocal;
 DAGetInfo(da,PETSC_IGNORE,&Mx,PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,...);
 hx = 1.0/(PetscReal)(Mx-1);
 DAGetLocalVector(da,&xlocal); DAGlobalToLocalBegin(da,x,INSERT_VALUES,xlocal); DAGlobalToLocalEnd(da,x,...);
 DAVecGetArray(da,xlocal,&xx);
 DAGetCorners(da,&xs,PETSC NULL,PETSC NULL,&xm,PETSC NULL,PETSC NULL);

 for (i=xs; i<xs+xm; i++) {
 if (i == 0 || i == Mx-1) { MatSetValue(*J,i,i,1.0/hx,INSERT_VALUES); }
 else {
 MatSetValue(*J,i,i-1,-1.0/hx,INSERT_VALUES);
 MatSetValue(*J,i,i,2.0/hx - hx*PetscExpScalar(xx[i]),INSERT_VALUES);
 MatSetValue(*J,i,i+1,-1.0/hx,INSERT_VALUES);
 }
 }
 MatAssemblyBegin(*J,MAT_FINAL_ASSEMBLY); MatAssemblyEnd(*J,MAT_FINAL_ASSEMBLY); *flag = SAME_NONZERO...
 DAVecRestoreArray(da,xlocal,&xx); DARestoreLocalVector(da,&xlocal);
 return 0;
}

```

**PETSc (Portable, Extensible Toolkit for Scientific Computation).** Fig. 2 Example of nonlinear solver usage in PETSc in C

Because PETSc is focused on PDE problems, row-based storage of the sparse matrices (each process holds a collection of contiguous rows of the matrix) is satisfactory for higher-performance parallel matrix operations. Hence, all of PETSc's built-in sparse matrix implementations use this approach. Custom formats can be provided to handle parallelism for

"arrow-head" matrices where row-based distribution does not scale.

PETSc has the point-block-based storage of sparse matrices for faster performance. The speed of sparse matrix computations is essentially always strongly limited by the memory bandwidth of the system, not by the CPU speed. The reason is that sparse matrix

```

import sys, petsc4py
petsc4py.init(sys.argv)
from petsc4py import PETSc
import math

class MyODE:
 def __init__(self,da):
 self.da = da
 def function(self, ts,t,x,f):
 mx = da.getSizes(); mx = mx[0]; hx = 1.0/mx
 (xs,xm) = da.getCorners(); xs = xs[0]; xm = xm[0]
 xx = da.createLocalVector()
 da.globalToLocal(x,xx)
 dt = ts.getTimeStep()
 x0 = ts.getSolution()
 if xs == 0: f[0] = xx[0]/hx; xs = 1;
 if xs+xm >= mx: f[mx-1] = xx[xm-(xs==1)]/hx; xm = xm-(xs==1);
 for i in range(xs,xs+xm-1):
 f[i] = (xx[i-xs+1]-x0[i])/dt + (2.0*xx[i-xs+1]-xx[i-xs]-xx[i-xs+2])/hx - hx*math.exp(xx[i-xs+1])
 f.assemble()
 def jacobian(self,ts,t,x,J,P):
 mx = da.getSizes(); mx = mx[0]; hx = 1.0/mx
 (xs,xm) = da.getCorners(); xs = xs[0]; xm = xm[0]
 xx = da.createLocalVector()
 da.globalToLocal(x,xx)
 x0 = ts.getSolution()
 dt = ts.getTimeStep()
 P.zeroEntries()
 if xs == 0: P.setValues([0],[0],1.0/hx); xs = 1;
 if xs+xm >= mx: P.setValues([mx-1],[mx-1],1.0/hx); xm = xm-(xs==1);
 for i in range(xs,xs+xm-1):
 P.setValues([i],[i-1,i,i+1],[-1.0/hx,1.0/dt+2.0/hx-hx*math.exp(xx[i-xs+1]),-1.0/hx])
 P.assemble()
 return True # same_nz

da = PETSc.DA().create([9],comm=PETSc.COMM_WORLD)
f = da.createGlobalVector()
x = f.duplicate()
J = da.getMatrix(PETSc.MatType.AIJ);

ts = PETSc.TS().create(PETSc.COMM_WORLD)
ts.setProblemType(PETSc.TS.ProblemType.NONLINEAR)
ts.setType('python')

ode = MyODE(da)
ts.setFunction(ode.function, f)
ts.setJacobian(ode.jacobian, J, J)

ts.setTimeStep(0.1)
ts.setDuration(10, 1.0)
ts.setFromOptions()
x.set(1.0)
ts.solve(x)

```

**PETSc (Portable, Extensible Toolkit for Scientific Computation).** Fig. 3 Example of ODE usage in PETSc in Python

computations involve few operations per matrix entry. For example, for matrix-vector products there are two floating-point operations (a multiply and an addition) for each entry in the matrix. Memory-bandwidth-limited computations are sometimes said to hit the memory wall. In the CSR format, there is a column

index for every nonzero entry in the matrix, and the matrix-vector product is coded as  $y[i] = \sum_{j=nz_{i-1}}^{j<nz_i} aa[j] * x[aj[j]]$ . For each multiply in the computation, a double-precision value of  $aa[]$  must be loaded as well as an integer value  $aj[]$ . Thus, 12 bytes are loaded per multiply. In the point-block CSR format (with block

```

SNES Object:
 type: ls
 line search variant: SNESLineSearchCubic
 alpha=0.0001, maxstep=1e+08, minlambda=1e-12
 maximum iterations=50, maximum function evaluations=10000
 tolerances: relative=1e-08, absolute=1e-50, solution=1e-08
KSP Object:
 type: fgmres
 GMRES: restart=30, using Classical (unmodified) Gram-Schmidt Orthogonalization
 GMRES: happy breakdown tolerance 1e-30
 maximum iterations=10000, initial guess is zero
 tolerances: relative=1e-05, absolute=1e-50, divergence=10000
 right preconditioning
 using UNPRECONDITIONED norm type for convergence test
PC Object:
 type: mg
 MG: type is FULL, levels=2 cycles=v
 Coarse grid solver -- level 0 presmooths=1 postsmooths=1 -----
 KSP Object:(mg_coarse_)
 type: preonly
 PC Object:(mg_coarse_)
 type: lu
 LU: out-of-place factorization
 matrix ordering: nd
 LU: tolerance for zero pivot 1e-12
 LU: factor fill ratio needed 1.875
 Matrix Object:
 type=seqaij, rows=64, cols=64
 total: nonzeros=1024, allocated nonzeros=1024
 using I-node routines: found 16 nodes, limit used is 5
 Down solver (pre-smoother) on level 1 smooths=1 -----
 KSP Object:(mg_levels_1_)
 type: gmres
 GMRES: restart=30, using Classical (unmodified) Gram-Schmidt Orthogonalization
 GMRES: happy breakdown tolerance 1e-30
 maximum iterations=1
 tolerances: relative=1e-05, absolute=1e-50, divergence=10000
 left preconditioning
 using nonzero initial guess
 using PRECONDITIONED norm type for convergence test
 PC Object:(mg_levels_1_)
 type: ilu
 ILU: 0 levels of fill
 ILU: factor fill ratio allocated 1
 ILU: tolerance for zero pivot 1e-12
 Matrix Object:
 type=seqaij, rows=196, cols=196
 total: nonzeros=3472, allocated nonzeros=3472
 using I-node routines: found 49 nodes, limit used is 5
 Matrix Object:
 type=seqaij, rows=196, cols=196
 total: nonzeros=3472, allocated nonzeros=3472
 using I-node routines: found 49 nodes, limit used is 5

```

PETSc (Portable, Extensible Toolkit for Scientific Computation). Fig. 4 Example of output using **SNESView()**

size  $bs$ ), there is one column index per block, and the matrix-vector product may be coded as  $y[bs * i + k] = \sum_{j=nz_i}^{j<nz_i} \sum_{l=0}^{l<bs} aa[bs * (j + l) + k] * x[aj[j] + l]$ . Here, for every  $(bs * bs)$  multiplies,  $(bs * bs)$  loads of  $aa[]$  are needed, but only a single integer  $aj[]$ . For even moderate block size, this approach reduces the loads per multiply from 12 to less than 8.5 bytes. In addition, the same  $x[]$  values are used repeatedly for each  $k$ , and a smart

unrolling can keep the reused values in registers. Using the block CSR when appropriate, depending on the particular processor, can improve the performance of the sparse matrix operators by a factor of 2 to 3.

The **KSP** Krylov accelerator class provides over a dozen Krylov methods; see Table 3. The data encapsulation and polymorphic design of the **Vec**, **Mat**, and **PC** classes in PETSc allow the immediate use of any of their

**PETSc (Portable, Extensible Toolkit for Scientific Computation).** **Table 3** Partial list of Krylov methods available in PETSc

|                                                              |
|--------------------------------------------------------------|
| Richardson (simple) iteration, $x^{n+1} = x^n + B(b - Ax^n)$ |
| Chebychev iteration                                          |
| Conjugate gradient method                                    |
| Biconjugate gradient                                         |
| Biconjugate gradient stabilized (bi-CG-stab)                 |
| Conjugate residuals                                          |
| Conjugate gradient squared                                   |
| Minimum residuals (MINRES)                                   |
| Generalized minimal residual (GMRES)                         |
| Flexible GMRES (fgmres)                                      |
| Transpose-free quasi-minimal residuals (QMR)                 |

implementations with any of the Krylov solvers. When possible, these are implemented to allow left, right, or symmetric preconditioning and the use of various norms of the residual in the convergence tests including the “natural” (energy) norm. Custom convergence tests and monitoring routines can be provided to any of the solvers.

The **PC** preconditioners class contains a variety of both classical and modern preconditioners including incomplete factorizations, domain decomposition methods, and multigrid methods. See [Table 2](#) for a partial list. In addition, several preconditioner classes are designed to allow composition of solvers. These include **PCKSP**, which allows using a Krylov method as a preconditioner; **PCFieldSplit**, which allows constructing solvers by composing solvers for different fields of the solution; **KSPCOMPOSITE**, which allows combining arbitrary solvers; and **PCGALERKIN**, which constructs preconditioners by the Galerkin process (that is, as projections of the error in some appropriate inner product). **PCFieldSplit** preconditioners are often called block preconditioners; for example, when one field is velocity and another pressure, the resulting Stokes solver is often solved with one block for velocity and one for pressure.

## Applications

A wide variety of simulation applications have been written by using PETSc. These include fluid flow for

aircraft, ship, and automobile design; blood flow simulation for medical device design; porous media flow for oil reservoir simulation for energy development and groundwater contamination modeling; modeling of materials properties; economic modeling; structural mechanics for construction design; combustion modeling; and nuclear fission and fusion for energy development.

PETSc-FUN3d was an early application based on Kyle Anderson’s NASA code, FUN3d, that solves the Euler and Navier–Stokes equations including both compressible and incompressible on unstructured grids. PETSc-Fun3d won a Gordon Bell special prize in 1999 running on over 6,000 of the ASCI Red processors. This application, the dissertation work of Dinesh Kaushik, motivated many of the early optimizations of PETSc.

The forward and inverse modeling of earthquakes using the PETSc algebraic solvers, developed by Volkan Akcelik, Jacobo Bielak, George Biros, Ioannis Epanomeritakis, Antonio Fernandez Omar Ghattas, Eui Joong Kim, David O’Hallaron, and Tiansai Tu, resulted in a 2003 Gordon Bell special prize.

The algebraic multigrid solver Prometheus was written by Mark Adams using the PETSc **Vec**, **Mat**, **KSP**, and **PC** classes. It takes advantage of the block CSR sparse format in PETSc to maximize performance. It was to simulate whole-bone micromechanics with over half a billion degrees of freedom, resulting in a 2004 Gordon Bell special prize.

PETSc has been used by several research groups to simulate heart arrhythmias, which are the cause of the majority of sudden cardiac deaths. These applications involve solving the nonlinear bidomain equations, which are two coupled partial differential equations that model the intracellular and extracellular potential of the heart. Numerical solutions to these equations (and more sophisticated models) explain much of the electrical behavior of the heart, including defibrillation.

PFLOTRAN, led by Peter Lichtner of Los Alamos National Laboratory, is a subsurface flow and contaminant transport simulator that uses the PETSc **DM** class to manage the parallelism of its mesh, the **SNES** nonlinear solver class for the solutions needed at each time step, the **Mat** class to contain the sparse Jacobians, and the **Vec** class for its flow and contaminant’s solutions. It has been run on up to 64,000 cores of the Cray XT5

and has been used to more accurately model uranium plumes at DOE's Hanford site.

The UNIC neutronics package developed by Mike Smith and Dinesh Kaushik of Argonne National Laboratory has run full reactor core simulations on 290,000 cores of the IBM Blue Gene/P. It supports both the second-order Pn and Sn methods with dozens of energy groups. It parallelizes simultaneously over the geometry by means of domain decomposition and angles using a hierarchy of MPI communicators and PETSc solver objects.

## Related Entries

- ▶ [Algebraic Multigrid](#)
- ▶ [BLAS \(Basic Linear Algebra Subprograms\)](#)
- ▶ [Chaco](#)
- ▶ [Distributed-Memory Multiprocessor](#)
- ▶ [Domain Decomposition](#)
- ▶ [LAPACK](#)
- ▶ [Memory Wall](#)
- ▶ [METIS and ParMETIS](#)
- ▶ [MPI \(Message Passing Interface\)](#)
- ▶ [PLAPACK](#)
- ▶ [Scalability](#)
- ▶ [ScaLAPACK](#)
- ▶ [SPAI \(SParse Approximate Inverse\)](#)
- ▶ [SPMD Computational Model](#)
- ▶ [SuperLU](#)

## Bibliographic Notes and Further Reading

The PETSc web site is the best location for up-to-date information on PETSc [8]. A complete list of external packages that PETSc can use is given in [5]. More details of the applications developed by using PETSc can be found at [7]. Further details on the design decisions made in PETSc may be found in [2].

Other related parallel solver packages include TRILINOS [9], hypre [10], and SUNDIALS [11]. TRILINOS is a large, general-purpose solver package much in the spirit of PETSc and written largely in C++; it currently has little support for use from Fortran. The hypre package specializes in high-performance preconditioners and includes a scalable algebraic multigrid

solver BoomerAMG. SUNDIALS specializes in nonlinear solvers and adaptive ODE integrators; it expects the required linear solver to be provided by the user or another package. Many of the solvers in these other packages can be called through PETSc.

## Bibliography

1. Balay S, Buschelman K, Eijkhout V, Gropp WD, Kaushik D, Knepley MG, McInnes LC, Smith BF, Zhang H (2008) PETSc Users Manual, Argonne National Laboratory Technical Report ANL0-95/11 - Revision 3.0.0
2. Balay S, Gropp WD, McInnes LC, Smith BF (1997) Efficient management of parallelism in object oriented numerical software libraries. In: Arge E, Bruaset AM, Langtangen HP (eds) Modern software tools in scientific computing. Birkhauser Press, Boston, pp 163–202
3. Balay S, Gropp WD, McInnes LC, Smith BF (2002) Software for the scalable solution of PDEs. In: Dongarra J, Foster I, Fox G, Gropp B, Kennedy K, Torczon L, White A (eds) CRPC handbook of parallel computing. Morgan Kaufmann Publishers
4. J Dongarra's freely available software for linear algebra. <http://www.netlib.org/utk/people/JackDongarra/la-sw.html>
5. List of external software packages available from PETSc. <http://www.mcs.anl.gov/petsc/petsc-as/miscellaneous/external.html>
6. Saad Y (2003) Iterative methods for sparse linear systems, 2nd edn. SIAM
7. Partial list of applications written using PETSc. <http://www.mcs.anl.gov/petsc/petsc-as/publications/petscapps.html>
8. PETSc's webpage. <http://www.mcs.anl.gov/petsc>
9. TRILINOS's webpage. <http://trilinos.sandia.gov>
10. Hypre's webpage. [https://computation.llnl.gov/casc/linear\\_solvers/sls\\_hypre.html](https://computation.llnl.gov/casc/linear_solvers/sls_hypre.html)
11. SUNDIALS's webpage. <https://computation.llnl.gov/casc/sundials/main.html>

## PGAS (Partitioned Global Address Space) Languages

GEORGE ALMASI  
IBM, Yorktown Heights, NY, USA

### Definition

PGAS (Partitioned Global Address Space) is a programming model suited for shared and distributed memory parallel machines, e.g., machines consisting of many (up to hundreds of thousands of) CPUs.

Shared memory in this context means that the total of the memory space is available to every processor in the system (although access time to different banks of this memory can be different on each processor). Distributed memory is scattered across processors; access to other processors' memory is usually through a network.

A PGAS system, therefore, consists of the following components:

- A set of processors, each with attached local storage. Parts of this local storage can be declared *private* by the programming model, and is not visible to other processors.
- A mechanism by which at least a part of each processor's storage can be *shared* with others. Sharing can be implemented through the network device with system software support, or through hardware shared memory with cache coherence. This, of course, can result in large variations of memory access latency (typically, a few orders of magnitude) depending on the location and the underlying access method to a particular address.
- Every shared memory location has an *affinity* – a processor on which the location is local and therefore access is quick. Affinity is exposed to the programmer in order to facilitate performance and scalability stemming from “owner compute” strategies.

All PGAS programming languages contain the components enumerated above, although the ways in which these are made available to the programmer differ. Every PGAS language allows programmers to distinguish between private and shared memory locations, and to determine the affinity of shared memory locations. Some PGAS languages provide work distribution primitives such as parallel loops based on affinity, or program syntax to allow special handling of remote (and therefore, slow) data accesses. The rest of this entry expands some of these differences between PGAS languages.

## Discussion

### Introduction

There exist a variety of choices for PGAS languages and implementations. Some of these choices are about the

ubiquity of shared memory, the method of accessing remote memory, or the choice of a parent programming language. Consequently there is a wide variety of PGAS-like languages and libraries:

- UPC [9] (Unified Parallel C) is a language descended from C. It extends C arrays and pointers with shared arrays and shared pointers that address into global memory. UPC also features a `forall` loop that distributes iterations based on affinity of array elements.
- Coarray Fortran [5] is a Fortran-based language that extends Fortran arrays with co-dimensions that allow accessing arrays on other processes (called images). A variant of Coarray Fortran is included in the Fortran 2008 standard, making it the only PGAS language with ISO approval.
- Split-C [8] is a C-based PGAS language that acknowledges the latency of remote memory accesses by allowing split-phase, or non-blocking, transactions. This allows overlapping of remote accesses with computation, hiding latency.
- Titanium [11] is a Java-based PGAS language. Titanium features SPMD parallelism, pointers to shared data and an advanced distributed array model.
- ZPL [1] is an array-based language featuring the global view programming model.
- Chapel [2] is Cray Inc's flagship modern programming language. It incorporates elements of ZPL but also features the multiresolution paradigm, allowing users to bore down to performance from an initial high-level program.
- X10 [10] is a PGAS language that provides task parallelism as well as data parallelism. The key feature of X10 is asynchronous task dispatching.
- HPF [3] (High-Performance Fortran) is an early attempt to solidify concepts from global view array programming in a Fortran-based language. It is one of the bases from which the PGAS concepts grew.
- MPI [7] (Message Passing Interface) is the de facto standard for high-performance parallel programming. It does not implement the PGAS programming model, since it does not have the concept of global memory: All inter-processor data exchange is explicit. However, MPI contains many ideas and concepts relevant to PGAS and that makes it worth mentioning in this context.

- OpenMP [6] is a cross-language standard for shared-memory programming used widely in the high-performance computing world. The standard allows loops to be annotated as executed in parallel, and variables as shared or private; the newer standard has task-parallel features as well. OpenMP is in a similar situation to MPI: not a PGAS language, but containing many relevant concepts.
- Global Arrays [4] is a library or parallel array computing. It provides an abstraction of a shared array but is backed by distributed memory. Actual memory operations are implemented by a one-sided messaging library called ARMCI.
- HTAs (Hierarchical Tiled Arrays) are another library-based approach, providing the user with an array abstraction embedded into the multiple levels of a distributed system's memory hierarchy. HTAs can be laid out to reflect this hierarchy: levels of cache, shared memory with affinity to particular processors, and of course nonlocal memory accessed (under the covers) by messaging.

## Local Versus Shared Memory

While all PGAS languages distinguish between local, shared local and shared remote memory. However, the default assignment of memory to the local versus shared space greatly varies across the space of languages.

All memory in MPI (the Message Passing Interface standard) is local, and the only way to convey information to another process space is through messages. In contrast all memory in OpenMP (a GAS programming paradigm) is global, and the only way to make memory locations safe from other threads is to explicitly denote it as thread private. In UPC, Coarray Fortran and Split-C memory is declared as private by default, and has to be made global with an explicit declaration modifier. In Titanium program stacks are thread-private, but the heap is shared by default. In the array language ZPL and in the HTA library all arrays are shared by default. In X10, memory is local and only accessible by sending units of work ("asyncs") to the remote locations to execute.

## Computation and Address Spaces

Parallelism implies multiple processing units executing a particular program. However, the relationship between executing programs and address spaces differs

across programming languages. In UPC and Coarray Fortran address spaces are tightly bound to computation. Execution units are called threads in UPC; Address affinity is calculated relative to UPC threads. Titanium calls the execution units processes, and locality is bound to these implicitly. By contrast, in Coarray Fortran it is the address spaces themselves that are named – images – and the implication is that each image has computation executing on it. X10 completely separates the notions of address space and computation. Every address space is called a place, and multiple computational threads called activities are allowed to execute simultaneously, subject to the capability of the hardware.

## Messaging

The PGAS programming model does not make any representation about the mechanics of accessing data in nonlocal address spaces. On distributed-memory hardware data exchange is done by exchanging messages across any network devices available on the hardware in question; PGAS programming models are implemented on top of a messaging system.

The preferred messaging system for PGAS implementations is *one sided*: That is, one of the participants is active and is responsible for specifying all parameters of the exchange (identities of sender, receiver, addresses on both ends, amount of data), while the other participant is passive and contributes nothing but the data itself.

*Active messages* are also used preferentially by PGAS languages. Active messages vary from one-sided messages in that the passive participant is called upon to execute user code as part of receiving the message.

Every PGAS language makes a choice as to what extent language syntax hides the underlying messaging system. In Split-C messages look like assignments, and provisions are made to hide the large latency of such messages. In UPC, local and shared assignments have the same syntax, making the indistinguishable; however, the programmer is allowed to write explicit one-sided messages into the program. Even third-party messages are allowed (e.g., UPC thread A specifying a data transfer between threads B and C). Coarray Fortran and Chapel do not allow explicit messaging. X10 exposes messaging to the programmer in the form of asyncs which are very close in concept to active messages.

## References to Remote Memory

Just as in the C language arrays and pointers are two sides of the same coin, in PGAS languages there is a close relationship between arrays and references in global address space especially in those languages rooted in C syntax, like Split-C and UPC. The syntax and semantics of references to remote memory, including pointer arithmetic, tends to follow that of normal pointers. The unique features of remote pointer access revolve around hiding of access latency. Remote accesses tend to be orders of magnitude slower than local ones. The increased latency can be partially mitigated by posting remote operations as soon as the initial conditions are met, e.g., both source data and destination buffers are ready for transfer. However, the operation need not be complete until the data is actually needed on the destination end. To implement this, Split-C features the split assignment operator, and the Berkeley UPC extensions (not part of the UPC standard) allow non-blocking remote memory operations.

## Array Programming and Implicit Parallelism

Array programming is a generic term describing a programming environment suitable for the processing of n-dimensional arrays. In these environments arrays are first-class citizens, allowing compact declaration and operators (unlike in conventional imperative programming languages where arrays are handled by loops). Some examples of array programming languages/environments are APL, Fortran 90, MATLAB, and R.

The attraction of array languages is their ability to express operations on large amounts of data with few instructions. This has many benefits, including efficient programming of vector processors (e.g., Intel SSE3, IBM Altivec) and graphics processors (NVIDIA GPUs), but array languages also lend themselves to explicit SPMD parallelism with the PGAS programming model. The programmer specifies the layout of array elements in distributed memory. The compiler and/or the runtime optimize array operations by staying as close as possible to the owner compute rule, i.e., scheduling computation on the CPUs closest to each array element. The execution model of pure array languages is SIMD; conceptually there is a single thread of control acting on a large amount of data. PGAS languages have

borrowed heavily from the array processing paradigm. The Fortran D and High-Performance Fortran (HPF) languages allow users to specify data layouts with the TEMPLATE and DISTRIBUTE commands. An HPF template declares a processor layout (and hence the structure of the partitioned address space). Global arrays are distributed across this template. A large set of intrinsic operators allow the concise expression of operations like shifting/transposing/summing up array slices.

In Coarray Fortran, array data distribution takes the form of a co-dimension. Vector indexing in the Fortran 90 style is permitted. The Chapel and ZPL languages offer a refinement of the Fortran 90/Matlab vector syntax by means of regions, or named subsets/slices of arrays: shifts, reductions, dimensional floods (i.e., broadcasts), boundary exchanges can be expressed this way. The partitioned global address space is set up by means of distributions, an analogue of HPF templates. The Titanium programming language also follows this approach.

Less conventional runtime-only approaches include the Hierarchical Tiled Arrays (HTAs) library a pure runtime solution that provides multiple levels of data decomposition, one for each level of non-locality in a modern computer architecture. The Global Arrays toolkit also allows programmers to specify and optimize their own array layouts. The Matlab Parallel Toolbox uses the `spmd` keyword and specialized array distribution syntax to control data parallel execution.

There is a natural affinity between array processing and parallelism. By putting arrays into global memory one transcends the memory limitations of any single CPU, while still allowing for quick access to the array from anywhere. Array operations are generally floating-point intensive, and therefore natural candidates for parallelization. The large number of operations causes more granular computation, resulting in less parallelism overhead and therefore fewer losses to Amdahl's law.

Well-known parallel algorithms exist for many array operators. Some of these algorithms have good scaling properties (i.e., low cross-CPU communication requirements, good load balance) and can be coded into the supporting runtime system or even the compiler, allowing the programmer instant access to high-performance parallel array operations.

## Parallel Loops and Explicit Data Parallelism

The parallel loop construct is an established way of expressing explicit parallelism; Fortran's DOALL statement is one of the oldest such constructs. The essence of the construct is to divide the iteration space of a loop nest among processors, either statically or dynamically. OpenMP in particular is known for a wide variety of parallel loop options.

Several PGAS languages have their own versions of parallel loops. Perhaps the most prominent of these is the UPC forall construct which ties execution of particular iterations to an affinity expression that can depend on the induction variable of the loop. ZPL, Chapel, X10, and Titanium allow parallel loops to be run on affinity sets which implicitly determine which CPU executes what iteration.

## Collectives, Teams, and Synchronization

Collective operations in parallel programming languages denote operations that potentially involve more than two participants. Collective communication concepts were popularized by MPI, although basic ideas like parallel prefix are considerably older.

Collectives are important in the context of parallel programming models for two major reasons. First, collective communication primitives succinctly express complex data movement operations, contributing to brevity and clarity in parallel programs. Second, because of their relatively simple and well-studied semantics, collectives are good optimization targets, resulting in improved performance and scalability.

Collective operations are either pure data exchange protocols (such as broadcast, scatter, and all-to-all exchanges), or computational collectives (like reductions, where data are interpreted and recomputed during the collective).

Another way to describe collectives is based on whether they have synchronizing properties. For example, the Alltoall collective causes synchronization between every pair of tasks involved, since completion of the collective involves bidirectional data dependencies on every pair. Other collectives, like Scatter, create much fewer data dependencies and therefore do not cause global synchronization. Finally, Barrier is an example of a collective that exchanges no data at all; its only purpose is to effect a synchronization.

Collective communication is further categorized by the number of participants. The simplest case is that of every task in a job participating in a collective. However, arbitrary teams of tasks (called communicators in MPI) can be set up for collective communication.

There are several intriguing aspects that cause the mapping of collective communication to be nonobvious in a PGAS context. The most immediate of these involves data integrity. A natural way to think about data integrity in collective communication is as follows: Data buffers passed from the caller to a collective cannot be touched (either read or written) by the user until the collective completes. In other words, data buffers' ownership changes when the collective is invoked and when it terminates.

However, on a system with shared memory and one-sided data access the invocation boundary is fuzzy. The collective may be entered at different times on different processors. For example, in the presence of one-sided communication the calling process is unable to provide a strong guarantee to the collective that the data buffer will not be touched – since other processes may not yet have entered the collective and may be in the middle of a remote update to the very buffer being processed by the collective.

UPC attempts to deal with this problem by allowing the programmer to state pre- and post-conditions on the boundaries of a collective operation with regard to shared data.

Just like the PGAS model extends point-to-point communication with non-blocking and split-phase transactions, a similar extension can be envisaged for collective communication. The evident advantage of non-blocking collective communication is the ability to overlap it with computation or other communication. There is a case to be made for one-sided collectives. Far from a contradiction in terms, one-sided collective communication involves the address spaces of multiple tasks, but possibly not every task participates actively in the collective. An example would be a one-sided broadcast similar to a one-sided write operation but targeting multiple address spaces.

Some PGAS languages, like Coarray Fortran, feature synchronization operation on teams of tasks designated on an ad hoc basis. This extends the MPI notion in which MPI communicators are predefined

and relatively heavyweight objects. Collective communication exists in some of the PGAS languages today. Titanium features teams and exchange, broadcast, reduction collectives. UPC has a usual complement of collectives but no teams. Coarray Fortran features ad hoc teams in its synchronization operations. Many array languages feature array operations that are essentially “syntax sugar” for collective operations.

## Memory Consistency

The memory consistency model of PGAS programs deals with the effect of writing to remote memory; i.e., under what conditions does a remote write become visible by the source, destination, or third parties. The gold standard for memory consistency is sequential consistency: In this model, the memory behaves as if it were written by a single processor at a time.

Sequential consistency is expensive to implement in a distributed memory system because performance can be gated by the slowest write. Therefore, most PGAS languages implement a weak consistency model. For example, Coarray Fortran’s consistency model is designed to avoid conflicts and allows compiler optimization. Ordering of memory accesses made to remote locations is done explicitly by the programmer by breaking the program into ordered segments. Conflicting writes in the same segment are disallowed: The basic constraint is that if a variable is defined in a segment, it cannot be read or written by any other image in the same segment. UPC has two memory consistency modes, strict and relaxed, where strict consistency is understood to be sequential consistency. In Titanium, local dependencies are observed. Shared reads and writes performed in critical sections cannot appear to have executed outside.

## Future Trends

The future of the Partitioned Global Address Space programming model is difficult to predict. A variety of programming languages based on the model have been proposed, none meeting with universal approval. The state of the art in parallel programming continues to be MPI and OpenMP programming; it is safe to say that the programming model has not yet fulfilled its promise.

The face of parallel computing is continuously changing. While single processor performance has stopped following Moore’s law, peak performance on the Top500 website continues to track an exponential

growth curve. This growth is achieved by machines with hybrid (shared and distributed memory) architectures, forcing a change in programming technology. Also, an increasing share of high-performance programming also targets hybrid architectures of another kind: dedicated accelerators based on, e.g., GPU compute engines. OpenMP and MPI face some difficulty in coping with these challenges, and may leave the field open for new software technology.

Recognizing that PGAS languages are unlikely to replace MPI, the current trend is to enhance interoperability, allowing coexistence of multiple languages in the same executable. The challenge is both conceptual and practical, and includes reconciliation of the execution models, data representations, and execution semantics of different programming models. On a practical level, the trend is toward shared infrastructure with MPI and an expression of PGAS functionality through library calls to enable a multiplicity of language implementations.

## Related Entries

- ▶ [Array Languages](#)
- ▶ [Chapel \(Cray Inc. HPCS Language\)](#)
- ▶ [Coarray Fortran](#)
- ▶ [Collective Communication](#)
- ▶ [Fortress \(Sun HPCS Language\)](#)
- ▶ [Global Arrays Parallel Programming Toolkit](#)
- ▶ [HPF \(High Performance Fortran\)](#)
- ▶ [Memory Models](#)
- ▶ [MPI \(Message Passing Interface\)](#)
- ▶ [OpenMP](#)
- ▶ [SPMD Computational Model](#)
- ▶ [Titanium](#)
- ▶ [UPC](#)
- ▶ [ZPL](#)

## Bibliography

1. Chamberlain BL, Choi S-E, Christopher Lewis E, Lin C, Snyder L, Weathersby D (2000) ZPL: a machine independent programming language for parallel computers. *Softw Eng* 26(3):197–211
2. The cascade high productivity language. *HIPS*, 00:52–60, 2004
3. High Performance Fortran Forum (1993). High performance Fortran language specification, version 1.0. Technical report CRPC-TR92225, Houston
4. Nieplocha J, Palmer B, Tippuraju V, Krishnan M, Trease H, Apra E (2006). Advances, applications and performance of the global arrays shared memory programming toolkit. *Int J High Perform Comput Appl* 20:203–231

5. Numrich RW, Reid J (1998) Co-array fortran for parallel programming. SIGPLAN Fortran Forum 17(2):1-31
6. Open MP (2000) Simple, portable, scalable SMP programming. <http://www.openmp.org/>
7. Snir M, Otto S, Huss-Lederman S, Walker D, Dongarra J. MPI-the complete reference. The MPI Core, vol 1. MIT Press, Cambridge, MA
8. Split-C website. <http://www.eecs.berkeley.edu/Research/Projects/CS/parallel/castle/split-c/>
9. UPC language Specification, V1.2, May 2005
10. The X10 programming language. <http://x10.sourceforge.net>, 2004
11. Yelick K, Semenzato L, Pike G, Miyamoto C, Liblit B, Krishnamurthy A, Hilfinger P, Graham S, Gay D, Colella P, Aiken A (1998). Titanium: a high-performance Java dialect. Concurrency Pract Experience 10(11-13):825-836

discrete criteria that rely on counting changes in the sequence data date back to the late 1960s and early 1970s.

Computationally, likelihood-based phylogenetic inference approaches represent a major challenge, because of high memory footprints and of floating point intensive computations.

The goal of phylogenetic inference consists in reconstructing the evolutionary history of a set of  $n$  present-day organisms for which molecular sequence data can be obtained. In some cases it is also possible to extract ancient DNA or establish the morphological properties (traits) of fossil records.

## Input

The input for a phylogenetic analysis is a list of organism names and their associated DNA or protein sequence data. Note that the DNA sequences for distinct organisms will typically have different lengths. In modern phylogenetics, instead of using the raw sequence data, a so-called multiple sequence alignment (MSA) of the molecular data of the organisms is used as input. Multiple sequence alignment is an important – generally NP-hard – bioinformatics problem. The key goal of MSA is to infer homology, that is, determine which nucleotide characters in the sequence data share a common evolutionary history. Because insertions and/or deletions of nucleotides may have occurred during the evolutionary history of the organisms (represented by their DNA sequences), such events are denoted by inserting the gap symbol – into the sequences during the MSA process. After the alignment step, all  $n$  sequences will have the same length  $m$ , that is, the MSA has  $m$  alignment columns (also called: characters, sites, positions). A simple example for an MSA of DNA data for the Human, the Mouse, the Cow, and the Chicken with  $n = 4$  species and  $m = 27$  sites is provided below:

|         |                               |
|---------|-------------------------------|
| Cow     | ATGGCATATCCCA - ACAACTAGGATT  |
| Chicken | ATGGCCAACCACTCCCAACTAGGCTTA   |
| Human   | ATGGCACAT --- GCGCAAGTAGGTCTA |
| Mouse   | ATGG--- CCCATTCCAACTTGGTCTA   |

## Output

The output of a phylogenetic analysis is mostly an *unrooted* binary tree topology. The present-day organisms under study (for which DNA data can be extracted) are assigned to the leaves (tips) of such a tree, whereas the inner nodes represent common extinct ancestors. The branch lengths of the tree represent the relative

## Phylogenetic Inference

► Phylogenetics

## Phylogenetics

ALEXANDROS STAMATAKIS

Heidelberg Institute for Theoretical Studies, Heidelberg, Germany

### Synonyms

Molecular evolution; Phylogenetic inference; Reconstruction of evolutionary trees

### Definition

Phylogenetics, or phylogenetic inference (bioinformatics discipline), deals with models and algorithms for reconstruction of the evolutionary history – mostly in form of a (binary) evolutionary tree – for a set of living biological organisms based upon their molecular (DNA) or morphological (morphological traits) sequence data.

### Discussion

#### Introduction

The reconstruction of phylogenetic (evolutionary) trees from molecular or morphological sequence data is a comparatively old bioinformatics discipline, given that likelihood-based statistical models for phylogenetic inference were introduced in the early 1980s, while

evolutionary time between two nodes in the tree. A likelihood-based phylogenetic tree for the Human, the Mouse, the Cow, and the Chicken using the above MSA is provided in Fig. 1. The biological interpretation of this tree is that the Mouse and the Human are more closely related to each other than to the Cow and the Chicken.

### Combinatorial Optimization

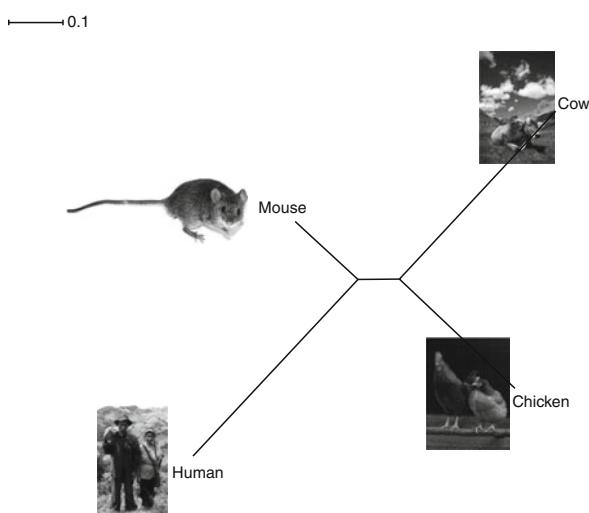
In order to reconstruct a phylogenetic tree, criteria are required to assess how well a specific tree topology explains (fits) the underlying molecular sequence data. One may think of this as an abstract function  $f()$  that scores alternative tree topologies for a given, fixed MSA. Thus, the goal of a phylogenetic tree reconstruction algorithm is to find the tree topology with the best score according to  $f()$ , that is, phylogenetic inference is a combinatorial optimization problem. The algorithmic problem in phylogenetics is characterized by the number of possible distinct unrooted binary tree topologies for  $n$  organisms that is given by:  $\prod_{i=3}^n (2i - 5)$ . For  $n = 50$ , there already exist approximately  $10^{80}$  possible tree topologies; this number corresponds to the number of atoms in the universe. Because of the size of the tree search space, phylogenetic inference under commonly used scoring criteria  $f()$  such as maximum likelihood or maximum parsimony is NP-hard. Therefore, a significant amount of research effort in phylogenetics has

focused on the design of efficient heuristic search strategies. Moreover, the optimization of the scoring function  $f()$  by algorithmic and technical means also represents an important research objective in phylogenetics.

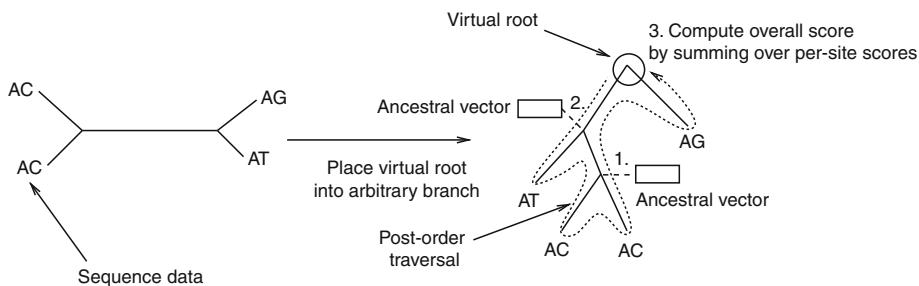
### Optimality Criteria

The most straightforward approach to phylogenetic inference is to use distance-based methods as opposed to character-based methods (see below). Distance-based methods rely on initially building a symmetric  $n \times n$  matrix  $D$  of pair-wise distances between the organisms under consideration, which is subsequently used to infer a tree. The optimality of a tree with given branch lengths is determined via a least squares method that is deployed to quantify the difference between the distances given by  $D$  and the distances induced by the tree topology (also called patristic distances). Thus, least squares optimization strives to find the tree that minimizes the difference between the pair-wise distances induced by the tree and the corresponding distances in  $D$  and is known to be NP-hard. Commonly used heuristics for distance-based tree reconstruction are the Unweighted Pair Group Method with Arithmetic mean (UPGMA) and Neighbor Joining (NJ) methods.

Character-based methods (parsimony and likelihood) directly operate on the sequences of the MSA. The sequences are assigned to the leaves of the tree and an overall score for the tree is computed via a post-order tree traversal with respect to a virtual root. One of the main properties of the likelihood and parsimony criteria is that the respective scores (function  $f()$ ) are invariant to the placement of such a virtual root, that is, the scores will be identical, irrespective of where the virtual root is placed. Parsimony and likelihood criteria are characterized by two additional properties. (1) They assume that MSA columns have evolved independently, that is, given a fixed tree topology, one can simultaneously compute a parsimony or likelihood score for each column of the MSA. To obtain the overall tree score, the sum over all  $m$ , where  $m$  is number of columns in the MSA, per-column likelihood or parsimony scores at the virtual root is computed. (2) Likelihood and parsimony scores are computed via a post-order tree traversal that proceeds from the tips toward the virtual root and computes ancestral sequence or ancestral probability vectors of length  $m$  at each inner node that is visited (see Fig. 2).



**Phylogenetics. Fig. 1** Likelihood-based tree for the Cow, the Chicken, the Mouse, and the Human



**Phylogenetics. Fig. 2** Virtual rooting and post-order traversal of a phylogenetic tree. During the post-order traversal, ancestral state vectors are computed. The per-column parsimony or likelihood scores are summed up at the root to obtain the overall tree score

The parsimony criterion intends to minimize the number of nucleotide changes on a tree, while maximum likelihood strives to maximize the fit between the tree and the data (the MSA) using an explicit statistical model of sequence evolution. Bayesian approaches that integrate over the tree (parameter) space using (Metropolis-Coupled) Markov-Chain Monte-Carlo approaches have become popular since the late 1990s. The underlying computational problems are similar, because, as for maximum likelihood, execution times are dominated (85–95%) by evaluations of the phylogenetic likelihood function.

Finally, there exist methods that do not directly use molecular sequence data for phylogeny reconstruction. Instead, these methods use gene order data as input, that is, they strive to infer evolutionary relationships based on distinct arrangements of corresponding genes along the chromosome(s) of the organisms under study. In this context, two organisms are more closely related to each other, if the order of their corresponding genes has not substantially changed, that is, if their chromosomes have not been rearranged to a large extent in the course of evolution. The overall goal can be formulated as inferring a tree that explains the evolutionary history in a parsimonious way, that is, with a minimum number of gene rearrangement events. Even simple versions of this problem are NP-hard [13].

## Vectorization

Both likelihood and parsimony computations can be vectorized at a low level.

Parsimony operations for counting the number of changes in a tree can be represented as operations on bit vectors, since ancestral parsimony vectors require 4

bits per alignment column to denote the presence or absence of one of the DNA characters A, C, G, T. The bit-level operations that are required are: bit-wise and, bit-wise or, bit-wise nand, and population count (popcount; counting the number of bits that are set to 1 in a data word) operations. Using Streaming Single Instruction Multiple Data (SIMD) Extensions 3 (SSE3) vector instructions on x86 architectures, which are 128 bits wide, 32 ancestral states (128 divided by 4) can be computed during a single CPU cycle.

Likelihood computations can also be vectorized, since the computation of the ancestral state at a position  $c$ , where  $c = 1 \dots m$ , of the alignment entails computing the probabilities  $P(A), P(C), P(G), P(T)$  of observing a nucleotide A, C, G, or T at this position. At an abstract level, the phylogenetic likelihood computations for DNA data are dominated by a dense matrix-matrix multiplication of a  $4 \times 4$  floating point matrix (nucleotide substitution matrix) with a  $4 \times m$  floating point matrix (ancestral probability vector).

The open-source phylogenetic inference program Randomized Axelerated Maximum Likelihood (RAxML) by Stamatakis [53] provides SSE3-vectorized implementations of the likelihood and parsimony functions for DNA data. Vector instructions for the likelihood function have also been deployed on the IBM CELL architecture by Blagojevic et al. [8]. In general, both optimality criteria allow for vectorization using wider (e.g., 512-bit) vector lengths. Auto-vectorization is not always possible because of code complexity or because the codes need to be redesigned to take advantage of vector instructions. In RAxML, for instance, intrinsic SSE3 functions have been used for explicit vectorization.

Distance-based methods can in principle also be vectorized, but the specific strategy depends on the function used to compute the pair-wise distances between sequences and also on the heuristic search strategy that is deployed.

### Fine-Grain Parallelization

As outlined in Fig. 2, the computations of per-site parsimony or likelihood scores are completely independent of each other until the virtual root is reached. Given an MSA with  $m = 1000$  sites, this means that all per-site scores can be computed simultaneously and in parallel. The only limitation is that, to obtain the overall score for the tree, the per-site scores need to be accumulated when the virtual root is reached via a respective parallel reduction operation.

The parallel efficiency of this approach depends on the speed of reduction operations in a parallel system and on the number of sites  $m$  in the MSA. Generally, scalability increases with  $m$  since a large  $m$  will yield a more favorable computation to synchronization (via a reduction operation) ratio. Both vectorization and fine-grain parallelization approaches for the likelihood and parsimony criteria are independent of the search strategy used.

To date, fine-grain parallelism has mainly been adopted by likelihood-based programs, since the likelihood function has significantly higher memory and computational requirements than parsimony.

In terms of parallel programming paradigms, Open Multi-Processing (OpenMP), POSIX threads (Pthreads), the Message Passing Interface (MPI), and the Compute Unified Device Architecture (CUDA) have been deployed for exploring fine-grain parallelism. The following types of parallel computer architectures or accelerator devices have been used for phylogenetic likelihood computations to date: Field Programmable Gate Arrays (FPGAs), Graphics Processing Units (GPUs), the IBM Playstation 3 and the IBM CELL processor, Symmetric Multi-Processors (SMPs), multi-core architectures, clusters of SMPs with InfiniBand and Gigabit-Ethernet interconnects, the massively parallel IBM BlueGene/L, and a shared-memory Silicon Graphics Instruments (SGI) Altix 4700 supercomputer.

### Medium-Grain Parallelization

Medium-grain parallelization refers to parallelizing the search algorithms of parsimony- or likelihood-based methods and is also called inference parallelism. However, because of their diversity and complexity, the parallelization of the steps of heuristic search algorithms is a nontrivial task. Moreover, every parallelization will need to be highly algorithm-specific and thereby not be generally applicable to other phylogenetic inference programs. An additional problem is that many modern search algorithms, as implemented for instances by Ronquist and Huelsenbeck in MrBayes [50], by Zwickl in GARLI [68], by Guindon and Gascuel in PHYML [28], by Stamatakis in RAxML [53], or by Goloboff in TNT [26], are characterized by hard-to-resolve sequential dependencies. However, other search algorithms like IQPNNI by Minh et al. [38] and TreePuzzle by Strimmer et al. [62] are straightforward to parallelize at this level.

### Coarse-Grain Parallelization

Likelihood-based and distance-based phylogenetic inference algorithms exhibit relatively easy-to-exploit sources of coarse-grain/embarrassing parallelism, which are discussed below.

#### Coarse-Grain Parallelism in Distance-Based Analyses

Parallelization of distance-based analyses is straightforward. The execution times of distance-based analyses are dominated by the computation of the  $n \times n$  symmetric distance matrix  $D$  that contains the pair-wise distances between all  $n$  organisms. All  $n^2$  entries of  $D$  are independent of each other and can hence be computed in parallel.

#### Coarse-Grain Parallelism in Maximum Likelihood Analyses

There are two sources of coarse-grain parallelism in ML analyses. One can conduct a number of completely independent tree searches, starting from different reasonable, that is, nonrandom, starting trees. Such reasonable starting trees can, for instance, be obtained by using simpler (and less computationally intensive)

methods such as neighbor joining or maximum parsimony. Given a collection of distinct, nonrandom, starting trees, individual ML tree searches can be conducted to find the best-scoring (remember that ML optimization is NP-hard) tree. Thereby, one may find a tree with a better likelihood score than by conducting a single search. The same technique can also be applied to parsimony searches for finding the most parsimonious tree.

The second source of embarrassing parallelism in ML phylogenetic analyses is the phylogenetic bootstrapping procedure that was proposed by Felsenstein [20]. Phylogenetic bootstrapping serves as a mechanism for inferring support values, that is, for assigning confidence values to inner branches of a phylogenetic tree. Those confidence values at the inner branches are usually interpreted as the certainty that a particular evolutionary split of the set of organisms has been correctly inferred. Cutting the tree into two parts at an inner branch generates a split (also called bipartition) of the organisms into two disjoint sets. Therefore, the goal consists in obtaining support values for all possible splits/bipartitions induced by the internal branches of a tree.

The phylogenetic bootstrap procedure works as follows: Initially, the input alignment is perturbed by drawing columns/sites (with replacement) at random to assemble a bootstrap replicate alignment of length  $m$ . Thus, a bootstrapped alignment has the same number  $m$  of sites/columns as the original alignment, but exhibits a different site composition. This re-sampling process is repeated 100–1,000 times, that is, 100–1,000 bootstrap replicate alignments are generated. When those 100–1,000 bootstrapped alignments have been generated, one applies the phylogenetic inference method of choice to infer a tree for each of the replicates. Thereby, as many trees as there are bootstrap replicates are generated. This set of bootstrap trees is then used to answer the question: How stable is the tree topology under slight alterations of the input data?

This collection of bootstrapped trees is then either used to compute a consensus tree, that is, compute the “average” tree topology or for drawing support values on the best-known ML tree obtained on the original – non-bootstrapped – alignment. In the latter case, one just needs to count how frequently each bipartition of the

best-known ML tree occurs in the set of bootstrapped trees.

The bootstrapping procedure is embarrassingly parallel because tree searches on individual bootstrap replicates are completely independent from each other and can be parallelized by executing the 100–1,000 bootstrap inferences on a cluster, GRID, or cloud.

A question that naturally arises in this context is: How many bootstrap replicates are required to obtain reliable support values? Hedges [29] proposed a theoretical upper bound for phylogenetic bootstrapping. The number of required bootstrap replicates also appears to depend on the input data. Therefore, adaptive criteria as proposed by Patterson et al. [45] may be well-suited to determine a sufficient number of bootstrap replicates.

### Coarse-Grain Parallelism in Bayesian Analyses

Bayesian analyses exhibit a source of coarse grain parallelism at the level of executing multiple chains in parallel, that is, using the Metropolis-Coupled Markov-Chain Monte-Carlo (MC<sup>3</sup>) approach (see Metropolis et al. [37]). In a typical program run of the widely used MrBayes code by Ronquist and Huelsenbeck [50] for Bayesian phylogenetic inference, the program will use three heated Markov chains that accept more radical moves in parameter space (this entails topological moves as well as moves to sample other parameters of the likelihood model) and a cold chain that only accepts more conservative moves. The cold chain operates on the tree with the currently best likelihood. However, one of the heated chains may at some point encounter a tree with a higher likelihood than the cold chain. In this case, the cold chain and the heated chain with the better tree need to exchange states, that is, the cold chain will become a heated chain and the heated chain will become the cold chain. Therefore, while those four chains (three heated chains and one cold chain) may be started as independent Message Passing Interface (MPI) processes, the chains will need to be synchronized after a certain number of, for instance 100, generations (proposals) to assess if states need to be exchanged between chains. Thus, parallel load balance is a critical issue, because the chains will need to run at similar speeds in order to minimize synchronization delays. On a homogeneous cluster (equipped with a single CPU type) this

is not problematic, because the average execution times for 100 or 1,000 proposals are expected to be very similar. This good expected load balance is due to the fact that the same average number of proposal types (tree proposal, model parameter proposal) will be executed in each chain. Finally, completely independent runs can be executed in an embarrassingly parallel manner.

## Phylogenetics Today

Phylogenetics currently face two main challenges. One major challenge is the significant advances in wet-lab sequencing technologies that have led to an unprecedented molecular data “flood.” In fact, the amount of publicly available molecular data increases at a significantly higher speed than the number of transistors according to Moore’s law (see Goldman and Yang [25] for a respective plot). To this end, researchers today do not use sequence data from a single gene or just a couple of genes to reconstruct trees for the organisms they study. Instead, they use data from several hundreds or even thousands of genes (e.g., Hejnol et al. [30]). Thus, the number of sites  $m$  in alignments has increased from 1,000–10,000 to over 1,000,000. This transition from single-/few-gene phylogenies to many-gene phylogenies is also reflected by the increased use of the term phylogenomics, that is, phylogenetic inference at (almost) the whole-genome level.

In current phylogenomic analyses the number of organisms  $n$  ranges between 50 and 500, but given the constant innovations in wet-lab sequencing, this number may soon increase by one order of magnitude. Because of sequencing innovations, input datasets are also growing with respect to  $n$ , that is, analyses using up to ten genes for 10,000–70,000 organisms are becoming more common. To date, the largest published likelihood-based tree contained 13,000 organisms (see Smith and Donoghue [52]) while the largest published parsimony-based tree contained 73,000 organisms (see Goloboff et al. [27]). The general growth in dataset sizes poses problems with respect to the memory requirements of phylogenetic analyses, in particular with respect to likelihood-based approaches. Memory footprints for computing the likelihood on a single tree, exceeding 50GB are not uncommon any more. The highest reported memory footprint the author is aware of was 180GB for a likelihood-based analysis

of 35 mammalian genomes (Ziheng Yang, personal communication, April 2010). Memory-related problems also affect distance-based methods (for large  $n$ ) because of the space requirements of the  $n \times n$  distance matrix (Wheeler [65] and Price et al. [48] discuss some technical solutions to this problem). Apart from memory-related problems, the computation of trees with more than 10,000 organisms also poses novel algorithmic challenges. Finally, the increasing complexity of the statistical models that are used for phylogenetic inference, such as, mixture models (e.g., Lartillot and Philippe [31]), further increase the computational complexity of likelihood-based approaches.

The second major challenge is the emergence of multi-core systems at the desktop level and of accelerator architectures such as GPUs (Graphics Processing Units) or the IBM Cell. Those new parallel architectures pose challenges regarding the parallelization of the likelihood or parsimony functions. Given the aforementioned high memory footprints, a parallelization at a fine-grain level, that is, multiple processors/cores working together to compute the score on a single tree, is required. In order to be of value and to be used by the biological user community, such parallelizations need to be readily available at the production level and also need to be easy to install and use. Developers of phylogenetics software, which has come off age by now, are also increasingly facing software engineering issues, because the codes have become more complex over recent years. Programs such as RAxML or MrBayes provide a plethora of substitution models and search algorithms. They also allow for analyzing different data types, for instance, binary, morphological, DNA, RNA secondary structure (see Savill et al. [51] for a discussion of RNA secondary structure evolution models), or protein data. Therefore, the phyloinformatics and HPC communities are facing an unprecedented challenge in trying to keep pace with data accumulation and parallel architecture innovations to provide ever more scalable and powerful analysis tools. HPC in phylogenetics has thus become a key to the success of the field.

## Future Directions

One of the major future challenges in phylogenetics consist of efficiently exploiting parallel architectures to compute the likelihood or parsimony scoring

functions at the production level. There exists an ongoing effort to implement an open-source likelihood function library (<http://code.google.com/p/beagle-lib/>) that can be executed on GPUs and multi-core architectures (including a vectorization using SSE3 intrinsics and an OpenMP-based fine-grain parallelization). Fine-grain parallelizations of those functions are required to be able to accommodate and distribute the growing memory space requirements across several cores or – potentially hybrid – multi-core nodes. Nonetheless, alternative directions for handling memory consumption should be explored. While there exist suggestions for reducing memory requirements via algorithmic means (see Stamatakis and Ott [57] and Stamatakis and Alachiotis [54]), trade-offs between using single and double precision arithmetics as well as their impact on numerical stability need to be further explored. Out-of-core execution may provide a solution for executing large-scale analyses on the desktop. Analyses of trees with over 10,000 organisms require a – potentially difficult – parallelization at the algorithmic level, because scalability of fine-grain parallelism is limited to 8 or 16 cores due to the relatively small number of sites in such alignments. Finally, the simultaneous development of parallelization and algorithmic strategies appears to be the most promising approach to address current and future challenges.

## Related Entries

- ▶ [Bioinformatics](#)
- ▶ [Cell Broadband Engine Processor](#)
- ▶ [Clusters](#)
- ▶ [Collective Communication](#)
- ▶ [Distributed-Memory Multiprocessor](#)
- ▶ [Ethernet](#)
- ▶ [Genome Assembly](#)
- ▶ [Hybrid Programming With SIMPLE](#)
- ▶ [IBM Blue Gene Supercomputer](#)
- ▶ [InfiniBand](#)
- ▶ [Load Balancing, Distributed Memory](#)
- ▶ [Loops, Parallel](#)
- ▶ [MPI \(Message Passing Interface\)](#)
- ▶ [Multi-Threaded Processors](#)
- ▶ [NVIDIA GPU](#)
- ▶ [OpenMP](#)

- ▶ [Parallelization, Automatic](#)
- ▶ [POSIX Threads \(Pthreads\)](#)
- ▶ [Reconfigurable Computers](#)

## Bibliographic Notes and Further Reading

The text-books by Felsenstein [21] and Yang [66] provide a detailed introduction to the field of phylogenetic inference and the underlying models of evolution. The phylogenetic likelihood function was introduced by Felsenstein in 1981 [19]. One of the early papers dealing with parsimony was published by Fitch and Margoliash [23]. NP-hardness was demonstrated by Day [17] for the least squares approach, by Day et al. for parsimony [18], and by Chor and Tuller for likelihood [16] (also see Roch [49] for a shorter proof). Morrison [42] provides an overview and discussion of current heuristic search strategies. The vectorization of the likelihood function as well as numerical aspects with respect to single and double precision implementations of the likelihood function are addressed by Stamatakis and Berger [7]. FPGA implementations are covered by Alachiotis et al. [3, 4], Mak and Lam [34–36], Zierke and Bakos [67], and Bakos [5, 6]; GPU implementations by Charalambous et al. [15], Suchard and Rambaut [63], while Pratas et al. describe a performance comparison between GPUs, the IBM CELL, and general purpose multi-core architectures [47]. Blagojevic et al. have published a series of papers on porting RAxML to the IBM CELL [8–10, 55]. Pthreads and OpenMP-based parallelization of the parsimony and likelihood functions have been described by Stamatakis et al. [57, 60]. Ott et al. [43, 44] and Feng et al. [22] describe fine-grain parallelizations with MPI on distributed memory machines. Stamatakis and Ott also address load-balance issues [59] and assess performance of Pthreads versus MPI versus OpenMP for fine-grained parallelism in the likelihood function [58]. Medium-grain parallelizations are covered by Minh et al. [38], Stamatakis et al. [56], Stewart et al. [61], and Ceron et al. [14]. Hybrid parallelizations have been described by Minh et al. [39], Feng et al. [22], and Pfeiffer and Stamatakis [46]. The two post-analysis steps for analyzing bootstrapped trees (consensus tree building and drawing bipartitions on the best-known tree) have also been parallelized (see Aberer et al. [1, 2]; the papers also

contain a detailed description of the discrete algorithms for consensus tree building and drawing bipartitions on trees). A related discrete problem on trees, that of reconstructing a species tree from (potentially incongruent) per-gene trees by the minimum possible number of gene duplication events, has been parallelized by Wehe et al. [64] on an IBM BlueGene/L supercomputer. The review by Maddison [33] provides an overview of the gene tree species tree problem. Heuristic algorithms for gene order phylogeny reconstruction have been implemented and optimized by Moret et al. [40, 41] in a tool called GRAPPA. GRAPPA has also been parallelized using a coarse-grain approach to simultaneously enumerate and evaluate all possible trees for 13 organisms on a cluster with 512 cores. The original heuristic algorithm has been proposed by Blanchette et al. [11]. Finally, the papers by Fleissner et al. [24], Loytynoja and Goldman [32], or Bradley et al. [12], for instance, deal with the more challenging and advanced problem of simultaneous alignment (MSA) and tree building.

## Bibliography

1. Aberer A, Pattengale N, Stamatakis A (2010) Parallel computation of phylogenetic consensus trees. *Procedia Comput Sci* 1(1): 1059–1067
2. Aberer A, Pattengale N, Stamatakis A (2010) Parallelized phylogenetic post-analysis on multi-core architectures. *J Comput Sci* 1(2):107–114
3. Alachiotis N, Sotiriades E, Dollas A, Stamatakis A (2009) Exploring FPGAs for accelerating the phylogenetic likelihood function. In: IEEE international symposium on parallel & distributed processing, 2009. IPDPS 2009, pp 1–8. IEEE
4. Alachiotis N, Stamatakis A, Sotiriades E, Dollas A (2009) A reconfigurable architecture for the Phylogenetic Likelihood Function. In: International Conference on Field Programmable Logic and Applications, 2009. FPL 2009, pp 674–678. IEEE, 2009
5. Bakos J (2007) FPGA acceleration of gene rearrangement analysis. In: Proceedings of 15th annual IEEE symposium on field-programmable custom computing machines. IEEE, Napa, CA, pp 85–94
6. Bakos J, Elenis P, Tang J (2007) FPGA acceleration of phylogeny reconstruction for whole genome data. In: Proceedings of the 7th IEEE international conference on bioinformatics and bio engineering. IEEE, Boston, MA, pp 888–895
7. Berger S, Stamatakis A (2010) Accuracy and performance of single versus double precision arithmetics for maximum likelihood phylogeny reconstruction. *Lecture notes in computer science*, vol 6068. Springer, pp 270–279
8. Blagojevic F, Nikolopoulos D, Stamatakis A, Antonopoulos C (2007) Dynamic multigrain parallelization on the cell broadband engine. In: Proceedings of PPoPP 2007, San Jose, CA, March 2007, pp 90–100
9. Blagojevic F, Nikolopoulos D, Stamatakis A, Antonopoulos C, Curtis-Maury M (2007) Runtime scheduling of dynamic parallelism on accelerator-based multi-core systems. *Parallel Comput* 33:700–719
10. Blagojevic F, Nikolopoulos DS, Stamatakis A, Antonopoulos CD (2007) RAxML-Cell: Parallel phylogenetic tree inference on the cell broadband engine. In: Proceedings of international parallel and distributed processing symposium (IPDPS2007), 2007
11. Blanchette M, Bourque G, Sankoff D (1997) Breakpoint phylogenies. In: Miyano S, Takagi T (eds) *Workshop on genome informatics*, vol 8. Univ. Academy Press, pp 25–34
12. Bradley R, Roberts A, Smoot M, Juvekar S, Do J, Dewey C, Holmes I, Pachter L (2009) Fast statistical alignment. *PLoS Comput Biol* 5(5):e1000392
13. Bryant D (1998) The complexity of the breakpoint median problem. Technical report, University of Montreal, Canada
14. Ceron C, Dopazo J, Zapata E, Carazo J, Trelles O (1998) Parallel implementation of DNAML program on message-passing architectures. *Parallel Comput* 24(5–6):701–716
15. Charalambous M, Trancoso P, Stamatakis A (2005) Initial experiences porting a bioinformatics application to a graphics processor. *Lecture notes in computer science*, vol 3746. Springer, New York, pp 415–425
16. Chor B, Tuller T (2005) Maximum likelihood of evolutionary trees: hardness and approximation. *Bioinformatics* 21(1):97–106
17. Day W (1987) Computational complexity of inferring phylogenies from dissimilarity matrices. *Bulletin of Mathematical Biology* 49(4):461–467
18. Day W, Johnson D, Sankoff D (1986) The computational complexity of inferring rooted phylogenies by parsimony. *Mathematical biosciences* 81(33–42):299
19. Felsenstein J (1981) Evolutionary trees from DNA sequences: a maximum likelihood approach. *J Mol Evol* 17:368–376
20. Felsenstein J (1985) Confidence limits on phylogenies: an approach using the bootstrap. *Evolution* 39(4):783–791
21. Felsenstein J (2004) Inferring phylogenies. Sinauer Associates, Sunderland
22. Feng X, Cameron K, Sosa C, Smith B (2007) Building the tree of life on terascale systems. In: Proceedings of international parallel and distributed processing symposium (IPDPS2007), 2007
23. Fitch W, Margoliash E (1967) Construction of phylogenetic trees. *Science* 155(3760):279–284
24. Fleissner R, Metzler D, Haeseler A (2005) Simultaneous statistical multiple alignment and phylogeny reconstruction. *Syst Biol* 54:548–561
25. Goldman N, Yang Z (2008) Introduction. statistical and computational challenges in molecular phylogenetics and evolution. *Philos Trans R Soc B Biol Sci* 363(1512):3889
26. Goloboff P (1999) Analyzing large data sets in reasonable times: solution for composite optima. *Cladistics* 15:415–428
27. Goloboff PA, Catalano SA, Mirande JM, Szumik CA, Arias JS, Källersjö M, Farris JS (2009) Phylogenetic analysis of 73060 taxa corroborates major eukaryotic groups. *Cladistics* 25:1–20
28. Guindon S, Gascuel O (2003) A simple, fast, and accurate algorithm to estimate large phylogenies by maximum likelihood. *Syst Biol* 52(5):696–704

29. Hedges S (1992) The number of replications needed for accurate estimation of the bootstrap P value in phylogenetic studies. *Mol Biol Evolution* 9(2):366–369
30. Hejnol A, Obst M, Stamatakis A, Ott M, Rouse G, Edgecombe G, Martinez P, Baguna J, Bailly X, Jondelius U, Wiens M, Müller W, Seaver E, Wheeler W, Martindale M, Giribet G, Dunn C (2009) Rooting the bilaterian tree with scalable phylogenomic and supercomputing tools. *Proc R Soc B* 276:4261–4270
31. Lartillot N, Philippe H (2004) A Bayesian mixture model for across-site heterogeneities in the amino-acid replacement process. *Mol Biol Evol* 21(6):1095–1109
32. Loytynoja A, Goldman N (2008) Phylogeny-aware gap placement prevents errors in sequence alignment and evolutionary analysis. *Science* 320(5883):1632
33. Maddison W (1997) Gene trees in species trees. *Syst Biol* 46(3):523
34. Mak T, Lam K (2003) High speed GAML-based phylogenetic tree reconstruction using HW/SW codesign. In: Bioinformatics Conference, 2003. CSB 2003. Proceedings of the 2003 IEEE, pp 470–473
35. Mak T, Lam K (2004) Embedded computation of maximum-likelihood phylogeny inference using platform FPGA. In: Proceedings of IEEE Computational Systems Bioinformatics Conference (CSB 04), pp 512–514
36. Mak T, Lam K (2004) FPGA-Based Computation for Maximum Likelihood Phylogenetic Tree Evaluation. In: Lecture notes in computer science, pp 1076–1079
37. Metropolis N, Rosenbluth A, Rosenbluth M, Teller A, Teller E et al (1953) Equation of state calculations by fast computing machines. *J Chem Phys* 21(6):1087
38. Minh B, Vinh L, Haeseler A, Schmidt H (2005) pIQPNNI: parallel reconstruction of large maximum likelihood phylogenies. *Bioinformatics* 21(19):3794–3796
39. Minh B, Vinh L, Schmidt H, Haeseler A (2006) Large maximum likelihood trees. In: Proceedings of the NIC Symposium 2006, pp 357–365
40. Moret B, Tang J, Wang L, Warnow T (2002) Steps toward accurate reconstructions of phylogenies from gene-order data\*. 1. *J Comput Syst Sci* 65(3):508–525
41. Moret B, Wyman S, Bader D, Warnow T, Yan M (2001) A new implementation and detailed study of breakpoint analysis. In: Pacific symposium on biocomputing 6:583–594
42. Morrison D (2007) Increasing the efficiency of searches for the maximum likelihood tree in a phylogenetic analysis of up to 150 nucleotide sequences. *Syst Biol* 56(6):988–1010
43. Ott M, Zola J, Aluru S, Johnson A, Janies D, Stamatakis A (2008) Large-scale phylogenetic analysis on current HPC architectures. *Scientific Programming* 16(2–3):255–270
44. Ott M, Zola J, Aluru S, Stamatakis A (2007) Large-scale maximum likelihood-based phylogenetic analysis on the IBM BlueGene/L. In: Proceedings of IEEE/ACM Supercomputing Conference 2007 (SC2007), IEEE, Reno, Nevada
45. Patterson N, Alipour M, Bininda-Emonds O, Moret B, Stamatakis A (2010) How many bootstrap replicates are necessary? *J Comput Biol* 17(3):337–354
46. Pfeiffer W, Stamatakis A (2010) Hybrid MPI/Pthreads parallelization of the RAxML phylogenetics code. In: IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010, IEEE, Atlanta, Georgia, pp 1–8
47. Pratas F, Trancoso P, Stamatakis A, Sousa L (2009) Fine-grain Parallelism using multi-core, Cell/BE, and GPU systems: accelerating the phylogenetic likelihood function. In: International conference on parallel processing, 2009. ICPP'09, IEEE, Vienna, pp 9–17
48. Price M, Dehal P, Arkin A (2010) FastTree 2—approximately maximum likelihood trees for large alignments. *PLoS One* 5(3):e9490
49. Roch S (2006) A short proof that phylogenetic tree reconstruction by maximum likelihood is hard. *IEEE/ACM transactions on Computational Biology and Bioinformatics*, pp 92–94
50. Ronquist F, Huelsenbeck J (2003) MrBayes 3: Bayesian phylogenetic inference under mixed models. *Bioinformatics* 19(12):1572–1574
51. Savill NJ, Hoyle DC, Higgs PG (2001) RNA sequence evolution with secondary structure constraints: comparison of substitution rate models using maximum-likelihood methods. *Genetics* 157:399–411
52. Smith S, Donoghue M (2008) Rates of molecular evolution are linked to life history in flowering plants. *Science* 322(5898):86–89
53. Stamatakis A (2006) RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. *Bioinformatics* 22(21):2688–2690
54. Stamatakis A, Alachiotis N (2010) Time and memory efficient likelihoodbased tree searches on phylogenomic alignments with missing data. *Bioinformatics* 26(12):i132
55. Stamatakis A, Blagojevic F, Antonopoulos CD, Nikolopoulos DS (2007) Exploring new search algorithms and hardware for phylogenetics: RAxML meets the IBM Cell. *J VLSI Sig Proc Syst* 48(3):271–286
56. Stamatakis A, Ludwig T, Meier H (2004) Parallel inference of a 10,000-taxon phylogeny with maximum likelihood. In: Proceedings of Euro-Par 2004, September 2004, IEEE, Pisa Italy, pp 997–1004
57. Stamatakis A, Ott M (2008) Efficient computation of the phylogenetic likelihood function on multi-gene alignments and multi-core architectures. *Philos Trans R Soc B, Biol Sci* 363:3977–3984
58. Stamatakis A, Ott M (2008) Exploiting fine-grained parallelism in the phylogenetic likelihood function with MPI, Pthreads, and OpenMP: a performance study. In: Chetty M, Ngom A, Ahmad S (eds) PRIB, Lecture notes in computer science, vol 5265. Springer, Heidelberg, pp 424–435
59. Stamatakis A, Ott M (2009) Load balance in the phylogenetic likelihood kernel. In: International conference on parallel processing, 2009. ICPP'09, IEEE, Vienna, Austria, pp 348–355
60. Stamatakis A, Ott M, Ludwig T (2005) RAxML-OMP: an efficient program for phylogenetic inference on SMPs. Lecture notes in computer science, vol 3606. Springer, Berlin, Heidelberg, pp 288–302
61. Stewart C, Hart D, Berry D, Olsen G, Wernert E, Fischer W (2001) Parallel implementation and performance of fastDNAml – a program for maximum likelihood phylogenetic inference. In: Supercomputing, ACM/IEEE 2001 conference, ACM/IEEE, Denver, Colorado, pp 32–32

62. Strimmer K, Haeseler A (1996) Quartet puzzling: a quartet maximum likelihood method for reconstructing tree topologies. *Mol Biol Evol* 13:964–969
63. Suchard M, Rambaut A (2009) Many-core algorithms for statistical phylogenetics. *Bioinformatics* 25(11):1370
64. Wehe A, Chang W, Eulenstein O, Aluru S (2010) A scalable parallelization of the gene duplication problem. *J Parallel Distr Comput* 70(3):237–244
65. Wheeler T (2009) Large-scale neighbor-joining with ninja. Lecture notes in computer science, vol 5724. Springer, Berlin, pp 375–389
66. Yang Z (2006) Computational molecular evolution. Oxford University Press, USA
67. Zierke S, Bakos J (2010) FPGA acceleration of the phylogenetic likelihood function for Bayesian MCMC inference methods. *BMC Bioinformatics* 11(1):184
68. Zwickl D (2006) Genetic algorithm approaches for the phylogenetic analysis of large biological sequence datasets under the maximum likelihood criterion. PhD thesis, University of Texas at Austin, April 2006

## Pi-Calculus

DAVIDE SANGIORGI  
Universita' di Bologna, Bologna, Italy

### Synonyms

Calculus of mobile processes

### Definition

The  $\pi$ -calculus is a process calculus that models mobile systems, i.e., systems with a dynamically changing communication topology. It refines the constructs of the calculus of communicating systems (CCS) by allowing the exchange of communication links. Ideas from the  $\lambda$ -calculus have also been influential.

### Discussion

#### Introduction

A widely recognized practice for understanding programming languages, be they sequential or concurrent, is to distill small “core languages,” or “calculi,” that embody the essential ingredients of the languages. This is useful to develop the theory of the programming language (e.g., techniques for static analysis, behavioral specification, and verification), to study implementations, to devise new or better programming language constructs.

A well-known calculus in the realm of sequential languages is the  $\lambda$ -calculus. Invented by Church in the 1930s, it is a pure calculus of functions. Everything in the  $\lambda$ -calculus is a function, and computation is function application. The  $\lambda$ -calculus has been very influential in programming languages. It has been the basis for the so-called functional programming languages. Even more important, it has been essential in understanding concepts such as procedural abstraction and binding, parameter passing (e.g., call-by-name vs. call-by-value), and continuations. All these concepts have then made their way into mainstream programming languages. The  $\lambda$ -calculus can be regarded as a canonical model for sequential computations with functions: on the one hand, it describes the essential features of functions (function definition and function application) in a simple and intuitive way; on the other hand, all the known models of sequential computation have been proved to have the same expressive power as the  $\lambda$ -calculus. In other words, the  $\lambda$ -calculus describes all computable functions (a statement referred to as “Church’s thesis”).

For concurrent languages (possibly including distribution), a similar canonical model does not exist. This can be explained with the varieties of such systems. In concurrency, the central computational unit is a process. However, there is no universally accepted definition of what a process is. For instance, the ways of conceiving interaction among processes may be very diverse: via synchronous signals, via shared variables, via broadcasting, via asynchronous message passing, via rendezvous. Therefore, in concurrency one does not find a single calculus, but, rather, a variety of calculi, referred to as *process calculi*, or *process algebras* when one wishes to emphasize that both the analysis and the description of a system are carried out in an algebraic setting. The best known such calculi are calculus of communicating systems (CCS), communicating sequential processes (CSP), algebra of communicating processes (ACP), the  $\pi$ -calculus. Among these, the  $\pi$ -calculus is the closest to the  $\lambda$ -calculus. Many fundamental ideas from the  $\lambda$ -calculus can be carried over to the  $\pi$ -calculus. For instance, abstraction and type systems are central concepts in both calculi. Moreover as the  $\lambda$ -calculus underlies functional programming languages, so the  $\pi$ -calculus, or variants of it, are taken as the basis for new programming languages. The process concept of

the  $\pi$ -calculus is roughly that of structured entities interacting by means of message-passing.

The  $\pi$ -calculus has two aspects. First, it is a theory of mobile systems, with a rich blend of techniques for reasoning about the behavior of systems. Second, it is a general model of computation, in which everything is a process and computation is process interaction. Syntactically, the main difference between the  $\pi$ -calculus and its principal process calculus ancestor, CCS, is that communication links are first-class values, and may therefore be passed around in communications. Thus, the set of communication links used by a term may change dynamically, as computation evolves. Such systems are called *mobile*, and the  $\pi$ -calculus is hence called a *calculus of mobile processes*. In CCS, in contrast, the set of communication links for a term is fixed by its initial syntax, and only very limited forms of mobility can be described. Mobility gives the  $\pi$ -calculus an amazing expressive power. For instance, functions and the  $\lambda$ -calculus can be elegantly modeled in it, reducing function application to special forms of process interactions.

## Mobility

A *mobile* concurrent system has a communication topology that can dynamically change, as the system evolves. Mobility can be found in many areas of computer science, such as operating systems, distributed computing, higher-order concurrent programming, and object-oriented programming. Two kinds of mobility can be broadly distinguished. In one kind, *links* move in an abstract space of *linked processes*. For example, hypertext links can be created, can be passed around, and can disappear; the connections between cellular telephones and a network of base stations can change as the telephones are carried around; and references can be passed as arguments of method invocations in object-oriented systems. In the second kind of mobility, *processes* move in an abstract space of linked processes. For instance, code can be sent over a network and put to work at its destination; mobile devices can acquire new functionality; an active laptop computer can be moved from one location to another, and made to interact with resources in a different environment. Languages in which terms of the language itself, such as processes, are first-class values, are called *higher-order* (in this sense, the  $\lambda$ -calculus is higher-order too).

The  $\pi$ -calculus models the first kind of mobility: it directly expresses movement of links in a space of linked processes. There are two kinds of basic entity in the (untyped)  $\pi$ -calculus: names and processes. Names specify links. Processes can interact by using names that they share. The crux is the data that processes communicate in interactions are themselves names, and a name received in one interaction can be used to participate in another. By receiving a name, a process can acquire a capability to interact with processes that were previously unknown to it. Thus, the structure of a system – the connections among its component processes – can change over time, in arbitrary ways. The source of strength in the  $\pi$ -calculus is how it treats scoping of names and extrusion of names from their scopes.

There are various reasons for having only mobility of names in the  $\pi$ -calculus. First, naming and name-passing are ubiquitous: think of addresses, identifiers, links, pointers, and references. Second, as will be shown later, higher-order constructs can often be modeled within the  $\pi$ -calculus. Further, by passing a name, one can pass partial access to a process, an ability to interact with it only in a certain way. Similarly, with name-passing one can easily model sharing, for instance of a resource that can be used by different sets of clients at different times. It can be complicated to model these things when processes are the only transmissible values. Thirdly, the  $\pi$ -calculus has a rich and tractable theory. The theory of process-passing is harder, and important parts of it are not yet well understood.

## Syntax

As the  $\lambda$ -calculus, so the language of the  $\pi$ -calculus consists of a small set of primitive constructs. In the  $\lambda$ -calculus, they are constructs for building functions. In the  $\pi$ -calculus, they are constructs for building processes, similar to those one finds in CCS. The syntax is as follows. Capital letters range over processes, and small letters over names; the set of all names is infinite.

$$\begin{aligned} P ::= & \quad x(y).P \mid \bar{x}(y).P \mid \tau.P \mid \mathbf{0} \mid P + P' \mid P \mid P' \\ & \mid \nu x.P \mid !P \mid [x=y]P \end{aligned}$$

The *input prefix*  $x(z).P$  can receive any name via  $x$  and continue as  $P$  with the received name substituted for  $z$ . The substitution of the all occurrences of  $z$  in

$P$  with  $y$  is written  $P\{y/z\}$  (some care is needed here, to properly respect the bindings of a term). An output  $\bar{x}\langle y \rangle.P$  emits name  $y$  at  $x$  and then continues as  $P$ . The *unobservable prefix*  $\tau.P$  can autonomously evolve to  $P$ , without the help of the external environment. As in CCS,  $\tau$  can be thought of as expressing an internal action of a process. The *inactive* process  $\mathbf{0}$  can do nothing. For instance,  $x(z).\bar{z}\langle y \rangle.\mathbf{0}$  can receive any name via  $x$ , send  $y$  via the name received, and become inactive. A *choice* (or *sum*) process  $P + P'$  may evolve either as  $P$  or as  $P'$ ; if one of the processes exercises one of its capabilities, the other process is discarded. In the *composition*  $P \mid P'$ , the components  $P$  and  $P'$  can proceed independently and can interact via shared names. For instance,  $(x(z).\bar{z}\langle y \rangle.\mathbf{0} + \bar{w}\langle v \rangle.\mathbf{0}) \mid \bar{x}\langle u \rangle.\mathbf{0}$  has four capabilities: to receive a name via  $x$ , to send  $v$  via  $w$ , to send  $u$  via  $x$ , and to evolve invisibly as an effect of an interaction between its components via the shared name  $x$ . In the *restriction*  $\nu z.P$ , the scope of the name  $z$  is restricted to  $P$ . Components of  $P$  can use  $z$  to interact with one another but not with other processes. For instance,  $\nu x((x(z).\bar{z}\langle y \rangle.\mathbf{0} + \bar{w}\langle v \rangle.\mathbf{0}) \mid \bar{x}\langle u \rangle.\mathbf{0})$  has only two capabilities: to send  $v$  via  $w$ , and to evolve invisibly as an effect of an interaction between its components via  $x$ . The scope of a restriction may change as a result of interaction between processes. This important feature of the calculus will be explained later. (The  $\pi$ -calculus notation for the restriction operator is different from that of CCS; indeed, exchange of names makes restriction in the  $\pi$ -calculus semantically quite different from restriction in CCS.) The *replication*  $!P$  can be thought of as an infinite composition  $P \mid P \mid \dots$  or, equivalently, a process satisfying the equation  $!P = P \mid !P$ . Replication is the operator that makes it possible to express infinite behaviors. For example,  $!x(z).\bar{y}\langle z \rangle.\mathbf{0}$  can receive names via  $x$  repeatedly, and can repeatedly send via  $y$  any name it does receive. In certain presentations, *recursive process definitions* are used in place of replication: the expressiveness injected into the calculus by these two constructs is the same. A *match* process  $[x=y]P$  behaves as  $P$  if names  $x$  and  $y$  are equal, otherwise it does nothing. For instance,  $x(z).[z=y]\bar{z}\langle w \rangle.\mathbf{0}$ , on receiving a name via  $x$ , can send  $w$  via that name just if that name is  $y$ ; if it is not, the process can do nothing further.

This is the syntax of the pure, untyped,  $\pi$ -calculus. The calculus is monadic, in that only one value at a

time can be exchanged. In examples and applications, one often uses the *polyadic  $\pi$ -calculus*, in which the input and output constructs are refined as follows: a polyadic input process  $x(y_1, \dots, y_n).P$  waits for an  $n$ -tuple of names  $z_1, \dots, z_n$  at  $x$  and then continues as  $P\{z_1, \dots, z_n/y_1, \dots, y_n\}$  (i.e.,  $P$  with the  $y_i$ 's replaced by the  $z_i$ 's); a polyadic output process  $\bar{x}(y_1, \dots, y_n).P$  emits names  $y_1, \dots, y_n$  at  $x$  and then continues as  $P$ .

The input construct  $x(y).P$  and the restriction  $\nu y P$  are binders for the free occurrences of  $y$  in  $P$ , in the same sense as the abstraction construct of the  $\lambda$ -calculus. These binders give rise in the expected way to the definitions of *free* and *bound* names of a process.

When writing processes, parentheses are used to resolve ambiguity, and the conventions are observed that prefixing and restriction bind more tightly than composition and sum. Thus  $\bar{x}\langle v \rangle.P \mid Q$  is  $(\bar{x}\langle v \rangle.P) \mid Q$ , and  $\nu z P \mid Q$  is  $(\nu z P) \mid Q$ . A process  $\bar{x}\langle \cdot \rangle.P$  is abbreviated as  $\bar{x}.P$  and, similarly,  $x(\cdot).P$  as  $x.P$ .

## Examples

Below some simple examples are given to illustrate the use of the main constructs. The first example is about encoding of data types. In general, processes interact by passing one another data. The only data of the  $\pi$ -calculus are names. However, it may well be convenient to admit other atomic data, such as integers, and structured data, such as tuples and multisets. It is in the spirit of process calculi generally to allow the data language to be tailored to the application at hand, admitting sets, lists, trees, and so on as convenient. And one *should* admit the relevant data when using the  $\pi$ -calculus to reason about systems. Remarkably, however, all such data can be expressed. For instance,

$$T \stackrel{\text{def}}{=} b(x,y).\bar{x}.\mathbf{0} \quad \text{and} \quad F \stackrel{\text{def}}{=} b(x,y).\bar{y}.\mathbf{0}.$$

represent encodings of the boolean values true and false, located at name  $b$ . These processes receive a pair of names via  $b$ ; then  $T$  will respond by signaling on the first of them, whereas  $F$  will signal on the second. The following process  $R$  reads the value of a boolean located at  $b$  and, depending on the value of the boolean, it will behave as  $P$  or  $Q$ :

$$R \stackrel{\text{def}}{=} \nu t \nu f \bar{b}(t,f). (t.P + f.Q),$$

It is assumed that  $t$  and  $f$  are only used to read a boolean, hence they do not appear in  $P$  and  $Q$ . Now, a system

with both  $R$  and  $T$  can evolve as follows, using  $\rightarrow$  to indicate a single computation step, i.e., a single interaction; below  $\sim$  indicates the application of some simple garbage-collection algebraic laws of the  $\pi$ -calculus that will be discussed later (laws for garbage collecting trailing inactive processes and restrictions on name that are not used anymore).

$$\begin{aligned} T|R &\rightarrow \nu t \nu f (\bar{t}|(t.P+f.Q)) \\ &\rightarrow \nu t \nu f (\mathbf{0}|P) \\ &\sim P \end{aligned}$$

In the first step,  $R$  sends to  $T$  two names,  $t$  and  $f$ , that were initially private to  $R$ ; after the interaction, these names are shared between (what remains of)  $R$  and  $T$ ; no other process in the system (in principle, other processes could run in parallel) will now be able to interfere on the following interaction between  $R$  and  $T$  along  $t$ . The capability of creating new names, and sending them around, is at the heart of the expressiveness and the theory of the  $\pi$ -calculus; without it, the  $\pi$ -calculus would not be much different from CCS.

In the second step, the interaction along  $t$  selects  $P$ , and the other branch of the choice is discarded. In the final line, the garbage-collection laws are applied.

In the example, the act of interrogating a process  $T$  or  $F$  for its value destroys it. Persistence of data is represented using replication. The encoding of persistent booleans would then be

$$\text{TRUE} \stackrel{\text{def}}{=} !b(x,y).\bar{x}.\mathbf{0} \quad \text{FALSE} \stackrel{\text{def}}{=} !b(x,y).\bar{y}.\mathbf{0}$$

Instances of these processes can conduct arbitrarily many dialogues, yielding the same value in each. For example, with  $R$  as above, using  $\Rightarrow$  to indicate the transitive closure of  $\rightarrow$ , and using  $\sim$  as above to indicate application of some cleanup laws, it holds that

$$R|R|\text{TRUE} \Rightarrow \sim P|P|\text{TRUE}$$

and

$$R|R|\text{FALSE} \Rightarrow \sim Q|Q|\text{FALSE}.$$

Our second example is about the encoding of higher-order constructs in the  $\pi$ -calculus. Consider a higher-order language with constructs similar to those of the  $\pi$ -calculus but with the possibility of passing processes. In such a language one could write, for instance,

$$P \stackrel{\text{def}}{=} a(x).(x|x)|\bar{a}(R).Q$$

On the right-hand side of the composition, a process  $R$  is emitted along  $a$ ; on the left-hand side, an input at  $a$  is expecting a process and then two copies of it will be run. Process  $P$  evolves as follows:

$$P \Rightarrow R|R|Q$$

This behavior can be mimicked in the  $\pi$ -calculus as follows. The communication of the higher-order value  $R$  is translated as the communication of a private name that acts as a pointer to (the translation of)  $R$  and that the recipient can use to trigger a copy of (the translation of)  $R$ ; the mapping from the higher-order language into the  $\pi$ -calculus is indicated with  $\llbracket . \rrbracket$ :

$$\llbracket P \rrbracket \stackrel{\text{def}}{=} a(x).(\bar{x}.\mathbf{0}|\bar{x}.\mathbf{0})|\nu y \bar{a}(y).(Q|!y.R)$$

where name  $y$  does not occur elsewhere. It holds that

$$\begin{aligned} \llbracket P \rrbracket &\rightarrow \nu y (\bar{y}.\mathbf{0}|\bar{y}.\mathbf{0}|\llbracket Q \rrbracket|!y.\llbracket R \rrbracket) \\ &\rightarrow \rightarrow \nu y (\mathbf{0}|\mathbf{0}|\llbracket Q \rrbracket|\llbracket R \rrbracket|\llbracket R \rrbracket|!y.\llbracket R \rrbracket) \\ &\sim \llbracket Q \rrbracket|\llbracket R \rrbracket|\llbracket R \rrbracket \end{aligned}$$

where, on the second line,  $\rightarrow \rightarrow$  represents two consecutive reduction steps. On the third line, the same algebraic laws as above are used, plus a law for garbage-collecting an input-replicated process such as  $!y.\llbracket R \rrbracket$  if the initial name  $y$  is restricted and does not occur elsewhere in the system.

The translation separates the acts of *copying* and of *activating* the value  $R$ ; copying is rendered by the replication, and activation by communications along the pointer  $y$ . Here again, it is essential that the pointer  $y$  is initially private to the process emitting at  $a$ . This ensures that the following outputs at  $y$  are not intercepted by processes external to  $\llbracket P \rrbracket$ .

## Names

In the  $\pi$ -calculus, names specify links. But what is a link? The calculus is not prescriptive on this point: the notion of a *link* is construed very broadly, and names can be put to very many uses. This point is important and deserves some attention. For example, names can be thought of as channels that processes use to communicate. Also, by syntactic means and using type systems,  $\pi$ -calculus names can be used to represent names of processes or names of objects in the sense of object-oriented programming. Further, although the  $\pi$ -calculus does not mention locations explicitly, often

when describing systems in  $\pi$ -calculus, some names are naturally thought of as locations. Finally, some names can be thought of as encryption keys as done in calculi that apply ideas from the  $\pi$ -calculus to computer security.

## Types

A type system is, roughly, a mechanism for classifying the expressions of a program. Type systems are useful for several reasons: to perform optimizations in compilers; to detect simple kinds of programming errors at compilation time; to aid the structure and design of systems; to extract behavioral information that can be used for *reasoning* about programs. In sequential programming languages, type systems are widely used and generally well-understood. In concurrent programming languages, by contrast, the tradition of type systems is much less established.

In the  $\pi$ -calculus world, types have quickly emerged as an important part of its theory and of its applications, and as one of the most important differences with respect to CCS-like languages. The types that have been proposed for the  $\pi$ -calculus are often inspired by well-known type systems of sequential languages, especially  $\lambda$ -calculi. Also, type systems specific to processes have been investigated, for instance, for preventing certain forms of interferences among processes or certain forms of deadlocks.

One of the main reasons for which types are important for reasoning on  $\pi$ -calculus processes is the following. Although well-developed, the theory of the pure  $\pi$ -calculus is often insufficient to prove “expected” properties of processes. This is because a  $\pi$ -calculus programmer normally uses names according to some precise logical discipline (the same happens for the  $\lambda$ -calculus, which is hardly ever used untyped since each variable has usually an “intended” functionality). This discipline on names does not appear anywhere in the terms of the pure calculus, and therefore cannot be taken into account in proofs. Types can bring this structure back to light.

This point is illustrated with an example that has to do with *encapsulation*. Facilities for encapsulation are desirable in both sequential and concurrent languages, allowing one to place constraints on the access to components such as data and resources. The need of encapsulation has led to the development of abstract

data types and is a key feature of objects in object-oriented languages. In CCS, encapsulation is given by the *restriction* operator. Restricting a channel  $x$  on a process  $P$ , written  $\nu x P$ , guarantees that interactions along  $x$  between subcomponents of  $P$  occur without interference from outside. For instance, suppose one has two one-place buffers, Buf1 and Buf2, the first of which receives values along a channel  $x$  and resends them along  $y$ , whereas the second receives on  $y$  and resends on  $z$ . They can be composed into a two-place buffer that receives on  $x$  and resends on  $z$ ; thus:  $\nu y (\text{Buf1} \mid \text{Buf2})$ . Here, the restriction ensures us that actions on  $y$  from Buf1 and Buf2 are not stolen by processes in the external environment. With the formal definitions of Buf1 and Buf2 at hand, one can indeed prove that the system  $\nu y (\text{Buf1} \mid \text{Buf2})$  is behaviorally equivalent to a two-place buffer.

The restriction operator provides quite a satisfactory level of protection in CCS, where the visibility of channels in processes is fixed. By contrast, restriction alone is often not satisfactory in the  $\pi$ -calculus, where the visibility of channels may change dynamically. Consider the situation in which several client processes cooperate in the use of a shared resource such as a printer. Data are sent for printing by the client processes along a channel  $p$ . Clients may also communicate channel  $p$  so that new clients can get access to the printer. Suppose that initially there are two clients

$$\begin{aligned} C1 &= \bar{p}\langle j_1 \rangle . \bar{p}\langle j_2 \rangle \dots \\ C2 &= \bar{b}\langle p \rangle \end{aligned}$$

and therefore, writing  $P$  for the printer process, the initial system is

$$\nu p (P \mid C1 \mid C2).$$

One might wish to prove that  $C1$ 's print jobs represented by  $j_1$  and  $j_2$  are eventually received and processed in that order by the printer, possibly under some fairness condition on the printer scheduling policy. Unfortunately this is false: a misbehaving new client  $C3$  that has obtained  $p$  from  $C2$  can disrupt the protocol expected by  $P$  and  $C1$  just by reading print requests from  $p$  and throwing them away:

$$C3 = p(j) . p(j') . 0.$$

In the example, the protection of a resource (the printer) fails if the access to the resource is transmitted, because no assumptions on the use of that access by a recipient can be made. Simple and powerful encapsulation barriers against the mobility of names can be created using type concepts familiar from the literature of typed  $\lambda$ -calculi. For instance, the misbehaving printer client C3 can be recognized by distinguishing between the input and the output capabilities of a channel. It suffices to assign the input capability on channel  $p$  to the printer and the output capability to the initial clients C1 and C2. In this way, new clients that receive  $p$  from existing clients will only receive the output capability on  $p$ . The misbehaving C3 is thus ruled out as ill-typed, as it uses  $p$  for input.

The concept of a channel with direction can be formalized by means of type constructs, sometimes called the *input/output types*. They give rise to a natural subtyping relation, similar to those used for reference types in imperative languages. In the case of the  $\pi$ -calculus encodings of the  $\lambda$ -calculus, this subtyping validates the standard subtyping rules for function types. It is also important when modeling object-oriented languages, whose type systems usually incorporate some powerful form of subtyping.

In the  $\lambda$ -calculus, where functions are the unit of interaction, the key type construct is the arrow type. In the  $\pi$ -calculus names are the unit of interaction and therefore the key type construct is the *channel* (or *name*) type  $\# T$ . A type assignment  $a : \# T$  means that  $a$  can be used as a channel to carry values of type  $T$ . As names can carry names,  $T$  itself can be a channel type. If one adds a set of *basic types*, such as integer or boolean types, one obtains the analog of the simply-typed  $\lambda$ -calculus, which is therefore called the *simply-typed  $\pi$ -calculus*. Type constructs familiar from sequential languages, such as those for products, unions, records, variants, recursive types, polymorphism, subtyping, and linearity, can be adapted to the  $\pi$ -calculus. Having recursive types, one may avoid basic types as initial elements for defining types. The calculus with channel, product, and recursive types is the *polyadic  $\pi$ -calculus* mentioned earlier on.

Beyond types inherited from the  $\lambda$ -calculus, several other type systems have been put forward that are specific to processes; they formalize common *patterns of interaction* among processes. Types may even serve as

a specification of (parts of) the behavior of processes, as in *session types*. The behavioral guarantees that are guaranteed by types are typically *safety properties*, such as the absence of certain communication errors, of interferences, of deadlock, and information leakage in security protocols. Safety is expressed in the subject reduction theorem, a fundamental theorem of type theory stating the invariance of types under reduction. Type systems for *liveness properties* have been studied too, for instance for properties such as termination (the fact that computation in a concurrent system will eventually stop) and responsiveness (the fact that a process will eventually provide an answer along a certain name). However, liveness properties, already difficult enough to prove in the presence of concurrency, can be even harder to prove for mobile processes (in the sense that it may be difficult to design a type system capable of guaranteeing the property of interest while being expressive enough to handle most common programming idioms).

## Theory

Fundamental for a theory of a concurrent language is a means of stating what the behavior of a process is, and what does it mean for two behaviors to be equal. These notions in concurrency are usually treated via operational semantics. A brief and informal account of how these issues are treated in the  $\pi$ -calculus is given below, referring to [18] for details.

Traditionally, the operational semantics of a process algebra is given in terms of a *labeled transition system* describing the possible evolutions of a process. This contrasts with what happens in *term rewriting systems*, as based on an *unlabeled reduction system*. In the  $\lambda$ -calculus, probably the best known term-rewriting system, what makes a reduction system possible is that two terms having to interact are naturally in contiguous positions. This is not the case in process calculi, where interaction does not depend on physical contiguity. To put this another way, a *redex* of a  $\lambda$ -term is a subterm, while a “redex” in a process calculus is distributed over the term.

To allow a reduction semantics on processes, axioms for a *structural congruence* relation, usually written  $\equiv$ , are introduced prior to the reduction system, in order to break a rigid, geometrical view of concurrency; then reduction rules can easily be presented in which redexes are indeed subterms again.

The interpretation of the operators of the language emerges neatly with a reduction semantics, due to the compelling naturalness of each structural congruence and reduction rule. This is not quite the case in the labeled semantics, at least for process algebras expressing mobility: the manipulation of names and the side conditions in the rules are nontrivial and this can make it delicate understanding and justifying the choices made. However, if the reduction system is available, the correctness of the labeled transition system can be shown by proving the correspondence between the two systems.

On the other hand, the advantages of a labeled semantics appear later, when reasoning with processes. In a reduction semantics, the behavior of a process is understood relatively to a context in which it is contained and with which it interacts. Instead, with a labeled semantics every possible communication of a process can be determined in a direct way. This allows us to get simple characterizations of behavioral equivalences. Moreover, with a labeled semantics the proofs benefit from the possibility of reasoning in a purely structural way. Another possible problem with a reduction semantics is that it allows one to accommodate only limited forms of the choice operator. The conclusion is that both semantics are useful and that they integrate and support each other.

To see an example of the two semantics, consider the process

$$S \stackrel{\text{def}}{=} x(y). P|\bar{z}(w). R|\bar{x}(v). Q$$

A reduction semantics would only specify that  $S$  has a reduction

$$S \longrightarrow P\{v/y\}|\bar{z}(w). R|Q$$

To derive this, structural congruence is first used to bring the participant of the interaction into contiguous positions, using monoidal rules for parallel composition such as

$$T_1 | T_2 \equiv T_2 | T_1$$

$$T_1 | (T_2 | T_3) \equiv (T_1 | T_2) | T_3$$

with which the order of the components in parallel compositions can be modified. Then the rewriting rule

$$a(b). T|\bar{a}(u). T' \longrightarrow T\{u/b\}|T'$$

can be applied.

In contrast, a labeled transition system reveals, beside the above reduction, also the potentials for  $S$  to interact with the environment, namely the input and output transitions:

$$\begin{array}{ll} S \xrightarrow{x(y)} P|\bar{z}(w). R|\bar{x}(v). Q & S \xrightarrow{\bar{x}(v)} x(y). P|\bar{z}(w). R|Q \\ S \xrightarrow{\bar{z}(w)} x(y). P|R|\bar{x}(v). Q & \end{array}$$

(Note that, with a restriction in front of  $S$ , name  $x$  becomes private to  $S$  and the input and output actions along  $x$  are forbidden.) The rules for the labeled transition semantics are formalized following the style of Plotkin's Structured Operational Semantics, where the derivation proof of a transition is uniquely determined by the position, in the syntax of the term, of the prefixes consumed in the transition.

The notion of behavioral equivalence normally adopted in the  $\pi$ -calculus is *barbed congruence*. Its definition is couched in terms of a *bisimulation* game on reductions and a simple notion of *observation* on processes (for instance, a predicate revealing whether a process is capable of emitting a signal at some special name). Two  $\pi$ -terms are deemed barbed congruent if no difference can be observed between the processes obtained by placing them into an arbitrary  $\pi$ -context. The notion of observation has the flavor of the report of the successful outcome of an experiment, as in a theory of *testing*.

Barbed congruence has the advantage of being simple and robust, in that it can be applied to different calculi. Moreover, as its definition involves quantification over contexts, it gives a natural notion of equivalence, properly sensitive to the calculus under consideration (this is important when considering, for instance, type systems, as types implicitly limit the class of contexts in which a process may be placed; as a consequence, more equalities among processes hold). The quantification over contexts, however, makes the definition difficult to work with. This fact motivates the study of auxiliary notions of behavioral equivalences that afford tractable techniques. These *labeled* equivalences are based on direct comparison of the actions that processes can perform, rather than on the observation of arbitrary systems containing them. There are in fact several notions of labeled equivalence, each of which has characteristics that make it useful in some way. Examples are

*late bisimilarity*, *early bisimilarity*, and *open bisimilarity* [13, 18]. The labeled equivalences are not as robust as barbed congruence. For instance, they can be much too discriminating on refinements of the  $\pi$ -calculus with types.

A number of proof techniques for behavioral equalities on  $\pi$ -calculus processes have been developed. For instance, enhancements of the bisimulation proof method, sometimes called “up-to” techniques [17]. There has been also some work on the development of (semi-) automatic tools to assist in reasoning, though this remains an active research area. Other behavioral equivalences for the  $\pi$ -calculus have been studied, too, for instance *testing equivalence* [3].

The  $\pi$ -calculus has also a well-developed *algebraic* theory. Equational reasoning is a central technique in process calculus. In carrying out a calculation, appeal can be made to any axiom or rule sound for the equivalence in question.

In general, the equivalence of two processes is undecidable, indeed it is not even semi-decidable. On finite processes, however, axiomatizations may be possible. By an *axiomatization* of an equivalence on a set of terms, one means some equational axioms that, together with the rules of equational reasoning, suffice for proving all (and only) the valid equations. The rules of equational reasoning are reflexivity, symmetry, transitivity, and congruence rules that make it possible to replace any subterm of a process by an equivalent term. Here are examples of axioms for the  $\pi$ -calculus. These axioms are sound, in that the processes so equated are barbed congruent. See [18] for more details.

$$P + P = P$$

$$\nu x \bar{x}(v). P = 0$$

$$\nu x P = P \quad \text{if } x \text{ does not occur in } P$$

$$\begin{aligned} x(y). P | \bar{z}(v). Q &= x(y). (P | \bar{z}(v). Q) + \bar{z}(v). (x(y). P | Q) \\ &\quad + [x=z]\tau. (P\{v/y\}|Q) \end{aligned}$$

The first law is an idempotence law for choice. The second law shows that an output at a name  $x$  preceded by a restricted on  $x$  is a blocked process. The third law allows the removal a useless restriction. The final law, called *expansion*, explains the behavior of a parallel composition in terms of choice and matching.

A behavioral equivalence abstracts from internal action is sometimes called a *weak* equivalence, and one that does not a *strong* equivalence. The above laws are valid for strong barbed congruence, hence also for its weak counterpart. Here is an equality that is only valid for weak barbed congruence:

$$R|\nu x (x(y). P | \bar{x}(v). Q) = R|\nu x (P\{v/y\}|Q)$$

The law shows that an interaction between two processes along private names cannot be disturbed by other processes.

## Variants and Extensions

In the  $\pi$ -calculus, communication between processes is synchronous – it is a handshake synchronization between two processes. Variants of the  $\pi$ -calculus have been proposed in which communication may be thought of as being asynchronous. The most distinctive feature of such extensions is that there is no continuation underneath the output prefix (i.e., all outputs are of the form  $\bar{x}(v)$ ), and only limited forms of choice are allowed. A major advantage of the asynchronous variants is that their implementation is simpler, and indeed most programming languages, or constructs for programming languages, inspired by  $\pi$ -calculus are asynchronous.

The ordinary  $\pi$ -calculus does not distinguish between channels, or ports, and variables: they are all names. In some variants, such a separation is made. The  $\pi$ -calculus also does not explicitly mention location or distribution of mobile processes. The issue of location and distribution is orthogonal. Several extensions, or variants, of the  $\pi$ -calculus have appeared with the goal of explicitly addressing distribution and all the associated phenomena. Ideas from  $\pi$ -calculus have contributed to the development of their theories. Examples of languages are the Distributed Join Calculus [6], the Distributed  $\pi$ -calculus [8], the Ambient Calculus [5], and Oz [19].

## Related Entries

- ▶ [Actors](#)
- ▶ [Bisimulation](#)
- ▶ [CSP \(Communicating Sequential Processes\)](#)
- ▶ [Process Algebras](#)

## Bibliographic Notes and Further Reading

The first paper on the  $\pi$ -calculus, [12], was written by Milner et al. A book that gives a gentle introduction to both CCS and the  $\pi$ -calculus, with an emphasis on motivating the interest in them, is [11]. A book with an in-depth treatment of the theory of the  $\pi$ -calculus and its main variants is [18]. A book with a focus on a distributed extension of the  $\pi$ -calculus is [9]. These textbooks may be consulted for details on the concepts outlined above, including type systems, variants of the  $\pi$ -calculus, behavioral theory, and comparison with higher-order languages.

Recent developments, not covered in the above books, include session types [20], type systems for deadlock-freedom, and lock-freedom such as [10]. Experimental typed programming languages, or proposals for typed programming languages, inspired by the  $\pi$ -calculus include Pict [14], Join [7], *XLang*, developed at Microsoft as a component of the .NET platform. An active research area is the application of concepts from the  $\pi$ -calculus to languages aimed for Web services (see, e.g., [4]), biology (see, e.g., [15, 16]), and security (see, e.g., [1, 2]).

## Bibliography

1. Abadi M, Gordon AD (1999) A calculus for cryptographic protocols: the spi calculus. *Inf Comput* 148(1):1–70
2. Blanchet B, Abadi M, Fournet C (2008) Automated verification of selected equivalences for security protocols. *J Log Algebr Program* 75(1):3–51
3. Boreale M, De Nicola R (1995) Testing equivalence for mobile processes. *Inf Comput* 120:279–303
4. Carbone M, Honda K, Yoshida N (2007) Structured communication-centred programming for web services. In: Proceedings of the ESOP 2007, vol 4421, Lecture Notes in Computer Science. Springer, Heidelberg, pp 2–17, 2007
5. Cardelli L, Gordon AD (1998) Mobile ambients. In: Proceedings of the FoSSaCS'98, vol 1378, Lecture Notes in Computer Science. Springer, Heidelberg, pp 140–155, 1998
6. Fournet C, Gonthier G, Lévy J-J, Maranget L, Rémy D (1996) A calculus of mobile agents. In: Proceedings of the CONCUR'96, vol 1119, Lecture Notes in Computer Science. Springer, Heidelberg, pp 406–421, 1996
7. Fournet C, Gonthier G (2002) The join calculus: a language for distributed mobile programming. In: Summer School APPSEM 2000, vol 2395, Lecture Notes in Computer Science. Springer, Heidelberg, pp 268–332
8. Hennessy M, Riely J (1998) Resource access control in systems of mobile agents. In: Proceedings of the HLCL '98: High-Level Concurrent Languages, vol 16.3, ENTCS. Elsevier Science, 1998
9. Hennessy M (2007) A distributed pi-calculus. Cambridge University Press, New York
10. Kobayashi N (2006) A new type system for deadlock-free processes. In: Proceedings of the CONCUR'06, vol 4137, Lecture Notes in Computer Science. Springer, Bonn, pp 233–247, 2006
11. Milner R (1999) Communicating and mobile systems: the  $\frac{1}{4}$ -Calculus. Cambridge University Press, Cambridge
12. Milner R, Parrow J, Walker D (1993) A calculus of mobile processes, (Parts I and II). *Inf Comput* 100:1–77
13. Milner R, Parrow J, Walker D (1992) Modal logics for mobile processes. *Theor Comput Sci* 114:149–171
14. Pierce BC, Turner DN (2000) Pict: a programming language based on the pi-calculus. In: Proof, Language and Interaction: Essays in Honour of Robin Milner. MIT Press, Cambridge
15. Priami C, Quaglia P, Romanel A (2009) Blenx static and dynamic semantics. In: Proceedings of the CONCUR'09, vol 5710, Lecture Notes in Computer Science. Springer, Bologna, pp 37–52, 2009
16. Regev A, Panina EM, Silverman W, Cardelli L, Shapiro EY (2004) Bioambients: an abstraction for biological compartments. *Theor Comput Sci* 325(1):141–167
17. Sangiorgi D (1995) On the bisimulation proof method. In: Proceedings of the MFCS'95, vol 969, Lecture Notes in Computer Science. Springer, pp 479–488, 1995
18. Sangiorgi D, Walker D (2001) The  $\frac{1}{4}$ -calculus: a theory of mobile processes. Cambridge University Press, Cambridge
19. Smolka G (1994) The definition of kernel Oz. Research Report RR-94-23, Deutsches Forschungszentrum für Künstliche Intelligenz, Kaiserslautern, Germany
20. Vasconcelos VT (2009) Fundamentals of session types. In: SFM 2009 School, vol 5569, Lecture Notes in Computer Science. Springer, Heidelberg, pp 158–186

## Pipelining

Pipelining [1] is a parallel processing strategy in which an operation or a computation is partitioned into disjoint stages. The stages must be executed in a particular order (could be a partial order) for the operation or computation to complete successfully. Each stage is implemented as a component which could be a hardware device or a software thread. When a stage completes, it becomes available to do other work. Parallelism results from the execution of a sequence of operations

or computations so that at any given time several components of the sequence are under execution and each one of these is at a different stage of the pipeline.

Pipelining is pervasive in today's machines. Processor control units and arithmetic units are typically pipelined. Also, programs take advantage of pipelined parallelism by partitioning computations into stages.

## Related Entries

- ▶ [Cray Vector Computers](#)
- ▶ [Floating Point Systems FPS-120B and Derivatives](#)
- ▶ [Fujitsu Vector Computers](#)
- ▶ [Stream Programming Languages](#)

## Bibliography

1. Kogge PM (1981) The Architecture of Pipelined Computers. Hemisphere Publishing Corporation, New York

## Place-Transition Nets

- ▶ [Petri Nets](#)

## PLAPACK

JOHN A. GUNNELS  
IBM Corp, Yorktown Heights, NY, USA

## Definition

PLAPACK is a software library for dense parallel linear algebra computations.

## Discussion

### Introduction

The PLAPACK (van de Geijn, Robert., *Using PLAPACK*, pp. 1, 3, 4, 6, 43, 59, 85, ©1997 Massachusetts Institute of Technology, by permission of the MIT Press) library is a modern, dense parallel linear algebra library that is extensible, easy to use, and available under an open source license. It is designed to be user-friendly while offering competitive performance when compared to the more traditionally constructed ScaLAPACK library.

## The PLAPACK Project

**Motivation** PLAPACK's design was motivated by the observation that the parallel implementation of most dense linear algebra operations is a relatively well-understood process. Nonetheless, the creation of general-purpose, high-performance parallel dense linear algebra libraries is severely hampered by the fact that translating the sequential algorithms to a parallel code requires careful manipulation of indices and parameters describing the data, its distribution to processors, and/or the communication required. The creators of PLAPACK believe that these details make such parallel programming highly error prone and that this overhead stands in the way of the parallel implementation of more sophisticated algorithms.

**An Object-Based Approach** The PLAPACK library is constructed so as to allow the user to express their parallel algorithms in a very concise manner. It does this, largely, through the encapsulation of data in objects. To achieve this, PLAPACK adopted an “object-based” (or object-oriented) approach to programming, inspired by efforts including the Message Passing Interface (MPI) library [1], and the PETSc library [2]. In object-based programming, all of the data related to an operand (matrix, vector, etc.) is cohesive, maintained in a data structure that describes the object. Further, the descriptor is interrogated or modified only indirectly, through the use of accessor and modifier functions.

### Objects and Communications

**Object Types** In the PLAPACK library there are several different types of linear algebra objects. These object types have a one-to-one correspondence with the manner in which they are distributed.

The PLAPACK object types include:

- **PLA\_Matrix:** This object is distributed as a two-dimensional object, in a particular block cyclic fashion referred to as the Physically Based Matrix Distribution (PBMD).
- **PLA\_Mvector:** The multivector is distributed over the compute fabric in blocked fashion, viewing the grid as one dimensional. It is used as a vector operand and as an intermediate form for copying data from one object (form) to another.

A multivector object may consist of one or more “columns” of data.

- **PLA\_Pmvector:** The projected multivector object is distributed in the same manner as (a set of) rows or columns of a matrix. This object is often duplicated in one dimension of the computational grid. For example, a column-oriented projected multivector is distributed across rows of the processor mesh and may be duplicated across processor columns. PLAPACK code often uses this object type to create buffers for communication.
- **PLA\_Mscalar:** The multiscalar object is most conveniently thought of as a local matrix. It is not distributed, but replicated on all processors.

**The Special Role of the Multivector** Multivectors can be thought of as a number of vectors, side-by-side, distributed over the entire processor mesh, where that mesh is viewed as a one-dimensional array of processors. The multivector can be used as a first-class object or as an intermediate form in various kinds of communication. An example of the former would be the redistribution of the panel from a blocked LU decomposition. In taking the object from, typically, a column of processors to the entire processor mesh, more computation per unit of time can be applied to the object. The latter use of the multivector appears in the `PLA_Copy` and `PLA_Reduce` routines. Copying data from one object to another or reducing (e.g., summing) data from several objects into another potentially involves complex mapping and communication, but these are encapsulated in the `PLA_Copy` and `PLA_Reduce` routines, respectively. The copy routine is used to seemlessly move data between objects of potentially differing distributions and types while the reduction operator is used to do the same, but is logically limited to using a duplicated object as its source. The central nature of the multivector in the copy operation is illustrated in Fig. 1.

## Referencing (Sub)Objects

**Views** In order to both facilitate concise algorithmic encodings and to eliminate many indexing errors, PLAPACK makes heavy use of linear algebra object (matrix, vector, etc.) “views.” In this context, a view is, abstractly, a subset of an object (a submatrix is the most commonly used view) with its own descriptor.

The promotion of the view to a first-class entity in PLAPACK stems from the fact that many common linear algebra algorithms such as parallel BLAS routines, solvers, and factorization routines, can be expressed in terms of windows into matrices and vectors. Thus, the use of the view (implicit or explicit) is common. By formalizing the use of the view and localizing the functionality related to creating and modifying these objects, PLAPACK both removes the need for the user to create this functionality and places it in a well-tested routine. As the routines used for views are heavily tested and the level of abstraction used to create views is high, relative to explicit indexing, this functionality is intended to reduce the opportunity for programmer error and to make such errors, when introduced, easier to locate.

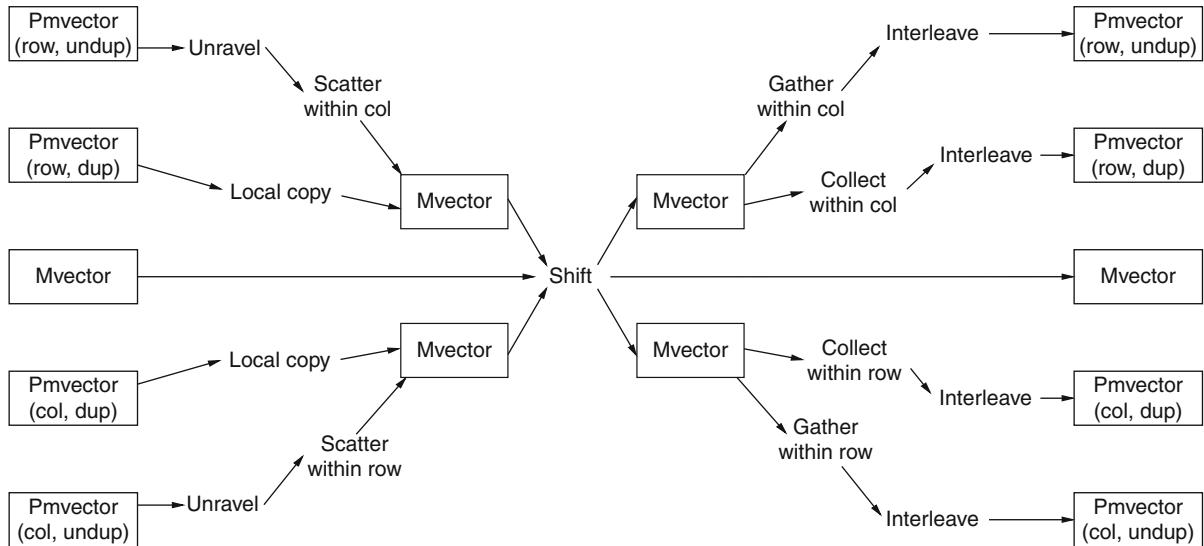
A view is identical to an object created via the `PLA_Obj_create` function except that it references the same data as the parent object or view from which it was derived. Thus, a change to the values in the data that the view translates to a corresponding change in the data that the parent object describes. It is legitimate in PLAPACK to derive views from views.

In PLAPACK, a view of a linear algebra object is created by a call to the `PLA_Obj_view` routine. The user specifies the dimensions of the new view as well as its offset relative to the parent object and this routine creates a new view into an existing object. Routines for shifting (sliding) views as well as specialized subview operators, partitioning matrices into multiple submatrices or coalescing submatrices into a single-view object are also supplied in the PLAPACK library.

## Distributing and Interfacing with Parallel Operands

**Physically Based Matrix Distribution** PLAPACK employs an object distribution called the “Physically Based Matrix Distribution” (PBMD). This distribution scheme is based on the thesis that the elements of vectors are typically associated with data of physical significance, and it is therefore their distribution to nodes that is directly related to the distribution of the problem to be solved. From this point of view, a matrix (discretized operator) merely represents the relation between two vectors (discretized spaces):

$$y = Ax \quad (1)$$



**PLAPACK. Fig. 1** A systematic approach to copying (duplicated) (projected) (multi)vectors from and to (duplicated) (projected) (multi)vectors

PLAPACK partitions  $x$  and  $y$ , and assigns portions of these vectors to nodes. The matrix  $A$  should then be distributed to nodes in a fashion consistent with the distribution of the vectors.

To employ PBMD, one must start by describing the distribution of the vectors, here,  $x$  and  $y$ , to nodes, after which the matrix distribution is induced (derived) by the vector distribution as illustrated in Fig. 2.

**The Application Programming Interface** PLAPACK was architected under the assumption that applications will employ the library to:

- Create PLAPACK vector, multivector, and matrix objects.
- Fill the entries in these PLAPACK objects.
- Perform a series of parallel linear algebra operations.
- Query elements of the updated PLAPACK objects.

Many applications are inherently set up to generate numerous sub-vector, sub-multivector, or submatrix contributions to global linear algebra objects. The contributions of these applications can be viewed as dimensional “sub-objects.” However, frequently they are also partial sums of the global linear algebra objects, in which case the contribution must be added to existing entries. One approach for parallelizing these applications is to partition the numerous sub-object computations among processors. Such a parallelization of

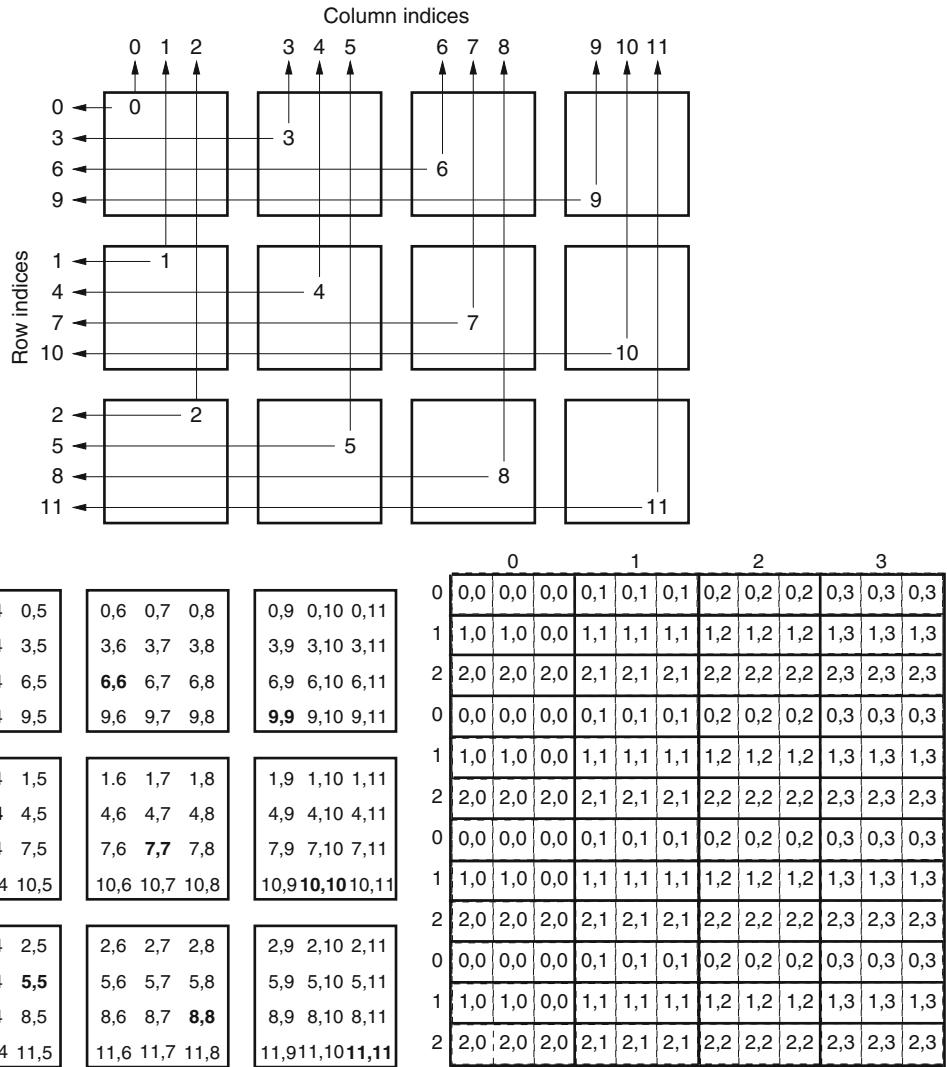
the applications’ linear algebra object generation phase produces sub-vector, sub-multivector, and submatrix partial sums that may or may not be entirely local with respect to the data distribution of the global linear algebra objects.

In order to fill or retrieve entries in a linear algebra object, an application enters the API-active state of PLAPACK. In this state, objects can be opened in a shared-memory-like mode, which allows an application to either fill or retrieve individual elements, sub-vectors, or subblocks of linear algebra objects. The PLAPACK application interface supports filling global linear algebra objects with sub-object partial sums.

### An Illustrative Example

**Parallel Cholesky Factorization** Fig. 3 is a PLAPACK implementation of a blocked, parallel Cholesky factorization. It utilizes level-3 BLAS routines locally and is a simple, but efficient implementation of this algorithm. The representation is compact and relatively easy to explain to those familiar with the underlying sequential algorithm (see, e.g., the encyclopedia entry on libflame).

The algorithm proceeds by applying (local) Cholesky factorization to the upper left corner of the matrix of interest, updating the remainder of the matrix accordingly, and reducing the active area (the part that will be further updated) of the matrix. The same algorithm



**PLAPACK.** Fig. 2 Inducing a matrix distribution from vector distributions. *Top:* Here each box represents a node of a  $3 \times 4$  mesh. The sub-vectors of  $x$  and  $y$  are assigned to the mesh in column-major order. Thus the number in the box represents the index of the sub-vector assigned to that node. By projecting the indices of  $y$  to the left, the distribution of the matrix row-blocks of  $A$  is established. By projecting the indices of  $x$  to the top, the distribution of the matrix column-blocks of  $A$  is determined. *Bottom-left:* The resulting distribution of the subblocks of  $A$  is given where the indices refer to the indices of the subblocks of  $A$ . *Bottom-right:* The same information, except now from the matrix point of view. The figure shows the matrix partitioned into subblocks, with the indices in the subblocks indicating the node to which the subblock is mapped

is applied to this active part of the matrix until there is no active matrix remaining and the algorithm is complete.

In the implementation in Fig. 3a, the while loop continues until the active portion of the matrix no longer exists. That condition is signaled when the top-left ( $A_{TL}$ ) quadrant of the matrix (the part that has

been completely factored) is the entire matrix. Since the matrix is assumed to be (globally) square this condition can be tested by comparing the (global) length of the entire matrix to the factored (sub)section.

The calls to `PLA_Obj_split_size` are used to determine the matrix dimensions of the top-left quadrant of the active matrix that resides on a single

```

int Chol_blk_var3(PLA_Obj A, int nb_alg)
{
 PLA_Obj ATL=NULL, ATR=NULL, A00=NULL, A01=NULL, A02=NULL,
 ABL=NULL, ABR=NULL, A10=NULL, A11=NULL, A12=NULL,
 A20=NULL, A21=NULL, A22=NULL;
 PLA_Obj MINUS_ONE=NULL, ZERO=NULL, ONE=NULL;
 int b;
 /* Create constants -1, 0, 1 of the appropriate type */
 PLA_Create_constants_conf_to(A, &MINUS_ONE, &ZERO, &ONE);
 PLA_Part_2x2(A, &ATL, &ATR,
 &ABL, &ABR, 0, 0, PLA_TL);
 while (PLA_Obj_length(ATL) < PLA_Obj_length(A)){
 /* Determine how big of a block A11 exists within one block */
 PLA_Obj_split_size(ABR, PLA_SIDE_TOP, &size_top, &owner_top);
 PLA_Obj_split_size(ABR, PLA_SIDE_LEFT, &size_left, &owner_left);
 b = min(min(size_top, size_left), nb_alg);
 PLA_Repart_2x2_to_3x3(ATL, /**/ ATR, &A00, /**/ &A01, &A02,
 /* ***** */ /* ***** */ &A10, /**/ &A11, &A12,
 ABL, /**/ ABR, &A20, /**/ &A21, &A22,
 b, b, PLA_BR);
 /*-----*/
 /* Update A_11 <- L_11 = Chol. Fact.(A_11) */
 PLA_Local_chol(PLA_LOWER_TRIANGULAR, A11);
 /* Update A_21 <- L_21 = A_21 inv(L_11') */
 PLA_Trsm(PLA_SIDE_RIGHT, PLA_LOWER_TRIANGULAR,
 PLA_TRANSPOSE, PLA_NONUNIT_DIAG, ONE, A11, A21);
 /* Update A_22 <- A_22 - L_21 * L_21' */
 PLA_Syrk(PLA_LOWER_TRIANGULAR, MINUS_ONE, A21, ONE, ABR);
 /*-----*/
 PLA_Cont_with_3x3_to_2x2(&ATL, /**/ &ATR, A00, A01, /**/ A02,
 A10, A11, /**/ A12,
 /* ***** */ /* ***** */ &ABL, /**/ &ABR, A20, A21, /**/ A22,
 PLA_TL);
 }
 PLA_Obj_free(&ATL); PLA_Obj_free(&ATR);
 PLA_Obj_free(&ABL); PLA_Obj_free(&ABR);
 PLA_Obj_free(&A00); PLA_Obj_free(&A01); PLA_Obj_free(&A02);
 PLA_Obj_free(&A10); PLA_Obj_free(&A11); PLA_Obj_free(&A12);
 PLA_Obj_free(&A20); PLA_Obj_free(&A21); PLA_Obj_free(&A22);
 PLA_Obj_free(&MINUS_ONE); PLA_Obj_free(&ZERO); PLA_Obj_free(&ONE);
 return PLA_SUCCESS;
}

```

**PLAPACK. Fig. 3** An efficient variant of Cholesky factorization implemented in the PLAPACK API

processor. This is done so that an efficient, local (no communication) Cholesky factorization can be performed via `PLA_Local_chol`.

The `PLA_Repart_2x2_to_3x3` and `PLA_Cont_with_3x3_to_2x2` calls create multiple views from a single object and coalesce multiple views into a single object, respectively. The comment bars in the code are visual cues as to the semantics of these functions. In the case of `PLA_Repart_2x2_to_3x3` function, ABR is split into four subviews. In order to unambiguously carry out this operation only the size

of the A11 subview needs to be specified (here, that size is b by b). The `PLA_Cont_with_3x3_to_2x2` function coalesces nine views into four. This is done in order to update the view of the active part of the matrix, shrinking it.

Computational work is performed through the calls to `PLA_Local_chol`, which performs the sequential Cholesky factorization, `PLA_Trsm` whose purpose is to perform a parallel triangular solve with multiple right-hand-sides, and `PLA_Syrk`, a parallel version of the symmetric rank-k update.

## Related Entries

- [BLAS \(Basic Linear Algebra Subprograms\)](#)
- [LAPACK](#)
- [libflame](#)
- [ScalAPACK](#)

## Bibliographic Notes and Further Reading

The PLAPACK notation and APIs were used as the basis for those employed in the FLAME [3] project and the libflame library [4].

The first papers that outlined the ideas that PLAPACK is based upon were published in 1997 [5] and a book has been published that describes PLAPACK in detail [6].

## Bibliography

1. Gropp W, Lusk E, Skjellum A (1994) Using MPI. The MIT Press, Cambridge, MA
2. Balay S, Gropp W, McInnes LC, Smith B (1996) PETSc 2.0 users manual. Technical report ANL-95/11, Argonne National Laboratory, Cass Avenue Argonne
3. Gunnels JA, Gustavson FG, Henry GM, van de Geijn RA (2001) FLAME: formal linear algebra methods environment. ACM T Math Software 27(4):422–455
4. Van Zee FG (2009) Libflame: the complete reference. <http://www.lulu.com/content/5915632/>
5. Alpatov P, Baker G, Edwards HC, Gunnels J, Morrow G, Overfelt J, van de Geijn R (1997) PLAPACK: parallel linear algebra package design overview. In: Proceedings of the 1997 ACM/IEEE conference on supercomputing, ACM, New York, pp 29
6. van de Geijn RA (1997) Using PLAPACK. The MIT Press, Cambridge, MA

---

## PLASMA

Jack Dongarra, Piotr, Luszczek  
University of Tennessee, Knoxville, TN, USA

### Definition

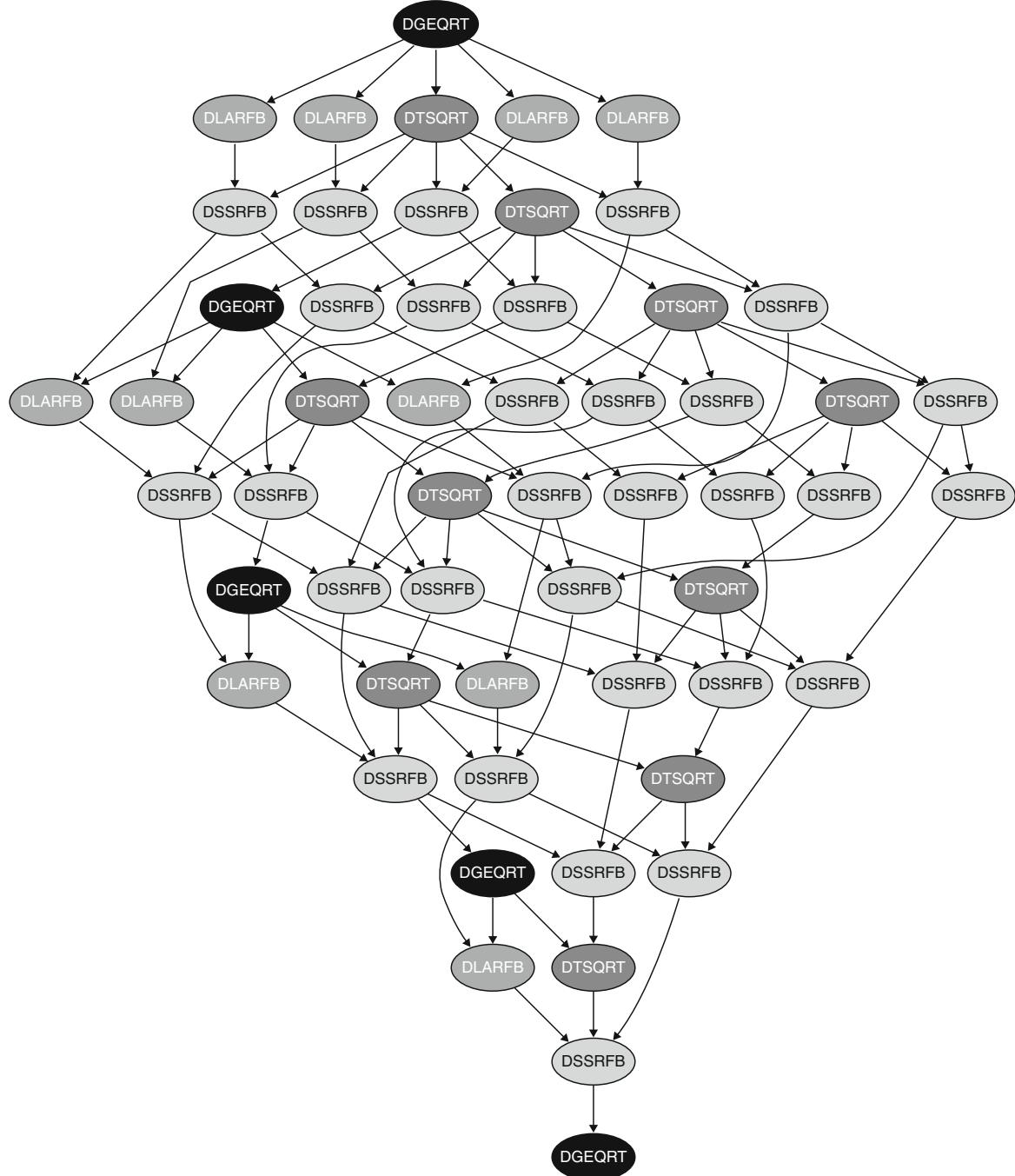
Parallel Linear Algebra Software for Multi-core Architectures (PLASMA) is a free and open-source software library for numerical solution of linear equation systems on shared memory computers with multi-core processors. In particular, PLASMA is designed to give high efficiency on homogeneous multi-core processors and

multi-socket systems of multi-core processors. As of today, majority of such systems are on-chip symmetric multiprocessors with classic super-scalar processors as their building blocks augmented with short-vector SIMD extensions (such as SSE and Altivec). PLASMA is available for download from the PLASMA Web site (To obtain the PLASMA library logon to: <http://icl.cs.utk.edu/plasma/>).

### Discussion

The emergence of multi-core microprocessor designs marked the beginning of a forced march toward an era of computing in which research applications must be able to exploit parallelism at a continuing pace and unprecedented scale [1]. To answer this challenge PLASMA redesigns LAPACK [2] and ScalAPACK [3] for current and future multi-core processor architectures. To achieve high performance on this type of architectures, PLASMA relies on tile algorithms, which provide fine granularity parallelism. The standard linear algebra algorithms can then be represented as a Directed Acyclic Graph (DAG) [4] where nodes represent tasks, either panel factorization or update of a block-column, and edges represent dependencies among them. Figure 1 shows a small DAG for a tile QR factorization. Tasks of the same type (implemented by the same function) share the same color of nodes.

Moreover, the development of programming models that enforce asynchronous, out of order scheduling of operations is the concept used as the basis for the definition of a scalable yet highly efficient software framework for computational linear algebra applications. In PLASMA, parallelism is no longer hidden inside Basic Linear Algebra Subprograms (BLAS – for more details see <http://www.netlib.org/blas/>) but is brought to the fore to yield much better performance. Each of the one-sided tile factorizations presents unique challenges to parallel programming. Cholesky factorization is represented by a DAG with relatively little work required on the critical path. LU and QR factorizations have corresponding dependency pattern between the nodes of the DAG. These two factorizations exhibit much more severe scheduling and constraints than the Cholesky factorization. Currently, PLASMA schedules tasks statically while balancing the trade-off between load balancing and data reuse. PLASMA's performance depends



**PLASMA. Fig. 1** Task DAG for tile QR factorization

strongly on tunable execution parameters, the outer and inner blocking sizes, that trade off utilization of different system resources. The outer block size (NB) trades off parallelization granularity and scheduling flexibility with single core utilization, while the inner block

size (IB) trades off memory load with extra flops due to redundant calculations. Tuning PLASMA consists of finding (NB, IB) pairs that maximize the performance depending on the matrix size and the number of cores.

In terms of numerical calculations, the Cholesky factorization represents one case in which a well-known LAPACK algorithm can be easily reformulated in a tiled fashion. Each operation that defines an atomic step of the LAPACK algorithm can be broken into a sequence of tasks where the same algebraic operation is performed on smaller portions of data, i.e., the tiles. In most of the cases, however, the same approach cannot be applied and novel algorithms must be introduced. For the LU and QR [5, 6] factorizations, algorithms based on the updating factorizations are used to reformulate the algorithms as was done for out-of-core solvers. Updating factorization methods can be used to derive tiled algorithms for LU and QR factorizations that provide very fine granularity of parallelism and the necessary flexibility that is required to exploit dynamic scheduling of the tasks. It is worth noting, however, that, as in any case where such fundamental changes are made, trade-offs have to be taken into account. For instance, in the case of the LU factorization the tiled algorithm replaces partial pivoting with block pairwise pivoting which results in, on average, slightly worse numerical stability. On the positive side, the current analysis of PLASMA's tile algorithms suggests that they are numerically backward stable and the ongoing research aims to determine if they are stable.

PLASMA provides routines to solve dense general systems of linear equations, symmetric positive definite systems of linear equations, and linear least squares problems using LU, Cholesky, QR, and LQ factorizations. Real arithmetic and complex arithmetic are supported in both single precision and double precision.

## Related Entries

- ▶ [LAPACK](#)
- ▶ [Linear Algebra, Numeric](#)
- ▶ [ScaLAPACK](#)

## Bibliography

1. Herb Sutter A (2005) Fundamental turn toward concurrency in software. Dr. Dobb's J 30(3):202–210
2. Anderson E, Bai Z, Bischof C, Blackford LS, Demmel JW, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D (1992) LAPACK Users' guide. SIAM, Philadelphia
3. Blackford LS, Choi J, Cleary A, D'Azevedo E, Demmel J, Dhillon I, Dongarra J, Hammarling S, Henry G, Petitet A, Stanley K, Walke D, Whaley RC (1997) ScaLAPACK users' guide. SIAM, Philadelphia

4. Christofides N (1975) Graph theory: an algorithmic approach. Academic, New York
5. Buttari A, Langou J, Kurzak J, Dongarra JJ (2006) Parallel tiled QR factorization for multicore architectures. In: PPAM'07: Seventh international conference on parallel processing and applied mathematics, Gdańsk, Poland, pp 639–648
6. Buttari A, Langou J, Kurzak J, Dongarra JJ (2007) Lapack working note 191: A class of parallel tiled linear algebra algorithms for multicore architectures. Technical Report UT-CS-07-600, Electrical Engineering and Computer Sciences Department, University of Tennessee

## PMPI Tools

BERND MOHR

Forschungszentrum Jülich GmbH, Jülich, Germany

## Synonyms

MPI introspection interface; MPI monitoring interface; MPI profiling interface; [PMPI](#)

## Definition

MPI is the predominant parallel programming model used in scientific computing which is demonstrated to work on the largest computer systems available. With PMPI there exists a standardized, and therefore portable, MPI monitoring interface. Since it was part of the MPI standard from the beginning, a multitude of tools for MPI programming exist. In the MPI standard, PMPI is called the *Profiling Interface*, which is a little bit misleading as it allows to intercept every MPI call made in a parallel program and can be used for all kind of tools, e.g., tracing or validation tools, and not just profiling ones.

## Discussion

Section 8 of the MPI standard [1] describes the so-called *Profiling Interface* of MPI. The objective of this interface is to ensure that it is relatively easy for authors of MPI tools to interface their codes to MPI implementations on different machines. Since MPI is a machine independent standard with many different implementations, it is unreasonable to expect that the authors of profiling tools for MPI will have access to the source code that implements MPI on any particular machine. It was therefore necessary to provide a portable mechanism by which the implementers of such tools can

collect whatever performance information they wish without access to the underlying implementation. This is accomplished by dictating that an implementation of the MPI functions must provide a mechanism through which all of the MPI-defined functions may be accessed with a name shift. Thus, all of the MPI functions (which normally start with the prefix “MPI”) should also be accessible with the prefix “PMPI.” This can be done by using weak symbols or simply by compiling each MPI function twice but with the different name.

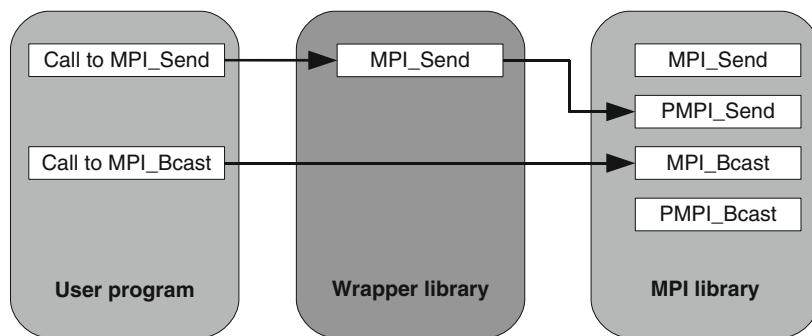
It is now possible for the implementer of an MPI tool to intercept all of the MPI calls that are made by the user program by implementing a library of so-called *wrapper functions*, which implement the same interface as the MPI function they intercept. In the wrapper function one can collect whatever information is required for the tool’s task before or after calling the underlying MPI implementation (through its name shifted entry points) to achieve the needed effects. One also has access to all parameter values passed to and returned from the MPI function.

The MPI standard also requires that the standard MPI library be built in such a way that the inclusion of MPI functions can be achieved one at a time. This is necessary so that the authors of the wrapper library need only define those MPI functions which they wish to intercept, references to any others being fulfilled by the normal MPI library. So in order to use an MPI tool, the program is first linked to the tool’s wrapper library and only then to the MPI library, as shown in Fig. 1.

## Simple Usage Example

The basic structures of any PMPI tool is a collection of wrapper routines that collect the necessary data for each MPI call. At the end of the program execution, e.g., in the `MPI_Finalize` wrapper, the collected data is potentially aggregated across all processes using MPI communication and then written to disk or presented to the user. Figure 2 shows a very simplistic version of an MPI performance tool that only counts the number of messages sent via `MPI_Send`.

A more realistic version of this wrapper library example would not only count messages, but much more performance metrics, e.g., the time spent in executing the function. Professional MPI performance tools also determine the distribution of the recorded metrics over the receiver rank or the amount of bytes sent. Of course, to get a complete recording of MPI messages all functions that send messages need to be wrapped, i.e., all variants of `MPI_Send` (`MPI_Bsend`, `MPI_Ssend`, `MPI_Rsend`, ...) and some other functions like `MPI_Start`. The latter function belongs to the group of functions implementing MPI-persistent communication. To monitor these messages, it is also necessary to track all MPI requests in the tool (implemented by wrapping all MPI calls that create, modify, or delete the opaque MPI request objects). Finally, to support all programming languages the MPI standard supports, both the C and the Fortran version of the wrappers need to be implemented. As the MPI standard defines over 300 functions, this means implementing



**PMPI Tools. Fig. 1** Linking of user program, wrapper library, and MPI library. The user program calls `MPI_Send` and `MPI_Bcast`. If the user program is first linked to the tool’s wrapper library and then to the MPI library, the tool only intercepts `MPI_Send` calls which uses `PMPI_Send` to implement the actual sending of the message. Calls to unwrapped functions (here `MPI_Bcast`) directly call the corresponding function of the MPI library

```
#include <stdio.h>
#include "mpi.h"

static int numsend = 0;

int MPI_Send(void *buf, int count, MPI_Datatype type,
 int dest, int tag, MPI_Comm comm) {
 numsend++;
 return PMPI_Send(buf, count, type, dest, tag, comm);
}

int MPI_Finalize() {
 int me;
 PMPI_Comm_rank(MPI_COMM_WORLD, &me);
 printf("%d sent %d messages.\n", me, numsend);
 return PMPI_Finalize();
}
```

**PMPI Tools. Fig. 2** Simplistic wrapper library example. The wrapper for `MPI_Send` increments a global variable which was initialized to zero. The wrapper for `MPI_Finalize` prints how many messages this particular rank had sent. As `MPI_Finalize` is executed by every rank of the program, each rank is reporting its own result. Inside the wrapper, the rank is determined via `PMPI_Comm_rank` in order to avoid invoking the corresponding wrapper function

over 600 wrapper functions; which is why, many MPI tool projects use wrapper generation tools to simplify the implementation of the wrapper library.

## Performance Measurement Tools Based on PMPI

Many MPI performance tools exist and all of them use the PMPI interface to implement the tool. Two widely accepted, portable, and scalable examples are FPMPI-2 and mpiP. They are both open-source, so that the reader can easily download them and study their implementation and usage.

**FPMPI-2** is a portable, open-source, very lightweight, and scalable MPI profiling library from Argonne National Laboratory [3]. For each MPI function, it provides the average and the maximum of the sum of metrics over all processes in a single textual output file. The provided metrics are the number of calls and the total execution time of each MPI function. FPMPI-2 has two special features that set it apart from other MPI profiling libraries: For communication functions, it records not only the amount of data transferred but also the distribution of the message sizes (by using an adaptive 32 bins histogram). Secondly, it optionally tries to determine the actual synchronization time within blocking MPI calls by replacing the actual MPI implementation with a logically

equivalent implementation using busy-wait on the user level, allowing the blocking time to be estimated.

**mpiP** is also a portable, more complete, but still scalable MPI profiling library originating from Lawrence Livermore National Laboratory but meanwhile maintained as an open-source project on sourceforge.net [4]. It provides also the number of calls, total execution time, bytes sent for each MPI function, and optionally for each MPI call-site. Captured call-paths are determined by a user-specified traceback level. Also, it provides MPI I/O statistics where applicable. The collected data is not aggregated across the processes but is collected in a scalable manner into one single output file which contains the complete data for all processes.

Besides these two MPI profiling libraries, there are many more performance tools for MPI that utilize the PMPI interface. Some other tools worth mentioning are:

- **Integrated Performance Monitoring (IPM)** [5, 6] is a portable, open-source toolkit with a focus on providing a low-overhead performance profile of the performance aspects and resource utilization in an MPI program. The level of provided detail is selectable at runtime and presented through a variety of text and web reports. Aside from overall performance, reports are available for load balance,

task topology, bottleneck detection, and message size distributions.

- There are also many vendor-specific MPI tools that are commercial products and typically only run on the system sold by the vendor. Examples here are Intel's **Trace Collector** or Cray's **CrayPat**.
- **VampirTrace** [7, 8] is a portable, open-source tool for collecting detailed traces of MPI programs in the *Open Trace Format* (OTF), which can be analyzed and visualized by the commercial Vampir trace browser. The latest version is also distributed as part of OpenMPI.
- The **TAU Parallel Performance System** [9, 10] is a very portable and versatile, open-source toolkit for performance analysis of parallel programs. Among many things, it supports profiling and tracing of MPI programs based on the PMPI interface.
- The **Scalasca** toolset [11, 12] provides call-path profiling and event tracing of MPI, OpenMP, and hybrid MPI/OpenMP programs. Its focus is on extreme scalability: In summer 2009, they reported the first successful tracing experiments done on a 2,94,912 core BlueGene/P system. It is, together with VampirTrace, the only MPI tools that does complete MPI communicator tracking enabling a detailed collective communication analysis. Finally, Scalasca is currently the only tool supporting a detailed analysis of MPI 2.0 RMA functions [13].

## Verification Tools Based on PMPI

While most of the PMPI tools measure and analyze the performance of MPI programs, there are certainly other uses for the PMPI interface. For example, it is also possible to verify the correct and portable use of the MPI standard. MPI is widely used to write parallel programs, but it does not guarantee full portability between different MPI implementations. When an application runs without any problems on one platform but crashes or gives wrong results on another platform, developers tend to blame the compiler/architecture/MPI implementation. However, in some cases, the problem is a subtle programming error in the application undetected on the first platform. Finding this bug can be a very strenuous and difficult task. The solution is to use an automated tool designed to check the correctness of MPI applications during runtime. Examples of

such violations are the introduction of irreproducibility, deadlocks, and incorrect management of resources such as communicators, groups, datatypes, etc., or the use of non-portable constructs.

A PMPI-based correctness checking tool has the advantage that it can be used with any standard-conforming MPI implementation and may thus be deployed on any development platform available to the programmer. Although high-quality MPI implementations detect some of these errors themselves, there are many cases where they do not give any warnings. For example, non-portable implementation-specific behavior is not indicated by the implementation itself, nor are checks performed that would decrease the performance too much, such as consistency checks. What is worse is that MPI implementations tolerate quite a few errors without warnings or crashing, by simply giving wrong results.

**MARMOT** [14] is one example of a PMPI-based tool designed to detect correctness and portability problems during runtime. For all tasks that require a global view, e.g., deadlock detection or the control of the execution flow, MARMOT uses an additional process, the so-called debug server. Each client registers at the debug server, which in turn gives its clients the permission for execution in a round robin way. In order to ensure that this additional debug process is transparent to the application, MARMOT maps `MPI_COMM_WORLD` to a MARMOT communicator containing only the application processes. Since all other communicators are derived from `MPI_COMM_WORLD` they automatically exclude the debug server process. Everything that can be checked locally, e.g., verification of arguments, such as tags, communicators, or ranks, is performed by the clients. Additionally, the clients and the debug server use MPI internally to transfer information. Unfortunately, this server/client architecture inflicts a bottleneck, thus affecting the scalability and performance of the tool, especially for communication-intensive applications.

**UMPIRE** [15] is a second example for a PMPI-based MPI correctness-checking tool. Like MARMOT, it performs checks on a rank-local and global level. UMPIRE uses time-out mechanism and dependency graphs to detect deadlocks. Further errors detected by UMPIRE include wrong ordering of collective communication

calls within a communicator, mismatching collective call operations, or errant writes to send buffers before non-blocking sends are completed. UMPIRE does extensive resource tracking. Consequently it is able to unearth resource leaks. For instance, applications can repeatedly create opaque objects without freeing them, leading to memory exhaustion, or there can be lost requests due to overwriting of request handles.

## Future Directions

While PMPI certainly has been successfully used for many tools, it also comes with a few limitations: One problem is that it requires users to relink their application. More importantly, the PMPI interface only supports one tool at a time. The user cannot use multiple tools in a single run of their application – a significant limitation since not only application programmers might want to perform multiple performance analyses concurrently, but also tool builders cannot use existing tools, such as an MPI profiler, to evaluate the quality of the implementation of new tools, such as a correctness checker. Further, tool builders cannot easily create tool modules since functionality cannot be split into individual PMPI-accessing layers that could be reused by other tools. Thus, it discourages code reuse during tool development.

To overcome these limitations, researchers at LLNL propose a new infrastructure called **P<sup>N</sup>MPI** [16, 17]. P<sup>N</sup>MPI allows users to dynamically load and execute one or more PMPI-based tools concurrently. This is accomplished by linking the P<sup>N</sup>MPI infrastructure into applications by default. P<sup>N</sup>MPI then transforms the wrapper libraries included in the PMPI tools into a single tool stack. Once initialized, P<sup>N</sup>MPI redirects any MPI routine executed by the application into this dynamically created stack and independently calls each tool that contains a wrapper for the routine. This eliminates the need to create a separate executable for each tool and to run each tool separately. Since P<sup>N</sup>MPI is lightweight by design, it can be included in the default build process thereby removing the need for recompilation to include or remove a tool. P<sup>N</sup>MPI also provides tool interaction functionality through services in the P<sup>N</sup>MPI core. Thus, separate modules can now implement common tool functionality to improve code reuse, modularity, and flexibility as well as tool interoperability. Possible usage scenarios for the P<sup>N</sup>MPI

infrastructure are the transparent use of tracing and profiling tools together, or efficient MPI debugging by combining deterministic reply mechanisms with MPI checker libraries like UMPIRE.

Another proposal, the so-called Universal MPI Correctness Interface (**UniMCI**) [18], specifically targets the efficient integration of performance and verification tools. Like P<sup>N</sup>MPI, it is based on the observation that using multiple tools simultaneously can help pinpoint issues faster, especially the combination of a tracing and a correctness tool can provide the history that leads to a correctness event and add further details to a detected error. Thus, it simplifies the identification of the root cause of an error. Unless P<sup>N</sup>MPI, it also provides a solution for a generic, portable coupling of any performance *host* tool with a correctness *guest* tool provided they both follow the UniMCI interface.

This interface provides two functions for each MPI call: one to analyze the initial arguments of the MPI call, called *pre* check, and one to analyze the results of the MPI call, called *post* check. All runtime MPI checkers will need an analysis of the initial MPI call arguments to detect errors in the given arguments, which is invoked with the pre check function of UniMCI. The post check is usually needed for MPI calls that create resources, e.g., `MPI_Isend`, which create new requests. MPI correctness tools have to add these new resources to their internal data structures, in order to be aware of all valid handles and their respective state. By splitting the analysis of the MPI call into two parts, it is possible to return the result of the check to the host tool before the actual MPI call is issued, which is important to guarantee that errors are handled before the application might crash. Results of checks are returned with additional interface functions that have to be issued after each check function. A simple correctness message record is used to return problems detected by the guest tool. Future versions of the interface will contain extensions and rules to handle asynchronous correctness checking tools and multi-threaded applications.

## Related Entries

- ▶ [Debugging](#)
- ▶ [Formal Methods-Based Tools for Race, Deadlock, and Other Errors](#)
- ▶ [Metrics](#)

- [MPI \(Message Passing Interface\)](#)
- [Parallel Tools Platform](#)
- [Performance Analysis Tools](#)
- [Periscope](#)
- [Profiling](#)
- [Scalasca](#)
- [TAU](#)
- [Tracing](#)
- [Vampir](#)

## Bibliography

1. Louis Turcotte (1994) Message passing interface forum: MPI: a message-passing interface standard. IJSA Special issue on MPI 8(3/4)
2. Accelerated strategic computing initiative: the ASC SMG2000 benchmark code (2001) [https://asc.llnl.gov/computing\\_resources/purple/archive/benchmarks/sm/](https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/sm/)
3. Argonne national laboratory: the FPMPI-2 MPI profiling library (2007) <http://www-unix.mcs.anl.gov/fpmi/>
4. Vetter J, Chambreau C (2007) The mpiP MPI profiling library. <http://mpip.sourceforge.net/>
5. Skinner D, Wright N, Fuerlinger K, Yellick K (2009) Allan Snavely: integrated performance monitoring. <http://ipm-hpc.sourceforge.net/>
6. Fürlinger K, Skinner D (August 2009) Capturing and visualizing event flow graphs of MPI applications. In: Workshop on productivity and performance (PROPER 2009) in conjunction with Euro-Par 2009
7. Jurenz M, Knüpfer A, Brendel R, Lieber M, Doleschal J, Mickler H, Hackenberg D, Heyde H, Müller M (2009) Vampir-Trace. <http://www.tu-dresden.de/zih/vampirtrace/>
8. Knüpfer A, Brunst H, Doleschal J, Jurenz M, Lieber M, Mickler H, Müller M, Nagel W (2008) The vampir performance analysis tool-set. In: Tools for high performance computing. Springer, Stuttgart, pp 139–155
9. Shende S, Malony A (2009) Tuning and analysis utilities. <http://tau.uoregon.edu/>
10. Shende S, Malony A (2006) The TAU parallel performance system. Int J High Perform Comput Appl 20:287–331, SAGE Publications
11. Wolf F, Mohr B, Wylie B, Geimer M (2009) Scalable performance analysis of large-scale applications. <http://www.scalasca.org/>
12. Geimer M, Wolf F, Wylie BJN, Mohr B (2009) A scalable tool architecture for diagnosing wait states in massively parallel applications. Parallel Comput 35:375–388
13. Hermanns M-A, Geimer M, Mohr B, Wolf F (2009) Scalable detection of MPI-2 remote memory access inefficiency patterns. In: Proc. of the 16th European PVM/MPI users' group meeting (EuroPVM/MPI), volume 5759 of Lecture Notes in Computer Science, Springer, Espoo, Finland, pp 31–41
14. Krammer B, Bidmon K, Müller M, Resch M (2003) MARMOT: an MPI analysis and checking tool. In: Proceedings of PARCO 2007, volume 13 of Advances in Parallel Computing, Elsevier, pp 493–500
15. Vetter J, de Supinski B (2000) Dynamic software testing of MPI applications with umpire. In: Proceedings of the 2000 ACM/IEEE conference on supercomputing (CDROM), Article No. 51
16. Schulz M, de Supinski B (2007) PnMPI tools: a whole lot greater than the sum of their parts. In: Proceedings of supercomputing, ACM/IEEE 2007 conference
17. Schulz M (2006) A flexible and dynamic infrastructure for MPI tool interoperability. International conference on parallel processing (ICPP)
18. Hilbrich T, Jurenz M, Mix H, Brunst H, Knüpfer A, Müller M, Nagel W (2010) An interface for integrated MPI correctness checking. In: Chapman B et al (eds) Advances in parallel computing. Parallel computing: from multicores and GPU's to petascale, vol 19. IOS Press, pp 693–700

## Pnetcdf

- [NetCDF I/O Library, Parallel](#)

## Point-to-Point Switch

- [Buses and Crossbars](#)

## Polaris

RUDOLF EIGENMANN  
Purdue University, West Lafayette, IN, USA

## Synonyms

- [Parallelization](#)

## Definition

Polaris is the name of a parallelizing compiler and research compiler infrastructure. Polaris performs source-to-source translation; programs written in the Fortran language are converted into restructured Fortran programs – typically annotated with directives that express parallelism. Polaris was created in the mid-1990s at the University of Illinois and was one of the most advanced freely available tools of its kind.

## Discussion

### Introduction

Polaris aimed at pushing the forefront of automatic parallelization (see ▶Parallelization, Automatic) and, at the same time, providing the research community with an infrastructure for exploring program analysis and transformation techniques. Polaris followed several earlier projects with similar goals, a key distinction being that the development of this new compiler was driven by real applications. While the benchmarks that prompted earlier parallelizing compiler research were typically small, challenging program kernels, the Polaris project was preceded and motivated by an effort to manually parallelize the most realistic application program suite available at that time – the Perfect Benchmarks [5]. This effort identified a number of beneficial transformation techniques [3], the automation of which constituted the Polaris project.

Polaris was able to improve the state of the art in automatic parallelization significantly. Earlier autoparallelizers were measured to parallelize to a significant degree only 2 of the 13 Perfect Benchmarks. By contrast, Polaris achieved success in six of them – increasing the parallelization success rate in this class of science and engineering applications from less than 20% to nearly 50%.

Polaris was originally developed at the University of Illinois from 1992 to 1995 [1], with significant extensions made at Purdue University and Texas A&M University, in later project phases. Among the examples and predecessors were several parallelizers developed at the University of Illinois and Rice University. An important contemporary was the Stanford SUIF project [6]. Among the more recent, related compiler infrastructures are Rose [4], LLVM [8], and Cetus [2].

The architecture of Polaris exhibits the classical structure of an autoparallelizer. A number of program analysis and transformation passes detect parallelism and map it to the target machine. The passes are supported by an internal program representation (IR) that represents the source code being transformed and offers a range of program manipulation functions.

### Detecting Parallelism

Polaris includes the program analysis and transformation techniques that were found to be most important

in a prior manual parallelization project [3]. At the core of any autoparallelizer is a data-dependence detection mechanism. Data dependences prevent parallelism, making dependence-removing techniques essential parts of an autoparallelizer's arsenal. To this end, Polaris includes passes for data privatization, reduction recognition, and induction variable substitution. The compiler focuses on detecting fully parallel loops, which have independent iterations and can thus be executed simultaneously by multiple processors. For the basics of the following techniques, see ▶Parallelization, Automatic.

*Data-dependence test.* Typical data-dependence tests detect whether or not two accesses to a data array in two different loop iterations could reference the same array element. The detection works well where array subscripts are linear – of the form  $a * i + b * j$ , where  $a, b$  are integer constants and  $i, j$  are index variables of enclosing loops. The Polaris project developed new dependence analysis techniques that are able to detect parallelism in the presence of symbolic and nonlinear array subscript expressions. For example, in the above expression, if  $a$  is a variable, the subscript is considered symbolic; if the term  $i^2$  appears, the subscript is nonlinear. If the compiler cannot determine the value of a symbolic term, it cannot assume it is linear. Hence, symbolic and nonlinear expressions are related. In real programs it is common for expressions, including array subscripts, to contain symbolic terms other than the loop indices. Through nonlinear, symbolic data-dependence testing, Polaris was able to parallelize several important programs that previous compilers could not.

*Privatization.* Data privatization [11] is a key enabler of improved parallelism detection. A privatization pattern can be viewed as one where a variable, say  $t$ , is being used as a temporary storage during a loop iteration. The compiler recognizes this pattern in that  $t$  is first defined (written) before used (read) in the loop iteration. By giving each iteration a separate copy of the storage space for  $t$ , accesses to  $t$  in multiple iterations do not conflict. Polaris extended the basic technique so that it could detect entire arrays that could be privatized. For example, the following loop can be parallelized after privatizing the array  $tmp$ . (the notation  $tmp(1:m)$  means “the array elements from index 1 to m”)

DO  $i=1, n$

```

tmp(1:m) = a(1:m)+b(1:m)
c(1:m) = tmp(1:m)+sqrt(
 (tmp(1:m))
ENDDO

```

Privatization gives each processor a separate instance of *tmp*. Without this transformation, each loop iteration would write to and read from the same *tmp* variable, creating a data dependence and thus inhibiting parallelization.

Implementing array privatization is substantially more complex than the basic scalar privatization technique. The compiler must determine the sections of each array that are being defined and used in a loop. If each element of an array that is being used has previously been defined in the same loop iteration, the array is privatizable. More sophisticated analysis may privatize sections of arrays that fit this pattern. To do so, the compiler analysis must be able to combine array accesses into sections. As array subscript expressions may contain symbolic terms, these operations must be supported by advanced symbolic manipulation functions, which is one of Polaris' strengths.

*Reduction recognition.* This transformation is another important enabler of parallelization. Similar to the way Polaris extended the privatization technique from scalars to arrays, it extended reduction recognition [9]. The following loop shows an example *array reduction* (sometimes referred to as irregular or histogram reduction). Different loop iterations modify different elements of the *hist* array. The pattern of modification is not important; any two loop iterations may modify the same or different array elements.

```

DO i=1,n
 val = <some computation>
 hist(tab(i)) = hist(tab(i))
 + val
ENDDO

```

Polaris recognizes array reductions by searching for loops that contain statements with the following pattern: An assignment statement has a right-hand-side expression that is the sum of the left-hand-side plus a term not involving the reduction array. A loop may have several such statements, but the reduction array must not be used in any other statement of the loop.

Because two iterations may access the same element, the compiler has to assume a possible dependence. Reduction parallelization takes advantage of the mathematical property that sum operations can be reordered (even under limited-precision computer arithmetic, reordering is usually valid, although not always). A reduction loop can be executed in parallel like this: Each processor performs the sum operations over the assigned loop iteration space on a private copy of the original reduction array (*hist*, in the above example). At the end of the loop, the local reduction arrays from all participating processors are summed into the original reduction array.

Reduction patterns are common in science and engineering applications. Array reduction parallelization was a key enabler of parallel performance in a number of important loops in the Perfect Benchmarks.

*Induction variable substitution.* Induction variable substitution eliminates dependences by replacing an arithmetic sequence by a closed-form computation. In the following loop, the induction variable *ind* forms a sequence.

```

ind = ind0
DO i=1,n
 ind = ind + k
 a(ind) = 0
ENDDO

```

In the basic form of an induction variable, the next value in the sequence is generated by adding a constant to the previous value. The term *k* could be an integer constant or a loop-invariant expression. The closed-form expression for the sequence is  $ind0+i*k$ . By substituting this expression into the array subscript,  $a(ind0+i*k)$ , the induction statement can be removed and the dependence on the previous value disappears.

Polaris extended this well-known transformation to a more general form, where *k* can be nonconstant, such as another induction variable. For example, if the expression *k* is the loop index ( $ind = ind + i$ ), the closed form becomes  $ind0+i*(i+1)/2$ .

The closed form of a generalized induction variable usually contains nonlinear terms. Parallelization in the presence of such terms requires the application of nonlinear data-dependence tests. Further complication arises when the induction variable is used after the loop. In this case, the compiler must compute and assign

the last value. In the above example, it would insert the statement  $ind = ind0 + n * k$  after the loop. Such *last-value assignment* will be correct if assignment to the induction variable is guaranteed in all iterations. Polaris makes use of symbolic program analysis techniques to make this guarantee where possible.

### Advanced Program Analysis

*Symbolic analysis.* In addition to powerful transformations, key to Polaris' performance is the availability of advanced symbolic analysis and manipulation utilities. For example, when performing generalized induction variable substitution, the compiler needs to evaluate sum expressions, such as  $\sum_1^n j = n(n+1)/2$ .

Polaris' symbolic range analysis technique is able to determine symbolic value ranges that program variables may assume during execution. The technique looks at assignment statements, loop statements, and condition statements to determine constraints on the value range of variables. After an assignment, the left-hand-side variable is known to have the newly given value or symbolic expression. Within a loop, the loop variable is guaranteed to be between the loop bounds. In the *then* clause of an *if* statement, the *if* condition is guaranteed to hold (and not hold in the *else* clause). For example, the analysis can determine that inside the following loop the inequality  $1 \leq i \leq n$  holds and the loop accesses the array elements from position 2 to  $n + 1$ .

```
DO i=1,n
 a(i+1)=0
ENDDO
```

Polaris created powerful tools for compilation passes to manipulate and reason with symbolic ranges, including comparison, intersection, and union operations. All of the described parallelization techniques depend on the availability of symbolic analysis.

*Interprocedural analysis.* Because structuring a program into subroutines is an important software engineering principle, the ability of compilers to analyze programs across procedure calls is crucial. Advanced parallelizers, such as Polaris, attempt to find parallelism in outer loops. Larger parallel regions can better amortize the overheads associated with parallel execution, as will be discussed later. However, outer loops tend to encompass subroutine calls, making the application of the described techniques difficult. Furthermore,

interprocedural analysis is important even for code sections that do not contain subroutine calls. For many optimization decisions, the compilation passes must collect information from across the program. For example, symbolic analysis will find the value ranges of program variables in the entire program and propagate them to the subroutines where needed.

The needs for interprocedural operation are different for each compiler technique. Creating specialized techniques for each optimization pass can be prohibitively expensive. Instead, Polaris includes a capability to expand subroutines inline. By default, small (by a configurable threshold) subroutines are expanded in place of their call statements. This capability obviates the need for specialized interprocedural analysis in most common cases. The drawback of subroutine inline expansion is code growth. Depending on the subroutine calling structure, a code expansion of an order of magnitude is possible. While Polaris has demonstrated that significant additional parallelism can be found this way, the needed compilation time can grow substantially. To address this issue, Polaris also included an interprocedural data access analysis framework [7] as a basis for unified interprocedural parallelism detection.

### Mapping Parallel Computation to the Target Machine

Mapping parallel computation to the target machine has two objectives. First, the parallelism uncovered by an autoparallelizer may not necessarily fit the model of parallel execution by the eventual machine platform. Additional transformations of the parallel execution or the program's data space may be needed. Second, almost all program transformations incur overheads. Privatization uses additional storage; reduction parallelization adds computation (e.g., summing up local reduction arrays); induction variable substitution creates expressions of higher strength (e.g., addition is replaced by multiplication); starting/ending parallel loops incur *fork/join* overheads (e.g., communicating to the participating processors what to do and warming up their cache). Advanced optimizing compilers make use of a performance model to estimate the overheads and to decide which transformations best to apply.

*Scheduling parallel execution.* Polaris detects loops that are dependence-free and marks them as fully parallel loops. To express parallel execution, it inserts OpenMP ([openmp.org](http://openmp.org)) directives, leaving the code generation up to the target platform's OpenMP compiler. A simple parallel loop in OpenMP looks like this.

```
!$OMP PARALLEL DO PRIVATE(t)
DO i=1,n
 t = a(i)+b(i)
 c(i) = t + t*t
ENDDO
```

The “OMP” directive expresses that the loop is to be scheduled for parallel execution and the data element  $t$  must be placed in private storage. By default, the OpenMP compiler assigns the loop iteration space to the available parallel threads (or cores) in chunks. For example, on an eight-core platform, the first core would usually execute the first  $n/8$  iterations, etc. Polaris can influence this choice via OpenMP “Schedule” clauses. For example, if the compiler detects that the amount of computation per loop iteration is irregular, it may choose a “dynamic” schedule, which maps iterations to available processors at runtime.

To determine profitability of parallel execution, Polaris applies a simple test. It estimates the size of a loop by considering the number of statements and the number of iterations. If the size can be determined and is below a configurable threshold, the compiler leaves the loop in its original, serial form. While the creation of advanced performance models is an important research area, this simple test worked well in practice.

*Data placement.* The data model is of further importance. Polaris assumes that all non-private variables are shared. All processors simply refer to the data in the same way the original, serial code does. In the above example, all processors participating in the execution of the parallel loop see the arrays  $a$ ,  $b$ , and  $c$  in the same way and have direct access. The variable  $t$  is stored in processor-private memory. Depending on the architecture, private storage may be actual processor-local memory or it may simply be a processor-private slice of the global address space.

Current multicore architectures implement this shared-memory model in hardware and are thus suitable targets for Polaris. Another important class

of parallel computers is not: Many high-performance computer platforms have a distributed memory model. Data need to be partitioned and distributed onto the different memories. Processors do not have direct access to data placed on other processors' memories; explicit communication must be inserted to read and write such data. Autoparallelizers, including Polaris, have not yet been successful in targeting this machine class. Explicit parallel programming by the software engineer is needed.

## Internal Organization

Polaris is organized into a number of program analysis and transformation passes, which make use of the functionality for program manipulation offered by the abstract internal program representation (IR). The IR represents the Fortran source program in a way that corresponds to the original code closely. The *Syntax Tree* has a structure that reflects the program's subroutines, statements, and expressions. The compiler passes see the IR in an abstract form – as a C++ class hierarchy; they perform all operations, such as finding properties of a statement or inserting an expression, via access functions.

The objects of the IR class hierarchy are organized in lists, which are traversable with several iteration methods. A typical compiler pass would begin by obtaining a list of Compilation Units (subroutines), traverse them to the desired subroutine, and obtain a reference to the list of statements of that subroutine. Next, the statement list would be traversed to the desired point, where the statement's properties and expressions could be obtained. Various “filters” are available that let pass writers iterate over objects of a specific type, only. For example, for some loop analysis pass, it may be desirable to skip from loop to loop, ignoring the statements in between them. Among the most advanced IR functions are those that allow a pass to traverse the IR until a certain statement or expression pattern is found.

A key design principle of Polaris was that these functions keep the IR consistent at all times. It would not be possible to insert a statement without properly connecting it to the surrounding scope or rename a variable without properly updating the symbol table. These bookkeeping operations are performed internally to the extent possible; they are not visible in the IR's access functions. This design offers the compiler researcher a

convenient, high-level interface. It was one reason for Polaris' popularity as a compiler infrastructure.

The implementation of the compiler consists of approximately 300,000 lines of C++ code. Forty-five percent of the code represents the IR with its access and manipulation functions; 55% implements the compilation passes.

## Uses of Polaris

The primary use of Polaris was as an autoparallelizer and compiler infrastructure in the research community. Many contributions to autoparallelization technology have been made using Polaris as an implementation and evaluation test bed. Polaris supported other applications as well. Its ability to perform source-to-source transformations made it a good platform for writing program instrumentation passes. A simple such pass might insert calls to a timing subroutine at the beginning and end of every loop, producing a tool that can create loop profiles. Other uses of Polaris included the creation of functionality that measures the maximum possible parallelism in a program, predicts the parallel performance of application programs, and creates profiles of detected dependences.

Eighteen years after it was first conceived, Polaris continues to be distributed to the research community. The last download was recorded in 2010, in the same month this entry was written.

## Challenges and Future Directions

Polaris was able to advance autoparallelization technology to the point where one in two science and engineering programs can be profitably executed in parallel on shared-memory machines. Nonnumerical programs and distributed-memory computer architectures are not yet amenable to this technology and remains an elusive research goal. In pursuing this goal, the increasing complexity of the compiler is a challenge. Learning the underlying theory and realizing its implementation is highly time-consuming for both the researchers exploring new analysis and transformation passes and the engineers developing production-strength compiler products.

One of the severe limitations of Polaris – and autoparallelization in general – is the lack of information that can be gathered from a program at compile time. Both the detection of parallelism and the mapping

to the target architecture depend on knowledge of information that may be available only from the program's input data set or the target platform. Therefore, many optimization decisions cannot be made at compile-time or can only be made by the compiler making guesses. Guesses are only legal where they do not affect the correctness of the transformed program. Therefore, compilers often must make *conservative assumptions*, which may limit the degree of optimization. Future compilers will increasingly need to merge with runtime techniques that gather information as the program executes and perform or tune optimizations dynamically. Polaris explored one aspect of this area with *runtime data-dependence techniques*. These techniques detect at runtime whether or not a loop is dependence-free and choose between serial and parallel execution [10].

## Related Entries

- ▶ [Banerjee's Dependence Test](#)
- ▶ [Code Generation](#)
- ▶ [Dependence Analysis](#)
- ▶ [GCD Test](#)
- ▶ [HPF \(High Performance Fortran\)](#)
- ▶ [Loop Nest Parallelization](#)
- ▶ [Modulo Scheduling and Loop Pipelining](#)
- ▶ [Omega Test](#)
- ▶ [Parallelization, Automatic](#)
- ▶ [Parallelization, Basic Block](#)
- ▶ [Pipelining](#)
- ▶ [Speculative Parallelization of Loops](#)
- ▶ [Run Time Parallelization](#)
- ▶ [Scheduling Algorithms](#)
- ▶ [Semantic Independence](#)
- ▶ [Speculation, Thread-Level](#)
- ▶ [Trace Scheduling](#)
- ▶ [Unimodular Transformations](#)

## Bibliography

1. Blume W, Doallo R, Eigenmann R, Grout J, Hoeflinger J, Lawrence T, Lee J, Padua D, Paek Y, Pottenger B, Rauchwerger L, Tu P (December 1996) Parallel programming with Polaris. *IEEE Comput* 29(12):78–82
2. Dave C, Bae H, Min S-J, Lee S, Eigenmann R, Midkiff S (2009) Cetus: a source-to-source compiler infrastructure for multicores. *IEEE Comput* 42(12):36–42
3. Eigenmann R, Hoeflinger J, Padua D (January 1998) On the automatic parallelization of the perfect benchmarks. *IEEE Trans Parallel Distrib Syst* 9(1):5–23

4. Quinlan DJ et al Rose compiler project. <http://www.rose-compiler.org/>
5. Berry M et al (1989) The perfect club benchmarks: effective performance evaluation of supercomputers. Int J Supercomput Appl 3(3):5–40
6. Hall MW, Anderson JM, Amarasinghe SP, Murphy BR, Liao S-W, Bugnion E, Lam MS (December 1996) Maximizing multiprocessor performance with the SUIF compiler. Computer, pp 84–89
7. Hoeflinger J, Paek Y, Yi K (2001) Unified interprocedural parallelism detection. Int J Parallel Program 29(2):185–215
8. Lattner C, Adve V (2004) LLVM: a compilation framework for lifelong program analysis & transformation. In: CGO'04: Proceedings of the international symposium on code generation and optimization. IEEE Computer Society, Washington, DC, p 75
9. Pottenger B, Eigenmann R (1995) Idiom recognition in the polaris parallelizing compiler. In: Proceedings of the 9th ACM international conference on supercomputing, Barcelona
10. Rauchwerger L, Padua D (1995) The LRPD test: speculative runtime parallelization of loops with privatization and reduction parallelization. In: PLDI'95: Proceedings of the ACM SIGPLAN 1995 conference on programming language design and implementation. ACM, New York, pp 218–232
11. Tu P, Padua D (August 1993) Automatic array privatization. In: Proceedings of the 6th workshop on languages and compilers for parallel computing, vol 768, Lecture notes in computer science, pp 500–521

## Polyhedra Scanning

► [Code Generation](#)

## Polyhedron Model

PAUL FEAUTRIER<sup>1</sup>, CHRISTIAN LENGAUER<sup>2</sup>

<sup>1</sup>CNRS École Normale Supérieure de Lyon,  
Lyon Cedex 07, France

<sup>2</sup>University of Passau, Passau, Germany

### Synonyms

Polytope model

### Definition

The polyhedron model (earlier known as the polytope model [21, 37]) is an abstract representation of a loop program as a computation graph in which questions such as program equivalence or the possibility and nature of parallel execution can be answered. The

nodes of the computation graph, each of which represents an iteration of a statement, are associated with points of  $\mathbb{Z}^n$ . These points belong to polyhedra, which are inferred from the bounds of the surrounding loops. In turn, these polyhedra can be analyzed and transformed with the help of linear programming tools. This enables the automatic exploration of the space of equivalent programs; one may even formulate an objective function (such as the minimum number of synchronization points) and ask the linear programming tool for an optimal solution. The polyhedron model has stringent applicability constraints (mainly to FOR loop programs acting on arrays), but extending its limits has been an active field of research. Beyond autoparallelization, the polyhedron model can be useful in many situations which call for a program transformation, such as in memory or performance optimization.

## Discussion

### The Basic Model

Every compiler must have representations of the source program in various stages of elaboration, as for instance by character strings, abstract syntax trees, control graphs, three-address codes, and many others. The basic component of all these representation is the statement, be it a high-level language statement or a machine instruction. Unfortunately, these representations do not meet the needs of an autoparallelizer simply because parallelism does not occur between statements, but between statement executions or *instances*. Consider:

```
for i = 0 to n-1 do
 S : a[i] = 0.0
 od
```

It makes no sense to ask whether *S* can be executed in parallel with itself; in this case, parallelism depends both on the way *S* accesses memory and on the way the loop counter *i* is updated at each iteration.

A loop program must therefore be represented as a set of instances, its *iteration domain*, here named *E*. Each instance has a distinct name and consists in the execution of the related statement or instruction, depending on the granularity of the analysis. This set is finite, in the case of a terminating program, or infinite, in the case of a reactive or streaming system.

However, this is not sufficient to specify the object program. One needs to know in which order the

instances are executed;  $E$  must be ordered by some relation  $\prec$ . If  $u, v \in E$ ,  $u \prec v$  means that  $u$  is executed before  $v$ . Since an operation cannot be executed before itself,  $\prec$  is a strict order. It is easy to see that the usual control constructs (sequences, loops, conditionals, jumps) are compact ways of defining  $\prec$ . It is also easy to see that, in a sequential program, two arbitrary instances are always ordered: one says that, in this case,  $\prec$  is a total order. Consideration of an elementary parallel program (in OpenMP notation):

```
#pragma omp parallel sections
 S1
#pragma omp section
 S2
#pragma omp end parallel sections
```

shows that  $S_1$  may be executed before or after or simultaneously with  $S_2$ , depending on the available resources (processors) and the overall state of the target system. In that case, neither  $S_1 \prec S_2$  nor  $S_2 \prec S_1$  are true: one says that  $\prec$  is a partial order. As an extreme case, an embarrassingly parallel program, in which instances can be executed in any order, has the empty execution order. Therefore, one may say that parallelization results in replacing the total execution order of a sequential program by a partial one, under the constraint that the outcome of the program is not modified. This in turn raises the following question: *Under which conditions are two programs with the same iteration domain but different execution orders equivalent?*

Since program equivalence is undecidable in general, one must be content with conservative answers, i.e., with sufficient but not necessary equivalence conditions. The usual approach is based on the concept of *dependences* (see also the [Dependences](#) entry in this encyclopedia). Assuming that, given the name of an instance  $u$ , one can characterize the sets (or supersets) of read and written memory cells,  $\mathcal{R}(u)$  and  $\mathcal{W}(u)$ ,  $u$  and  $v$  are in dependence, written  $u \delta v$ , if both access some memory cell shared by them and at least one of them modifies it. In symbols:  $u \delta v$  if at least one of the sets  $\mathcal{R}(u) \cap \mathcal{W}(v)$ ,  $\mathcal{W}(u) \cap \mathcal{R}(v)$  or  $\mathcal{W}(u) \cap \mathcal{W}(v)$  is not empty. The concept of a dependence was first formulated by Bernstein [8]. One can prove that two programs are equivalent if dependent instances are executed in the same order in both.

► **Aside.** Proving equivalence starts by showing that, under Bernstein's conditions, two independent consecutive instances can be interchanged without modifying the final state of memory. In the case of a terminating program, this is done by specifying a succession of interchanges that convert one order into the other without changing the final result. The proof is more complex for nonterminating programs and depends on a fairness hypothesis, namely, that every instance is to be executed eventually. One can then prove that the succession of values assigned to each variable – its history – is the same for both programs. One first shows that the succession of assignments to a given variable is the same for both programs since they are in dependence, and, as a consequence, that the assigned values are the same, provided all instances are deterministic, i.e., return the same value when executed with the same arguments (see also the [Bernstein's Conditions](#) in this encyclopedia).

To construct a parallel program, one wants to remove all orderings between independent instances, i.e., construct the relation  $\delta \cap \prec$ , and take its transitive closure. This execution order may be too complex to be represented by the available parallel constructs, like the parallel sections or the parallel loops of OpenMP. In this case, one has to trade some parallelism for a more compact program.

It remains to explain how to name instances, how to specify the index domain of a program and its execution order, and how to compute dependences. There are many possibilities, but most of them ask for the resolution of undecidable problems, which is unsuitable for a compiler. In the polyhedron model, sets are represented as *polyhedra* in  $\mathbb{Z}^n$ , i.e., sets of (integer) solutions of systems of affine inequalities (inequalities of the form  $Ax \leq b$ , where  $A$  is a constant matrix,  $x$  a variable vector, and  $b$  a constant vector). It so happens that these sets are the subject of a well-developed theory, (integer) linear programming [43], and that all the necessary tools have efficient implementations.

The crucial observation is that the iterations of a regular loop (a Fortran DO loop, or a Pascal FOR loop, or restricted forms of C, C++ and Java FOR loops) are represented by a segment (which is a one-dimensional polyhedron), and that the iterations of a regular loop nest are represented by a polyhedron

with as many dimensions as the nest has loops. Consider, for instance, the first statement of the loop program in Fig. 1a. It is enclosed in two loops. The instances it generates can be named by stating the values of  $i$  and  $j$ , and the iteration domain is defined by the constraints:

$$1 \leq i \leq n, \quad 1 \leq j \leq i + m$$

which are affine and therefore define a polyhedron. In the same way, the iteration domain of the second statement is  $1 \leq i \leq n$ . For better readability, the loop counters are usually arranged from outside inward in a vector, the *iteration vector* of the instance.

Observe that, in this representation,  $n$  and  $m$  are parameters, and that the size of the representation is independent of their values. Also, the upper bound of

the loop on  $j$  is not constant: iteration domains are not limited to parallelepipeds.

The iteration domain of the program is the disjoint union of these two polyhedra, as depicted in Fig. 1b with dependences. (The dependences and the right side of the figure are discussed below.) To distinguish the several components of the union, one can use statement labels,

as in:

$$E = \{\langle S_1, i, j \rangle \mid 1 \leq i \leq n, 1 \leq j \leq i + m\} \cup \{\langle S_2, i \rangle \mid 1 \leq i \leq n\}$$

The execution order can be deduced from two observations:

- In a program without control constructs, the execution order is the *textual order*. Let  $u <_{\text{txt}} v$  be true if  $u$  occurs before  $v$  in the program text.

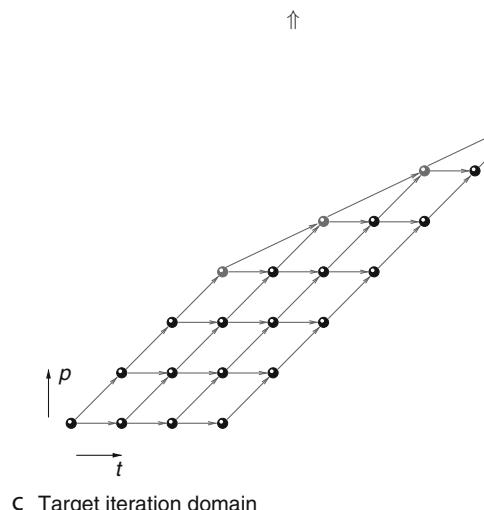
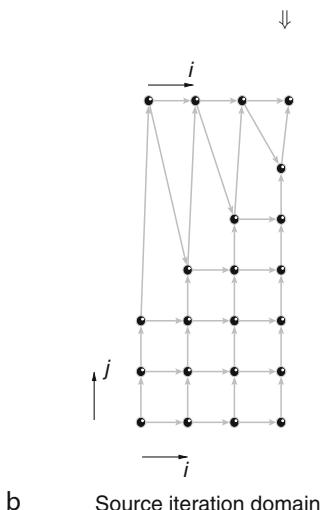
```

for t = 0 to m+2*n-1 do
 parfor p = max(0,t-n+1) to min(t,⌈(t+m)/2⌉) do
 if 2*p = t+m+1 then
 S2 : A(p-m,p+1) = A(p-m-1,p) + A(p-m,p)
 else
 S1 : A(t-p+1,p+1) = A(t-p,p+1) + A(t-p+1,p)
 fi
 od
od

```

a                  Source loop nest

d                  Target loop nest



Polyhedron Model. Fig. 1 Loop nest transformation in the basic polyhedron model

- Loop iterations are executed according to the *lexicographic order* of the iteration vectors. Let  $x <_{\text{lex}} y$  be true if the vector  $x$  is lexicographically less than  $y$ .

In more complex cases, these two observations may be combined to give:

$$\langle R, x \rangle < \langle S, y \rangle \equiv x[1:N] <_{\text{lex}} y[1:N] \vee \\ (x[1:N] = y[1:N] \wedge R <_{\text{txt}} S),$$

where  $R$  and  $S$  are two statements,  $x$  and  $y$  their iteration vectors,  $N$  is the number of loops which encloses both  $R$  and  $S$ , and  $x[1:N]$  is the vector  $x$  restricted to its  $N$  first components. Returning to Fig. 1, one has:

$$\langle S_1, i, j \rangle < \langle S_2, i' \rangle \equiv i < i' \vee (i = i' \wedge \text{true}),$$

which simplifies to  $\langle S_1, i, j \rangle < \langle S_2, i' \rangle \equiv i \leq i'$ .

The assumption behind dependence analysis is that the sets  $\mathcal{R}(u)$  and  $\mathcal{W}(u)$  above only depend on the name of the instance  $u$ . This is obviously not true in general. In the polyhedron model, one assumes that all accesses are to scalars and arrays, and that, in the latter case, subscripts are known functions of the surrounding loop counters. One usually also assumes that there is no *aliasing* – two arrays with different names do not overlap – and that subscripts are always within the array bounds. Techniques for detecting and correcting violations of these assumptions are beyond the scope of this entry. With these assumptions, two instances  $\langle R, x \rangle$  and  $\langle S, y \rangle$  are in dependence if both access the same array  $A$  of dimension  $d_A$ , and if the *subscript equations*

$$f_R(x) = f_S(y)$$

have solutions within the iteration domains of  $R$  and  $S$ . Here,  $f_R$  and  $f_S$  are the respective *subscript functions* of  $A$  in  $R$  and  $S$ . Solving such equations is easy only if each subscript is an affine function of the iteration vector:

$$f_R(x) = F_R x + g_R,$$

where  $F_R$  is a matrix of dimension  $d_A \times d_R$ , with  $d_R$  being the number of loops surrounding  $R$ , and  $g_R$  is a vector of dimension  $d_A$ . One may associate with each candidate dependence a system of constraints by gathering the subscript equations, the constraints which define the iteration domains of  $R$  and  $S$ , and the sequencing predicate above. All of these constraints are affine, with the exception of the sequencing predicate which is a disjunction of affine constraints. Each disjunct can be

tested for solutions, either by ad hoc conservative methods – see the ►Banerjee's Dependence Test entry in this encyclopedia – or by linear programming algorithms – see the ►Dependences entry.

In summary, a program can be handled in the polyhedron model – and is then called a *regular* or *static control program* – if its only control constructs are (also called regular) loops with affine bounds and its data structures are either scalars or arrays with affine subscripts in the surrounding loop counters. It should be noted that these restrictions must not be taken syntactically but semantically. For instance, in the program:

```
i = 0; k = 0;
while i < n do
 a[k] = 0.0;
 i = i + 1;
 k = k + 3
od
```

the loop is in fact regular with counter  $i$ , and the subscript of  $a$  is really  $3i$ , which is affine. There are many classical techniques – here, induction variable detection – for transforming such constructs into a more “polyhedron-friendly” form.

Regular programs are mainly found in scientific computing, linear algebra, and signal processing, where unbounded iteration domains are frequent. Perhaps more surprisingly, many variants of the Smith and Waterman algorithm [44], which is the basic tool for genetic sequence analysis, are regular and can be optimized with polyhedral tools [30]. Also, while large programs rarely fit in the model, it is often possible to extract regular kernels and to process them in isolation.

## Transformations

The main devices for program optimization in the polyhedron model are coordinate transformations of the iteration domain.

## An Example

Consider Fig. 1 as an illustration of the use of transformations. Figure 1a presents a sequential source program with two nested loops. The loop nest is *imperfect*: not all statements belong to the innermost loop body.

Figure 1b depicts the iteration domain of the source program, as explained in the previous section. The arrows represent dependences and impose a partial

order on the loop steps. Apart from these ordering constraints, steps can be executed in any order or in parallel.

In the source iteration domain, parallelism is in some sense hidden. The loop on  $j$  is sequential since the value stored in  $A(i,j)$  at iteration  $j$  is used as  $A(i,j-1)$  in the next iteration. The same is true for the loop on  $i$ . However, parallelism can be made visible by applying a skewing transformation as in Fig. 1c. For a given value of  $t$ , there are no dependences between iterations of the  $p$  loop, which is therefore parallel.

The required transformation can be viewed as a change of coordinates or a renaming:

$$S_1 : \begin{pmatrix} t \\ p \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -2 \\ -1 \end{pmatrix}$$

$$S_2 : \begin{pmatrix} t \\ p \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix} \begin{pmatrix} i \end{pmatrix} + \begin{pmatrix} m-1 \\ m \end{pmatrix}$$

Observe that the transformation for  $S_1$  has the non-singular matrix  $\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$  and, hence, is bijective. Furthermore, the determinant of this matrix is 1 (the matrix is *unimodular*), which means that the transformation is bijective in the integers.

A target loop nest which corresponds to the target iteration domain is depicted in Fig. 1d. The issue of target code generation is addressed later. For now, just note that the target loop nest is much more complex than the source loop nest, and that it would be cumbersome and error-prone to derive it manually. On the other hand, the fact that both transformation matrices are unimodular simplifies the target code: both loops have unit stride.

### The Search for a Transformation

The fundamental constraint on a transformation in the polyhedron model is *affinity*. As explained before, each row of the transformation matrix corresponds to one axis of the target coordinate system. Each axis represents either a sequential loop or a parallel loop. Iterations of a sequential loop are executed successively; hence, the loop counter can be interpreted as (logical)

time. Iterations of a parallel loop are executed simultaneously (available resources permitting) by different processors; the values of their loop counters correspond to processor names. Finding the coefficients for the sequential axes constitutes a problem of *scheduling*, finding the coefficients for the parallel axes one of *placement* or *allocation*. Different methods exist for solving these two problems.

The order in which sequential and parallel loops are nested is important. One can show that it is always possible to move the parallel loops deeper inside the loop nest, which generates lock-step parallelism, suitable for vector or VLIW processors. For less tightly coupled parallelism, suitable for multicores or message-passing architectures, one would like to move the parallel loops farther out, but this is not always possible.

### Scheduling

A schedule maps each instance in the iteration domain to a logical date. In contrast to what happens in (►task graph scheduling), the number of instances is large, or unknown at compile time, or even infinite, so that it is impossible to tabulate this mapping. The schedule must be a closed-form function of the iteration vector; it will become clear presently that its determination is easy only if restricted to affine functions.

Let  $\theta_R(i)$  be the schedule of instance  $\langle R, i \rangle$ . Since the source of a dependence must be executed before its destination, the schedule must satisfy the following *causality constraint*:

$$\forall i, j : \langle R, i \rangle \delta \langle S, j \rangle \Rightarrow \theta_R(i) < \theta_S(j).$$

There are as many such constraints as there are dependences in the program. The unknowns are the coefficients of  $\theta_R$  and  $\theta_S$ . The first step in the solution is the elimination of the quantifiers on  $i$  and  $j$ . There are general methods of quantifier elimination [38] but, due to the affinity of the constraints in the polyhedron model, more efficient methods can be applied. In fact, the form of the causality constraint above asserts that the affine delay  $\theta_S(j) - \theta_R(i)$  must be positive inside the *dependence polyhedron*  $\{i, j \mid \langle R, i \rangle \delta \langle S, j \rangle\}$ . To this end, it is necessary and sufficient that the delay be positive at the vertices of the dependence polyhedron, or that it be

an affine positive combination of the dependence constraints (Farkas lemma). The result of quantifier elimination is a linear system of inequalities which can be solved by any linear programming tool.

This system of constraints may not be *feasible*, i.e., it may have no solution. This means simply that no linear-time parallel execution exists for the source program. The solution is to construct a multidimensional schedule. In the target loop nest, there will be as many sequential loops as the schedule has dimensions.

More information on scheduling can be found in the ►[Scheduling Algorithms](#) entry of this encyclopedia.

## Placement

A placement maps each instance to a (virtual) processor number. Again, this mapping must be in the form of a closed affine function. In contrast to scheduling, there is no legality constraint for placements: any placement is valid, but may be inefficient.

For each dependence between instances that are assigned to distinct processors, one must generate a communication or a synchronization, depending on whether the target architecture has distributed or shared memory. These are costly operations, which must be kept at a minimum. Hence, the aim of a placement algorithm is to find a function:

$$\pi : E \rightarrow [0, P]$$

where  $P$  is the number of processors, such that the size of the set:

$$\mathcal{C} = \{u, v \in E \mid u \delta v, \pi(u) \neq \pi(v)\}$$

is minimal. Since counting integer points inside a polyhedron is difficult, one usually uses the following heuristics: try to “cut” as many dependences as possible. A dependence from statement  $R$  to  $S$  can be *cut* if the following constraint holds:

$$\langle R, i \rangle \delta \langle S, j \rangle \Rightarrow \pi_R(i) = \pi_S(j).$$

This condition can be transformed into a system of homogeneous linear equations for the coefficients of  $\pi$ . The problem is that, in most cases, if one tries to satisfy all the cutting constraints, the only solution is  $\pi(u) = 0$ ,

which corresponds to execution on only one processor: this, indeed, results in the minimal number of synchronizations (namely zero)! A possible way out is to solve the cutting constraints one at time, in order of decreasing size of the dependence polyhedron, and to stop just before generating the trivial solution. The uncut dependences induce synchronization operations. If all dependences can be cut, the program has *communication-free parallelism* and can be rewritten with one or more outermost parallel loops.

In the special case of a perfect loop nest with uniform dependences, one may approximate the dependence graph by the translations of the lattice generated by the dependence vectors. If the determinant of this lattice is larger than 1, the program can be split into as many independent parts [17].

Lastly, instead of assigning a processor number to each instance, one may assign all iterations of one statement to the same processor [35, 47]. This results in the construction of a Kahn process network [33].

## Code Generation

In the polyhedron model, a transformation of the source iteration domain which optimizes some objective function can be found automatically. The highest execution speed (i.e., the minimum number of steps to be executed in sequence) may be the first thing that comes to mind, but many other functions are possible.

Unfortunately, it is not trivial to generate efficient target code from the optimal solution in the model. There are several factors that can degrade performance seriously. The enumeration of the points in the target iteration domain involves tests for the lower and upper border. If the code is not chosen wisely, these tests will often degrade scalability. For example, in Fig. 1, a maximum and a minimum is involved. The example of Fig. 1 also shows that additional control (the IF statement) may be introduced, which degrades performance. Of course, synchronizations and communications can also degrade performance seriously.

For details on code generation in the polyhedron model, see the ►[Code Generation](#).

## Extensions

The following extensions have successively been made to the basic polyhedron model.

## WHILE Loops

The presence of a WHILE loop in the loop nest turns the iteration domain from a finite set (a polytope) into an infinite set (a polyhedron). If the control dependence that the termination test of the loop imposes is being respected, the iteration must necessarily be sequential. However, the steps of a WHILE loop in a nest with further (FOR or WHILE) loops may be distributed in space. There have been two approaches to the parallelization of WHILE loops.

The *conservative approach* [22, 25] respects the control dependence. One challenge here is the discovery of global termination. The *speculative approach* [14] does not respect the control dependence. Thus, several loop steps may be executed in parallel if there is no other dependence between them. The price paid is the need for storage of intermediate results, in case a rollback needs to be done when the point of termination has been discovered but further steps have already been executed. In some cases, overshooting the termination point does not jeopardize the correctness of the program and no rollback is needed. Discovering this property is beyond the capability of present compilers.

## Conditional Statements

The basic model permits only assignment statements in the loop body. The challenge of conditionals is that a dependence may hold only for certain executions, i.e., not for all branches. A static analysis can only reveal the union of these dependences [13].

## Iteration Domain Splitting

In some cases, the schedule can be improved by orders of magnitude if one splits the iteration domain in appropriate places [24]. One example is depicted in Fig. 2. With the best affine schedule of  $[i/2]$ , each parallel

step contains two loop iterations, i.e., the execution is sped up by a factor of 2. (The reason is that the shortest dependence has length 2.) The domain split on the right yields two partitions, each without dependences between its iterations. Thus, all iterations of the upper loop (enumerating the left partition) can be executed in a first parallel step, and the iterations of the lower loop (enumerating the right partition) in a second one, for a speedup of  $n/2$ .

## Tiling

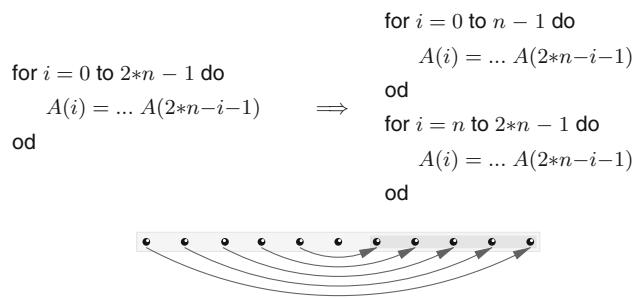
The technique of domain splitting has a further, larger significance. The polyhedron model is prone to yielding very fine-grained parallelism. To coarsen the grain when not enough processors are available, one partitions (parts of) the iteration domain in equally sized and shaped *tiles*. Each tile covers a set of iterations and the points in a tile are enumerated in time rather than in space, i.e., the iteration over a tile is resequentialized.

One can tile the source iteration domain or the target iteration domain. In the latter case, one can tile space and also time. Tiling time corresponds to adding hands to a clock and has the effect of coarsening the grain of processor communications. The habilitation thesis of Martin Griebel [23] offers a comprehensive treatment of this topic and an extensive bibliography. See also the ►Tiling entry of this encyclopedia.

P

## Treatment of Expressions

In the basic model, expressions are considered atomic. There is an extension of the polyhedron model to the parallelization of the evaluation of expressions [18]. It also permits the identification of common subexpressions and provides a means to choose automatically the suitable point in time and the suitable place at which to



Polyhedron Model. Fig. 2 Iteration domain splitting

evaluate it just once. Its value is then communicated to other places.

### Relaxations of Affinity

The requirement of affinity enters everywhere in the polyhedron model: in the loop bounds, in the array index expressions, in the transformations. Quickly, after the polyhedron model had been developed, the desire arose to transcend affinity in places. Iteration domain splitting is one example.

Lately, a more encompassing effort has been made to leave affinity behind. One circumstance that breaks the affinity of index expressions is that the so-called structure parameters (e.g., variables  $n$  and  $m$  in the loops of Figs. 1 and 2) enter multiplicatively as unevaluated variables, not as constants. For example, when a two-dimensional array is linearized, array subscripts are of the form  $ni + j$ , with  $i, j$  being the loop iterators. As a consequence, subscript equations are nonlinear in the structure parameters, too. An algorithm for computing the solutions of equation systems with exactly one such structure parameter exists [29].

In transformations and code generation, nonlinear structure parameters, as in expressions  $ni, n^2 i$ , or  $n m i$ , can be handled by generalizing existing algorithms (for the case without nonlinear parameters) using quantifier elimination [28]. Code generation can even be generalized to handle nonlinear loop indices, as in  $n i^2, n^2 i^2$ , or  $ij$ . To this end, cylindrical algebraic decomposition (CAD) [27], which corresponds to Fourier–Motzkin elimination in the basic model, is used for computing loops nests which enumerate the points in the transformed domains efficiently. This extends the frontier of code generation to arbitrary polynomial loop bounds.

## Applications Other than Loop Parallelization

### Array Expansion

It is easy to see that, if a loop modifies a scalar, there is a dependence between any two iterations, and the loop must remain sequential. When the modification occurs early in the loop body, before any use, the dependence can be removed by expanding the scalar to a new array, with the loop counter as its subscript. This idea can be extended to all cases in which a memory cell – be it a

scalar or part of an array – is modified more than once. The transformation proceeds in two steps:

- Replace the left side of each assignment by a fresh array, subscripted by the counters of all enclosing loops.
- Inspect all the right sides and replace each reference by its source [20].

The source of a use is the latest modification that precedes the use in the sequential execution order. It can be computed by *parametric integer programming*. The result of this transformation is a program in *dynamic single-assignment form*. Each memory cell is written to just once in the course of a program execution. As a consequence, the sets  $\mathcal{W}(u) \cap \mathcal{W}(v)$  are always empty: the transformed program has far fewer dependences and, occasionally, much more parallelism than the original.

### Array Shrinking

A consequence of the previous transformation is a large increase in the memory footprint of the program. In many cases, the same degree of parallelism can be achieved with less expansion, or the target architecture cannot exploit all parallelism there is, and some of the parallel loops have to be sequentialized. Another situation, in a purely sequential context, is when a careless programmer has used more memory than strictly necessary to implement an algorithm.

The aim of *array shrinking* is to detect these situations and to reduce the memory needs by inserting modulo operators in subscripts. Suppose, for instance, that in the following code:

```
for i = 0 to n-1 do
 a[i] = ... ;
od
```

one replaces  $a[i]$  by  $a[i \bmod 16]$ . The dimension of  $a$ , which is  $n$  in the first version, is reduced to 16 in the second version. Of course, this means that the value stored in  $a[i]$  is destroyed after 16 iterations of the loop. This transformation may change the outcome of the program, unless one can prove that the *lifetime* of  $a[i]$  does not exceed 16 iterations.

Finding an automatic solution to this problem has been the subject of much work since 1990 (Darte [16] offers a good discussion). The proposed solution is to construct an interference polyhedron for the elements

of a fixed array and to cover it by a maximally tight lattice such that only the lattice origin falls inside the polyhedron. The basis vectors of the lattice are taken as coordinate axes of the reduced array, and their lengths are related to the modulus of the new subscripts.

### Communication Generation

When constructing programs for distributed memory architectures, be it with data distribution directives in languages like High-Performance Fortran (HPF) or under the direction of a placement function, one has to generate communication code. It so happens that this is also a problem of polyhedron scanning. It can be solved by the same techniques and the same tools that are used for code generation.

### Locality Enhancement

Most modern processors have caches: small but fast memories that retain a copy of recently accessed memory cells. A program has locality if memory accesses are clustered such that there is a high likelihood of finding a copy of the needed information in cache rather than in main memory. Improving the locality of a program is highly beneficial for performance since caches are usually accessed in one cycle while memory latency may range from ten to a hundred cycles.

Since the cache controller returns old copies to memory in order to find room for new ones, locality is enhanced by changing the execution order such that the *reuse distance* between successive accesses to the same cell is minimal. This can be achieved, for instance, by moving all such accesses to the innermost loop of the program [49].

Another approach consists of dividing a program into *chunks* whose memory footprints are smaller than the cache size. Conceptually, the program is executed by filling the cache with the necessary data for one chunk, executing the chunk without any cache miss, and emptying the cache for the next chunk. One can show that the memory traffic will be minimal if each datum belongs to the footprint of only one chunk. The construction of chunks is somewhat similar to scheduling [7]. It is enough to have asymptotic estimates of the footprint sizes. One advantage of this method is that it can be adapted easily to the management of *scratchpad memories*, software-controlled caches as can be found in embedded processors.

### Dynamic Optimization

Dynamic optimization resulted from the observation that modern processors and compilers are so complex that building a realistic performance estimator is nearly impossible. The only way of evaluating the quality of a transformed program is to run it and take measurements.

In the polyhedron model, one can define the polyhedron of all legal schedules (see the previous section on ►Scheduling). Usually, one selects one schedule in this polyhedron according to some simple objective function. Another possibility is to generate one program for each legal schedule, measure its performance, and retain the best one. Experience shows that, in many cases, the best program is unexpected, the proof of its legality is not obvious, and the reasons for its efficiency are difficult to fathom. As soon as the source program has more than a few statements, the size of the polyhedron of legal schedules explodes: sophisticated techniques, including genetic algorithms and machine learning are needed to restrict the exploration to “interesting” solutions [39, 40].

### Tools

There is a variety of tools which support several phases in the polyhedral parallelization process.

### Mathematical Support

PIP [19] is an all integer implementation of the Simplex algorithm, augmented with Gomory cuts for integer programming [43]. The most interesting feature of PIP is that it can solve parametric problems, i.e., find the lexicographically minimal  $x$  such that

$$Ax \leq By + c$$

as a function of  $y$ .

Omega [41] is an extension of the Fourier–Motzkin elimination method to the case of integer variables. It has been extended into a full-fledged tool for the manipulation of Presburger formulas (logical formulas in which the atoms are affine constraints on integer variables).

There are many so-called polyhedral libraries; the oldest one is the PolyLib [11]. The core of these libraries is a tool for converting a system of affine constraints into the vertices of the polyhedron it defines, and back. The PolyLib also includes a tool for counting the number

of integer points inside a parametric polyhedron, the result being an Ehrhart polynomial [10]. More recent implementations of these tools, occasionally using different algorithms, are the Parma Polyhedral Library [2], the Integer Set Library [46], the Barvinok Library [48], and the Polka Library [32]. This list is probably not exhaustive.

### Code Generation

CLOOG [6] takes as input the description of an iteration domain, in the form of a disjoint union of polyhedra, and generates an efficient loop nest that scans all the points in the iteration domain in the order given by a set of scattering functions, which can be schedules, placements, tiling functions, and more. For a detailed description of CLOOG, see the ►Code Generation entry in this encyclopedia.

### Full-Fledged Loop Restructurers

LooPo [26] was the first polyhedral loop restructurer. Work on it was started at the University of Passau in 1994, and it was developed in steps over the years and is still being extended. LooPo is meant to be a research platform for trying out and comparing different methods and techniques based on the polyhedron model. It offers a number of schedulers and allocators and generates code for shared-memory and distributed memory architectures. All of the extensions mentioned above have been implemented and almost all are being maintained.

Pluto [9] was developed at Ohio State University. Its main objective is to use placement functions to improve locality and to integrate tiling into the polyhedron model. Its target architectures are multicores and graphical processing units (GPUs).

GRAPHITE [45] is an extension of the GCC compiler suite whose ultimate aim is to apply polyhedral optimization and parallelization techniques, where possible, to run-of-the-mill programs. GRAPHITE looks for static control parts (SCoPs) in the GCC intermediate representation, generates their polyhedral representation, applies transformations, and generates target code using CLOOG. At the time of writing, the set of available transformations is still rudimentary, but is supposed to grow.

### Related Entries

- Banerjee's Dependence Test
- Code Generation
- Dependence Abstractions
- Dependence Analysis
- HPF (High Performance Fortran)
- Loop Nest Parallelization
- OpenMP
- Scheduling Algorithms
- Speculative Parallelization of Loops
- Task Graph Scheduling
- Tiling

### Bibliographic Notes and Further Reading

The development of the polytope model was driven by two nearly disjoint communities. Hardware architects wanted to take a set of recurrence equations, expressing, for instance, a signal transformation, and derive a parallel processor array from it. Compiler designers wanted to take a sequential loop nest and derive parallel loop code from it.

One can view the seed of the model for architecture in the seminal paper by Karp, Miller, and Winograd on analyzing recurrence equations [34] and the seed for software in the seminal paper by Lamport on Fortran DO loop parallelization [36]. Lamport used hyperplanes (the slices in the polyhedron that make up the parallel steps), instead of polyhedra. In the early 1980s, Quinton drafted the components of the polyhedron model [42], still in the hardware context (at that time: systolic arrays).

The two communities met around the end of the 1980s at various workshops and conferences, notably the International Conference on Supercomputing and CONPAR and PARLE, the predecessors of the Euro-Par series. Two developments made the polyhedron model ready for compilers: parametric integer programming, worked out by Feautrier [19], which is used for dependence analysis, scheduling and code generation, and seminal work on code generation by Irigoin et al. [1, 31]. Finally, Lengauer [37] gave the model its name.

The 1990s saw the further development of the theory underlying the model's methods, particularly for scheduling, placement, and tiling. Extensions and

applications other than loop parallelization came mainly in the latter part of the 1990s and in the following decade.

A number of textbooks focus on polyhedral methods. There is the three-part series of Banerjee [3–5], a book on tiling by Xue [50], and a comprehensive book on scheduling by Darte et al. [15]. Collard [12] applies the model to the optimization of loop nests for sequential as well as parallel execution and studies a similar model for recursive programs.

In the past several years, the polyhedron model has become more mainstream. The seed of this development was an advance in code generation methods [6]. With the GCC community taking an interest, it is to be expected that polyhedral methods will increasingly find their way into production compilers.

## Bibliography

1. Ancourt C, Irigoin F (1991) Scanning polyhedra with DO loops. In: Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), ACM, pp 39–50
2. Bagnara R, Hill PM, Zaffanella E (2008) The Parma polyhedra library: toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci Comput Program* 72(1–2):3–21, <http://www.cs.unipr.it/ppl>
3. Banerjee U (1993) Loop transformations for restructuring compilers: the foundations. Series on Loop Transformations for Restructuring Compilers. Kluwer, Norwell
4. Banerjee U (1994) Loop parallelization. Series on Loop Transformations for Restructuring Compilers. Kluwer, Norwell
5. Banerjee U (1997) Dependence analysis. Series on Loop Transformations for Restructuring Compilers. Kluwer, Norwell
6. Bastoul C (2004) Code generation in the polyhedral model is easier than you think. In: Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT), IEEE Computer Society Press, pp 7–16, <http://www.cloog.org/>
7. Bastoul C, Feautrier P (2003) Improving data locality by chunking. In: Compiler Construction (CC), Lecture Notes in Computer Science, vol 2622. Springer, Berlin, pp 320–335
8. Bernstein AJ (1966) Analysis of programs for parallel processing. In: IEEE Transactions on Electronic Computers, EC-15:757–762
9. Bondhugula U, Hartono A, Ramanujam J, Sadayappan P (2008) A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Notices*, 43(6):101–113, 2008. <http://pluto-compiler.sourceforge.net/>
10. Clauss P (1996) Counting solutions to linear and non-linear constraints through Ehrhart polynomials. In: Proceedings of the ACM/IEEE Conference on Supercomputing, ACM, pp 278–285
11. Clauss P, Loechner V (1998) Parametric analysis of polyhedral iteration spaces, extended version. *J VLSI Signal Process* 19(2):179–194, <http://icps.u-strasbg.fr/polylib/>
12. Collard J-F (2003) Reasoning about program transformations – imperative programming and flow of data. Springer, Berlin
13. Collard J-F, Griebl M (1999) A precise fixpoint reaching definition analysis for arrays. In: Carter L, Ferrante J (eds) Languages and Compilers for Parallel Computing (LCPC), Lecture Notes in Computer Science, vol 1863, Springer, Berlin, pp 286–302
14. Collard J-F (1995) Automatic parallelization of while-loops using speculative execution. *Int J Parallel Program* 23(2):191–219
15. Darte A, Robert Y, Vivien F (2000) Scheduling and automatic parallelization. Birkhäuser, Boston
16. Darte A, Schreiber R, Villard G (2005) Lattice-based memory allocation. *IEEE Transaction on Computers* TC-54(10):1242–1257
17. D'Hollander EH (1992) Partitioning and labeling of loops by unimodular transformations. *IEEE Trans Parallel Distrib Syst* 3(4):465–476
18. Faber P (2007) Code optimization in the polyhedron model – improving the efficiency of parallel loop nests. PhD thesis, Department of Informatics and Mathematics, University of Passau, 2007. <http://www.fim.unipassau.de/cl/publications/docs/Faber07.pdf>
19. Feautrier P (1988) Parametric integer programming. *Oper Res* 22(3):243–268, <http://www.piplib.org>
20. Feautrier P (1991) Dataflow analysis of scalar and array references. *Parallel Program* 20(1):23–53
21. Feautrier P (1996) Automatic parallelization in the polytope model. In: Perrin G-R, Darte A (eds) The data parallel programming model. Lecture Notes in Computer Science, vol 1132, Springer, Berlin, pp 79–103
22. Griebl M (1997) The mechanical parallelization of loop nests containing WHILE loops. PhD thesis, Department of Mathematics and Informatics, University of Passau, January 1997. <http://www.fim.unipassau.de/cl/publications/docs/Gri96.pdf>
23. Griebl M (2004) Automatic parallelization of loop programs for distributed memory architectures. Habilitation thesis, Department of Informatics and Mathematics, University of Passau, June 2004. <http://www.fim.unipassau.de/cl/publications/docs/Gri04.pdf>
24. Griebl M, Feautrier P, Lengauer C (2000) Index set splitting. *Int J Parallel Process* 28(6):607–631 Special Issue on the International Conference on Parallel Architectures and Compilation Techniques (PACT'99)
25. Griebl M, Lengauer C (1994) On the space-time mapping of WHILE loops. *Parallel Process Lett* 4(3):221–232
26. Griebl M, Lengauer C (1997) The loop parallelizer LooPo – announcement. In: Sehr D (ed) Languages and Compilers for Parallel Computing (LCPC). Lecture Notes in Computer Science, vol 1239, pp 603–604. Springer, Berlin, <http://www.infosun.fim.uni-passau.de/cl/loopo/>
27. Größlinger A (2009) The challenges of non-linear parameters and variables in automatic loop parallelisation. PhD thesis, Department of Informatics and Mathematics, University of

- Passau, December 2009. <http://nbnresolving.de/urn:nbn:de:bvb:739-opus-17893>
28. Größlinger A, Griebl M, Lengauer C (2006) Quantifier elimination in automatic loop parallelization. *J Symbolic Computation* 41(11):1206–1221
  29. Größlinger A, Schuster S (2008) On computing solutions of linear diophantine equations with one non-linear parameter. In: Proceedings of the 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), IEEE Computer Society Press, pp 69–76
  30. Guerdoux-Jamet P, Lavenier D (1997) SAMBA: hardware accelerator for biological sequence comparison. *Comput Appl Biosci* 13(6):609–615
  31. Irigoin F, Triolet R (1989) Dependence approximation and global parallel code generation for nested loops. In: Cosnard M, Robert Y, Quinton P, Raynal M (eds) Parallel and distributed algorithms. Bonas, North-Holland, pp 297–308
  32. Jeannet B, Miné A (2009) APRON: a library of numerical abstract domains for static analysis. In: Computed Aided Verification (CAV). Lecture Notes in Computer Science, vol 5643, Springer, pp 662–667, <http://apron.cri.ensmp.fr/library>
  33. Kahn G (1974) The semantics of simple language for parallel programming. In: Proceedings of the IFIP Congress, Stockholm, pp 471–475
  34. Karp RM, Miller RE, Winograd S (1967) The organization of computations for uniform recurrence equations. *J ACM* 14(3):563–590
  35. Kienhuis B, Rijpkema E, Ed Deprettere F (2000) Compaan: deriving process networks from matlab for embedded signal processing architectures. In: Vahid F, Madsen J (eds) Proceedings of the Eighth International Workshop on Hardware/Software Codesign (CODES 2000), ACM pp 13–17
  36. Lamport L (1974) The parallel execution of DO loops. *Comm ACM* 17(2):83–93
  37. Lengauer C (1993) Loop parallelization in the polytope model. In: Best E (ed) CONCUR'93. Lecture Notes in Computer Science, vol 715, pp 398–416, Springer, 1993
  38. Loos R, Weispfenning V (1993) Applying linear quantifier elimination. *The Computer J* 36(5):450–462
  39. Pouchet L-N, Bastoul C, Cohen A, Cavazos J (2008) Iterative optimization in the polyhedral model: Part II, multidimensional time. *SIGPLAN Notices* 43(6):90–100
  40. Pouchet L-N, Bastoul C, Cohen A, Vasilache N (2007) Iterative optimization in the polyhedral model: Part I, one-dimensional time. In IEEE/ACM Fifth International Symposium on Code Generation and Optimization (CGO'07), IEEE Computer Society Press, pp 144–156
  41. Pugh W (1991) The Omega test: a fast and practical integer programming algorithm for dependence analysis. In: Proceedings of the 5th International Conference on Supercomputing, ACM, pp 4–13. 1991. <http://www.cs.umd.edu/projects/omega>
  42. Quinton P (1983) The systematic design of systolic arrays. In: Soulé FF, Robert Y, Tchuenté M (eds) Automata networks in computer science, chapter 9, pp 229–260. Manchester University Press, 1987. Also: Technical Reports 193 and 216, IRISA (INRIA-Rennes)
  43. Schrijver A (1986) Theory of linear and integer programming. Wiley, New York
  44. Smith TF, Waterman MS (1981) Identification of common molecular subsequences. *J Molecular Biology* 147(1):195–197
  45. Trifunovic K, Cohen A, Edelsohn D, Feng L, Grosser T, Jagasia H, Ladelsky R, Pop S, Sjödin J, Upadrashta R (2010) GRAPHITE two years after. In: Proceedings of the 2nd International Workshop on GCC Research Opportunities (GROW), pp 4–19, January 2010. <http://gcc.gnu.org/wiki/GROW-2010>
  46. Verdoolaege S (2009) An integer set library for program analysis. In: Advances in the theory of integer linear optimization and its extensions. AMS 2009 Western Section, 2009, <http://freshmeat.net/projects/isl/>
  47. Verdoolaege S, Nikolov H, Todor N, Stefanov P (2006) Improved derivation of process networks. In Proceedings of the 4th International Workshop on Optimization for DSP and Embedded Systems (ODES), 2006, [http://www.ece.vill.edu/~deepu/odes/odes-4\\_digest.pdf](http://www.ece.vill.edu/~deepu/odes/odes-4_digest.pdf)
  48. Verdoolaege S, Seghir R, Beyls K, Loechner V, Bruynooghe M (2007) Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica* 48(1):37–66, <http://freshmeat.net/projects/barvinok>
  49. Wolf ME, Lam MS (1991) A data locality optimizing algorithm. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM, pp 30–44
  50. Xue J (2000) Loop tiling for parallelism. Kluwer, Boston

## Polytope Model

► [Polyhedron Model](#)

## Position Tree

► [Suffix Trees](#)

## POSIX Threads (Pthreads)

POSIX Threads [1] are those created, managed, and synchronized following the POSIX standard API. POSIX stands for “Portable Operating System Interface [for Unix].”

## Bibliography

1. Nichols B, Buttlar D, Farrell JP (1996) Pthreads programming. O'Reilly & Associates, Inc., Sebastopol

## Power Wall

PRADIP BOSE

IBM Corp. T.J. Watson Research Center, Yorktown Heights, NY, USA

### Definition

The “Power Wall” refers to the difficulty of scaling the performance of computing chips and systems at historical levels, because of fundamental constraints imposed by affordable power delivery and dissipation. The single biggest factor that has led the industry into encountering this wall in the past decade is the significant change in traditional CMOS chip design evolution, which were driven previously by Dennard scaling rules [5, 15].

## Discussion

### Introduction

Power delivery and dissipation limits have emerged as a key constraint in the design of microprocessors and associated systems even for those targeted for the high end server product space. At the low end of the performance spectrum, power has always dominated over performance as the primary design constraint. However, while battery life expectancies have shown modest increases, the larger demand for increased functionality and speed has increased the severity of the power constraint in the world of handheld and mobile systems. At the high end, where performance was always the primary driver, we are witnessing a trend (dictated primarily by CMOS technology scaling constraints) where increasingly, energy and power limits are dictating the high-level processing paradigms, as well as the lower level issues related to clocking and circuit design. Thus, regardless of the application domain, power consumption constitutes a primary barrier or “wall” when it comes to achieving cost-effective performance growth in future systems. In this entry, we will

examine the fundamental causes that have led us up to this *power wall*, and discuss the key solution approaches at hand to circumvent this barrier.

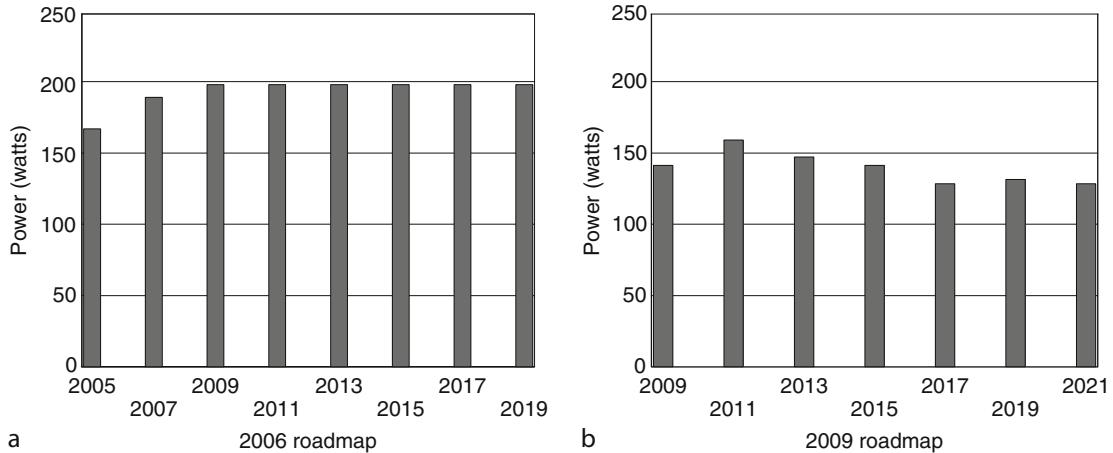
### Power Trends

**Figure 1** shows the expected maximum chip power (for high performance processors) through the year 2020. The data plotted is based on the 2006 (**Fig. 1a**) and then updated 2009 projections (**Fig. 1b**) made by the International Technology Roadmap for Semiconductors (ITRS) [61]. The 2006 projection indicated that beyond the continued growth period (through year 2008) for high-end microprocessors, there will be a saturation in the maximum chip power (with a projected cap of around 200 W from years 2008 all the way through 2020). This is due to thermal/packaging and die size limits that had already started to kick in by 2005. Single-thread performance scaling had actually started to get limited by chip-level power density since the latter part of the 1990s, as depicted in **Fig. 2** (adapted from a 2005 keynote speech by David Yen [65], who was then the executive VP of scalable systems at Sun Microsystems). Beyond a certain power (and power density) regime, air cooling is not sufficient to dissipate the heat generated; on the other hand, use of liquid cooling and refrigeration causes a sharp increase of the cost–performance ratio. Thus, power-aware design techniques, methodologies, and tools are of the essence at all levels of design. The 2009 ITRS projection shows that the long-term power cap for high-performance, server-class microprocessors has been revised downward to 130 W. This has to do with factors like the current delivery limits in typical blade server form factors that have dominated the midrange server products since 2006.

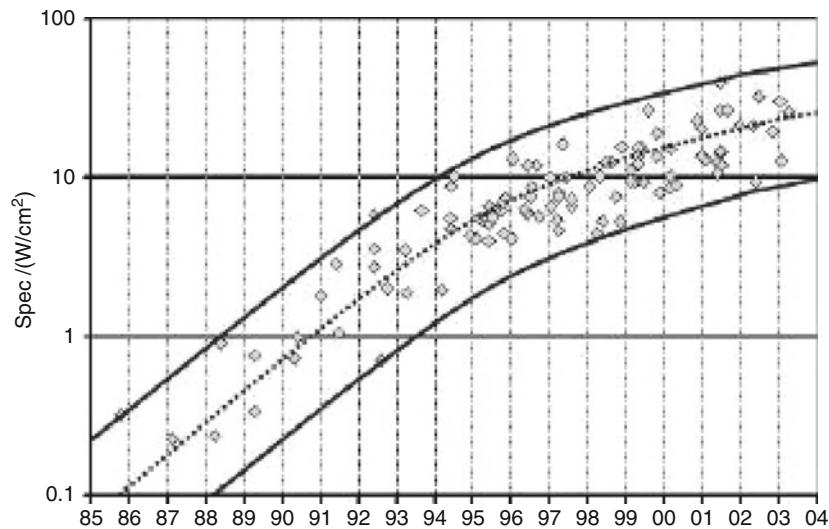
### CMOS Technology Determinants

In this subsection, we present a summary of the technological trends that have caused chip-level power and power density to reach the levels that have caused us to identify the *power wall* as a fundamental barrier to achieving cost-effective performance growth in future computing systems.

At the elementary transistor gate (e.g., an inverter) level, total power dissipation can be formulated as the



**Power Wall. Fig. 1** Maximum chip power projection – high performance with heatsink



**Power Wall. Fig. 2** Performance/power-density (SPEC/[W/cm<sup>2</sup>]) trends from 1985 to 2004 [65]

sum of three major components: switching loss, leakage and short-circuit loss [6, 12, 19, 21].

$$\begin{aligned} \text{Power}_{\text{device}} = & (1/2) C \cdot V_{dd} \cdot V_{\text{swing}} \cdot a \cdot f \\ & + I_{\text{leakage}} \cdot V_{dd} + I_{sc} \cdot V_{dd} \end{aligned} \quad (1)$$

where,  $C$  is the output capacitance,  $V_{dd}$  is the supply voltage,  $f$  is the chip clock frequency, and  $a$  is the activity factor ( $0 \leq a \leq 1$ ) which determines the device switching frequency;  $V_{\text{swing}}$  is the maximum voltage swing across the output capacitor, which in general can be less than  $V_{dd}$ ;  $I_{\text{leakage}}$  is the leakage current and  $I_{sc}$  is the short-circuit current. In the literature,  $V_{\text{swing}}$  is often approximated to be equal to  $V_{dd}$  (or simply  $V$

for short) making the switching loss  $\sim (1/2)C \cdot V^2 \cdot a \cdot f$ . Also, as discussed in [19], for a prior generation range of  $V_{dd}$  (say 1 V–3 V) switching loss:  $(1/2)CV^2af$  was the dominant component, assuming the activity factor to be above a reasonable minimum. So, as a first-order approximation, for chips belonging to the previous generation (e.g., CMOS 180 nm, and before), we may ignore the leakage and short-circuit components in Eq. 1. In other words, in the context of switching power dominated technology generations, we may formulate the power dissipation to be:

$$\text{Power}_{\text{chip}} = (1/2) \left[ \sum C_i \cdot V_i^2 \cdot a_i \cdot f_i \right] \quad (2)$$

- Where,  $C_i$ ,  $V_i$ ,  $a_i$ , and  $f_i$  are unit- or block-specific average values in the most general case; the summation is taken over all blocks or units  $i$ , at the microarchitecture level (e.g., icache, dcache, integer unit, floating point unit, load-store unit, register files and buses [if not included in individual units], etc). Also, for the voltage range considered, the operating frequency is roughly proportional to the supply voltage; and the capacitance  $C$  remains roughly the same if we keep the same design but scale the voltage. If a single voltage and clock frequency are used for the whole chip, the above reduces to:

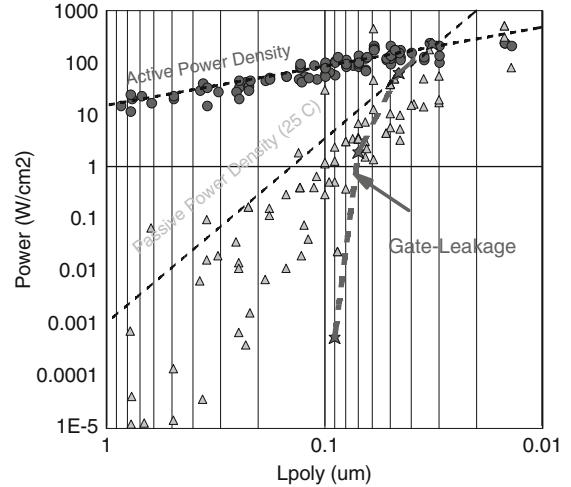
$$\text{Power}_{\text{chip}} = V^3 \cdot (\sum K_i^V \cdot a_i) = f^3 \cdot (\sum K_i^f \cdot a_i) \quad (3)$$

- If we consider the very worst-case activity factor for each unit  $i$ , i.e., if  $a_i = 1$  for all  $i$ , then, an upper bound on the maximum chip power may be formulated as:

$$\text{MaxPower}_{\text{chip}} = K_V \cdot V^3 = K_F \cdot f^3 \quad (4)$$

where  $K_V$  and  $K_F$  are design-specific constants. Note that an estimation of peak or maximum power is important, for the purposes of determining the packaging and cooling solution required. The larger the maximum power, the more expensive is the net cooling solution. Note that the formulation in Eq. 4 is overly conservative, as stated. In practice, it is possible to estimate the worst-case achievable maximum for the activity factors. This allows the designers to come up with a tighter bound on maximum power before the packaging decision is made.

The last Eq. 4 is what leads to the so-called cube-root rule [19], where redesigning a chip to operate at  $1/2$  the voltage (and frequency) results in the power dissipation being lowered to  $(1/2)^3$  or  $1/8$  of the original. This implies the single-most efficient method for reducing power dissipation for a processor that has already been designed to operate at high frequency, namely, reduce the voltage (and hence the frequency). There is a limit, however, of how low  $V_{dd}$  can be reduced (for a given technology), which has to do with manufacturability and circuit reliability issues. Thus, a combination of microarchitecture and circuit techniques to reduce power consumption, without necessarily employing multiple or variable supply voltages is of special relevance in the design of robust systems.



**Power Wall. Fig. 3** Active and major leakage power component trends [67]

Figure 3 shows the analytically projected trend [67] of escalation in three of the major components of microprocessor power consumption, namely, active or capacitive switching power, subthreshold leakage power and gate leakage power. (The leakage current referred to in Eq. 1 above consists of two major components: subthreshold leakage and gate leakage. There are other components of leakage as well, as discussed below. The short-circuit loss referred to in Eq. 1 is not a technology-dependent component, and so is ignored in this discussion). In post-180 nm technologies, static (i.e., leakage or standby) power has increasingly become a major (if not the dominating) component of chip power. As discussed in [1], the three major types of leakage effects are (a) sub-threshold, (b) gate, and (c) reverse-biased, drain- and source-substrate junction band-to-band tunneling (BTBT). With technology scaling, each of these leakage components tends to increase drastically. For example, as technology scales downward, the supply voltage ( $V_{dd}$ ) must also scale down to reduce dynamic power and maintain device reliability. However, this requires the scaling down of the threshold voltage ( $V_{th}$ ) to maintain reasonable gate overdrive (and therefore performance), which is a function of  $(V_{dd} - V_{th})$ . However, lowering the threshold voltage causes substantial increases in leakage current, and therefore standby power, inspite of the lower  $V_{dd}$ . The subthreshold channel leakage current in an MOS

device is governed by an equation that looks like [26]:

$$I_{\text{leakage}} = K_w * W \cdot 10^{-V_{\text{th}}/S} \quad (5)$$

where  $K_w$ , measured in units of micro-amps per micron ( $\mu\text{A}/\mu\text{m}$ ) can be thought of as the width-sensitivity coefficient;  $W$  is the device width and  $S$  is the subthreshold swing (measured in millivolts, like the threshold voltage  $V_{\text{th}}$ ). (In [26], the value of  $K_w$  is quoted to be 10).  $S$  is a parameter that is defined to characterize the efficiency of a device in turning on or off. It can be shown that the turn-off characteristic of a device is proportional to the thermal voltage ( $kT/q$ ) and the ratio of junction capacitance ( $C_j$ ) to oxide capacitance ( $C_{\text{ox}}$ ) [8]. The parameter  $S$  can be formulated as:

$$S = 2.3 (kT/q) \cdot (1 + C_j/C_{\text{ox}}) \quad (6)$$

This parameter is usually specified in units of millivolts per decade and it defines how many millivolts (mV) the gate voltage must drop before the drain current is reduced by one decade. The thermal voltage  $kT/q$  is equal to 26 mV at room temperature. Thus, at room temperature, the minimum value of  $S$  is about 60 mV per decade. This means that an ideal device at room temperature would experience a 10X reduction in drain current for every 60 mV reduction of the gate voltage  $V_{\text{gs}}$  in the subthreshold region. In the deep submicron era, a typical transistor device has an  $S$  value in the range of 85–90 mV per decade.

Note also, that the threshold voltage  $V_{\text{th}}$  (Eq. 5) is itself a function of temperature ( $T$ ); in fact  $V_{\text{th}}$  decreases by 2.5 mV/K as temperature increases. Also,  $K_w$  itself is a strong function of temperature ( $\sim T^2$ ). Thus, as  $T$  increases, leakage current goes up dramatically, both because of its dependence on  $T$  and because  $V_{\text{th}}$  goes down. The delay of an inverter gate is given by the alpha-power model [51] as:

$$T_g \sim \frac{L_{\text{eff}} V_{\text{dd}}}{\mu(T) \cdot (V_{\text{dd}} - V_{\text{th}})^{\alpha}} \quad (7)$$

where,  $\alpha$  is typically around 1.3 and  $\mu$  is the mobility of carriers (which is a function of temperature  $T$ ,  $\mu(T) \sim T^{-1.5}$ ). As  $V_{\text{th}}$  decreases,  $(V_{\text{dd}} - V_{\text{th}})$  increases so the inverter becomes faster. As  $T$  increases,  $(V_{\text{dd}} - V_{\text{th}})$  increases, but  $\mu(T)$  decreases [36]. This latter effect dominates; so, with higher temperatures, the logic gates in a processor generally become slower.

## Power-Performance Efficiency Metrics

The most common (and perhaps obvious) metric to characterize the power-performance efficiency of a microprocessor is a simple ratio, like mips/watt. This attempts to quantify the efficiency by projecting the performance achieved or gained (measured in millions of instructions per second) for every watt of power consumed. Clearly, the higher the number, the “better” the machine is. Dimensionally, mips/watt equates to the inverse of the average energy consumed per instruction. This seems a reasonable choice for some domains where battery life is important. However, there are strong arguments against it in many cases, especially when it comes to characterizing higher end processors. Performance has typically been the key driver of such server-class designs and cost or efficiency issues have been of secondary importance. Specifically, a design team may well choose a higher frequency design point (which meets maximum power budget constraints) even if it operates at a much lower mips/watt efficiency compared to one that operates at better efficiency but at a lower performance level. As such,  $(\text{mips})^2/\text{watt}$  or even  $(\text{mips})^3/\text{watt}$  may be the metric of choice at the high end. On the other hand, at the lowest end, where battery-life (or energy consumption) is the primary driver, one may want to put an even greater weight on the power aspect than the simplest mips/watt metric, i.e., one may just be interested in minimizing the watts for a given workload run, irrespective of the execution time performance, provided the latter does not exceed some specified upper limit.

The “mips” metric for performance and the “watts” value for power may refer to average or peak values, derived from the chip specifications. For example, for a 1GHz ( $=10^9$  cycles/s) processor which can complete up to 4 instructions per cycle, the theoretical peak performance is 4,000 mips. If the average completion rate for a given workload mix is  $p$  instructions per cycle, then the average mips would equal 1,000 times  $p$ . However, when it comes to workload-driven evaluation and characterization of processors, metrics are often controversial. Apart from the problem of deciding on a “representative” set of benchmark applications, there are fundamental questions which persist about how to boil down “performance” into a single (“average”) rating that is meaningful in comparing a set of machines. Since power consumption varies, depending on the

program being executed, the issue of benchmarking is also relevant in assigning an average power rating. In measuring power and performance together for a given program execution, one may use a fused metric like power-delay product (PDP) or energy-delay product (EDP) [21, 49]. In general, the PDP-based formulations are more appropriate for low-power, portable systems, where battery-life is the primary index of energy efficiency. The mips/watt metric is an inverse PDP formulation, where delay refers to average execution time per instruction. The power-delay product, being dimensionally equal to energy, is the natural metric for such systems. For higher end systems (e.g., workstations) the EDP-based formulations are deemed to be more appropriate, since the extra delay factor ensures a greater emphasis on performance. The  $(\text{mips})^2/\text{watt}$  metric is an inverse EDP formulation. For the highest performance, server-class machines, it may be appropriate to weight the “delay” part even more. This would point to the use of  $(\text{mips})^3/\text{watt}$ , which is an inverse  $\text{ED}^2\text{P}$  formulation. Alternatively, one may use  $(\text{cpi})^3 \cdot \text{watt}$  as a direct  $\text{ED}^2\text{P}$  metric, applicable on a “per instruction” basis (see [12]).

The  $\text{energy}^*(\text{delay})^2$  metric, or  $\text{perf}^3/\text{power}$  formula is analogous to the cube-root rule [19] which follows from constant voltage scaling arguments (see Section “CMOS Technology Determinants”, Eq. 4). Clearly, to formulate a voltage-invariant power-performance characterization metric, we need to think in terms of  $\text{perf}^3/(\text{power})$ . When we are dealing with the SPEC benchmarks, one may therefore evaluate efficiency as  $(\text{SPECrating})^x/\text{watt}$ , or  $(\text{SPEC})^x/\text{watt}$  for short; where the exponent value  $x$  ( $=1, 2$ , or  $3$ ) may depend on the class of processors being compared.

Brooks et al. [12] discuss the power-performance efficiency data for a range of commercial processors of approximately the same generation (circa year 2000).  $\text{SPEC}/\text{watt}$ ,  $\text{SPEC}^2/\text{watt}$ , and  $\text{SPEC}^3/\text{watt}$  are used as the alternative metrics, where SPEC stands for the processor’s SPEC rating [25]. The data validates our assertion that depending on the metric of choice, and the target market (determined by workload class and/or the power/cost) the conclusion drawn about efficiency can be quite different. For performance-optimized, high-end processors, the  $\text{SPEC}^3/\text{watt}$  metric seems to be fairest. For “power-first” processors,  $\text{SPEC}/\text{watt}$  seems to be the fairest.

Recently, there has been a strong motivation made for energy-proportional computing [4], in the context of future data centers and cloud computing. In this style of measuring system-level energy efficiency, it is not enough to assess the efficiency at the peak utilization levels; rather, how well the system is able to adjust the power level downward, as the workload demand decreases, is also of interest. The recently announced SPEC power benchmark [25] is intended, in part, to measure the degree to which a given server system is energy proportional. The currently used workload in SPECpower is the specjbb application, and the evaluation modality is to measure the power and performance across a range of system-level utilizations, from 100%, down through 0% (11 data points). The benchmark calls for adding up the corresponding power and performance numbers at those 11 data points and then computing the ratio of the summed performance value and the summed power number. The higher the number, the better is the energy proportional characteristic of the measured system.

## A Review of Key Ideas in Power-Aware Architectures

In this section, a brief review of power-efficient design concepts will be covered. The motivation, of course, is to examine solution approaches for avoiding the power wall, while preserving performance growth in next generation systems. The initial attention will be on dynamic (also known as “active” or “switching”) power governed by the  $\text{CV}^2\text{af}$  formula. Recall that  $C$  refers to the switching capacitance,  $V$  is the supply voltage,  $a$  is the activity factor ( $0 < a < 1$ ), and  $f$  is the operating clock frequency. Power reduction ideas must therefore focus on one or more of these basic parameters. Reducing active power generally results in reduction of on-chip temperatures, and this indirectly causes leakage power to go down as well. Similarly, any increase in efficiency directed at lowering the latch count (e.g., by reducing the basic pipeline depth, or by reducing the number of back-end execution pipes within a given functional unit) also results in area and leakage reduction as a side benefit. However, later in this section, we also deal with the problem of mitigating leakage power directly, by providing microarchitectural support to what are primarily circuit-level mechanisms.

## Power Efficiency at the Processor

### Core Level

In this sub-section, we examine the key ideas that have been proposed in terms of (micro)architectural support for power-efficiency, at the level of a single processor core. Early research ideas started being presented since 1998 at a few key conference workshops (e.g., [57–59]); later the field of power-aware microarchitectures matured to the point where all major computer architecture conferences now all have specific sessions devoted to ideas for performance growth in the current power-constrained design era.

The effective (average) value of  $C$  can be reduced by using: (a) area-efficient designs for various macros; and (b) adaptive structures, that change in effective size, latency, or communication bandwidth depending on the needs of the input workload. The average value of  $V$  can be reduced via dynamic voltage scaling, i.e., by reducing the voltage as and when required or possible. Microarchitectural support, in this case, is not required, unless the mechanisms to detect “idle” periods or temperature overruns make use of counter-based “proxies,” specially architected for this purpose. Note again, however, that since reducing  $V$  also requires (or results in) reduction of the operating frequency,  $f$ , net power reduction has a cubic effect; thus, dynamic voltage and frequency scaling (DVFS) is one of the most effective way of power reduction). Deciding when and how to apply DVFS, as a function of the input workload characteristics and overall operating environment, is very much a microarchitectural issue. It is a problem that is increasingly relevant in the era of variability-tolerant, power-efficient multi-core chip design, and we will touch on it briefly in section “►Conclusions.”

The average value of the activity factor,  $a$ , can be reduced by: (a) the use of clock-gating, where the normally free-running, synchronous clock is disabled in selected units or sub-units within the system based on information or predictions about current or future activity in those regions; (b) reducing unnecessary “speculative waste” resulting from executing instructions in mis-speculated branch paths or prefetching useless instructions and data into caches, based on wrong or ill-timed guesses; and (c) the use of data representations and instruction schedules that result in reduced switching.

Microarchitectural support is provided in the form of added mechanisms to: detect, predict, and control the generation of the applied gating signals; or aid in power-efficient data and instruction encodings. Compiler support for generating power-efficient instruction scheduling and data partitioning or special instructions for “nap/doze/sleep” control, if applicable, must also be considered under this category. Power-efficient task scheduling at the system software (i.e., OS and hypervisor) level is also an example of dynamic load balancing that can make the activity distribution uniform over a multi-core processor system, and thereby help reduce power density (and hence temperature and overall power).

While clock-gating helps eliminate (or drastically reduce) active or switching power when a given macro, sub-unit or unit is idle, power-gating can be used to also eliminate the residual leakage power of that idle entity. In this case, as described in detail later on, the power supply voltage  $V$  is itself gated off from the target circuit block, with the help of a header or footer transistor. Here, the need for microarchitectural support in the form of predictive control of the gating signal is even stronger because of the relatively large performance overheads that would be incurred without such support. There are other techniques, like adaptive body-biasing that are also targeted at leakage power control; and these too require some degree of microarchitectural support. However, these techniques are most relevant to bulk-CMOS designs (as opposed to silicon-on-insulator [SOI]-CMOS technology), and are predominantly device- and circuit-level methods. As such, we do not dwell on them in this entry.

Lastly, the average value of the design frequency,  $f$ , can be controlled or reduced by using: (a) variable, multiple or locally asynchronous (self-timed) clocks – e.g., in GALS [29] designs; (b) clock-throttling, where the frequency is reduced dynamically in response to power or temperature overrun indicators; (c) reduced pipeline depths in the baseline microarchitecture definition.

Henceforth, the focus of consideration is: power-aware microarchitectural constructs that use  $C$ ,  $a$ , or  $f$  as the primary lever for reducing active power; and those that use the supply voltage  $V$  as the primary lever for reducing leakage power. In any such proposed processor architecture, the efficacy of the particular

power reduction method that is used must be assessed by understanding the net performance impact. Here, depending on the application domain (or market), a PDP, EDP or ED<sup>2</sup>P metric for evaluating and comparing power-performance efficiencies must be used. (See earlier discussion in section “Power-Performance Efficiency Metrics”).

### Optimal Pipeline Depth

A fundamental question that is asked has to do with pipeline depth. Is a deeply pipelined, high frequency (“speed demon”) design better than an IPC-centric lower frequency (“braniac”) design? In the context of the topic of this entry, “better” must be judged in terms of power-performance efficiency.

Let us consider, first, a simple, hazard-free, linear pipeline flow process, with  $k$  stages. Let the time for the total logic (without latches) to compute one answer be  $T$ . Assuming that the  $k$  stages into which the logic is partitioned are of equal delay, the time per stage and thus the time per computation becomes (see [39], Chap. 2):

$$t = T/k + D \quad (8)$$

where  $D$  is the delay added due to the staging latch. The inverse of  $t$  determines the clocking rate or frequency of operation. Similarly, if the energy spent (per cycle, per second or over the duration of the program run) in the logic is  $W$  and the corresponding energy spent per level of staging latches is  $L$ , then the total energy equation for the  $k$ -stage pipelined version is roughly,

$$E = L \cdot k + W \quad (9)$$

The energy equation assumes that the clock is free-running, i.e., on every cycle, each level of staging latches

is clocked to enable the advancement of operations along the pipeline. (Later, we shall consider the effect of clock-gating). Equations 8 and 9, when plotted as a function of  $k$ , are depicted in Fig. 4a and b respectively.

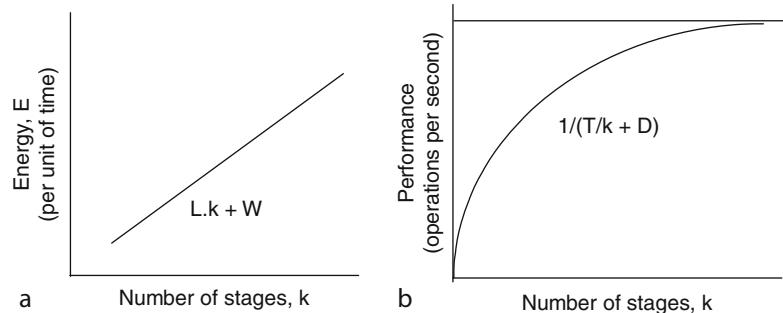
As the number of stages increases, the energy or power consumed increases linearly; while, the performance also increases, but not as fast. In order to consider the PDP-based power-performance efficiency, we compute the ratio:

$$\begin{aligned} \frac{\text{Power}}{\text{Performance}} &= (L \cdot k + W) (T/k + D) \\ &= L \cdot T + W \cdot D + (L \cdot D \cdot k^2 + W \cdot T) / k \end{aligned} \quad (10)$$

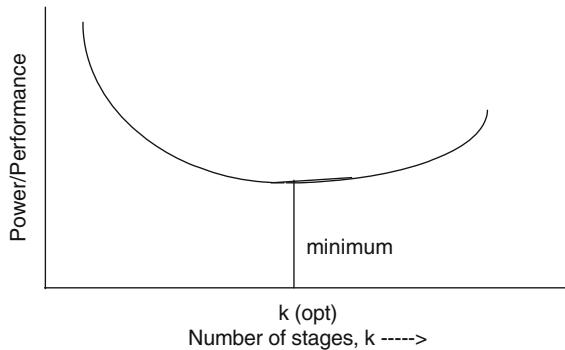
Figure 5 shows the general shape of this curve as a function of  $k$ . Differentiating the right hand side expression in Eq. 10 and setting it to zero, one can solve for the optimum value of  $k$  for which the power-performance efficiency is maximized, i.e., the minimum of the curve in Fig. 4b can be shown to occur when

$$k(\text{opt.}) = \sqrt{(W \cdot T) / (L \cdot D)} \quad (11)$$

Larson [42] first published the above analysis, albeit from a cost/performance perspective. This analysis shows that, at least for the simplest, hazard-free pipeline flow, the highest frequency operating point achievable in a given technology may not be the most energy-efficient! Rather, the optimal number of stages (and hence operating frequency) is expected to be at a point which increases for greater  $W$  or  $T$  and decreases for greater  $L$  or  $D$ . For a prior generation POWER4-class ( $\sim 0.18 \mu$ ) super scalar processor operating at around 1 GHz, [16, 60], the floating point arithmetic unit is estimated to yield values of  $T = 7.5$  ns,  $D = 0.15$  ns,



**Power Wall. Fig. 4** Power and performance curves for idealized pipeline flow



**Power Wall. Fig. 5** Power–performance ratio curve for idealized pipeline flow

$W = 0.15 \text{ W}$ , and  $L = 0.1 \text{ W}$ . This yields a  $k(\text{opt.}) \sim 8$  (rounded down from 8.67), if we use the idealized formalism (Eq. 11) above.

For real super scalar machines, the number of latches in the overall design tends to go up much more sharply with  $k$  than the linear assumption in the above model. This tends to make  $k$  (opt) even smaller. Also, in real pipeline flow with hazards, e.g., in the presence of branch-related stalls and disruptions, performance actually peaks at a certain value of  $k$  before decreasing [17, 24] (instead of the asymptotically increasing behavior shown in Fig. 4a). This effect would also lead to decreasing the effective value of  $k$  (opt). (However,  $k(\text{opt.})$  increases if we use EDP or  $\text{ED}^2\text{P}$  metrics instead of the PDP metric used.). As the number of pipeline stages ( $k$ ) is increased for a given computation data path, we say that the *depth* of the pipeline increases. This also implies that the levels of combinational logic within each pipeline stage decreases; so, the combinational logic delay per stage decreases as  $k$  increases. In detailed simulation-based analysis of a POWER4-class super scalar machine, it has been shown [56, 68] that the optimal pipeline depth using a  $\text{ED}^2\text{P}$  metric like  $(\text{BIPS})^3/\text{W}$  (where BIPS is the standard performance metric of billions of instructions completed per second) corresponds to around 18 FO4 (Fan-out-of-four (FO4) delay is defined as the delay of one inverter driving four copies of an equally sized inverter. The amount of logic and latch overhead per pipeline stage is often measured in terms of FO4 delay. Decreasing logic FO4 delay per pipeline stage means deeper pipelines (larger values of  $k$ ) and vice versa.) per pipe stage for

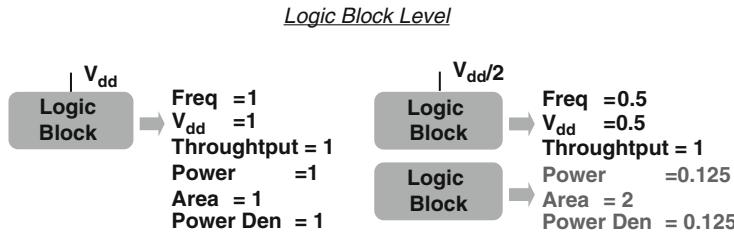
SPEC2000 workloads. For commercial workloads like TPC-C, the optimal point is shown to shift to shallower pipelines (25–28 FO4). In contrast, note that if one considered a power-unaware performance-only metric, like BIPS, the optimal pipeline depth for SPEC2000 is around 10 FO4 per stage. For TPC-C, the performance-only optimal point is reported to be [56, 68] pretty flat across the 10–14 FO4 points. For scientific workloads, loop tuning for performance optimization [7] can alter measured power-performance efficiency metrics significantly in some cases. Compiler techniques for super scalar efficiency enhancements are omitted for brevity in this entry.

### Exploiting Parallelism to Scale the Power Wall

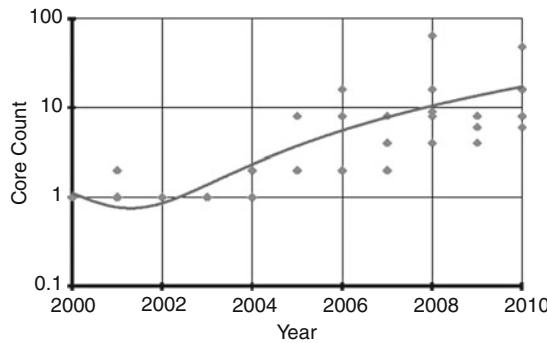
As single-thread frequency and performance growth stalls (driven by technology trends), multi-core parallelism is the new trend. Figure 6 is shown to explain the fundamentals of why parallelism helps power efficiency. If a single task, executed at a given voltage-frequency point, can be split into two independent tasks, each operating at half the voltage and frequency, the net throughput performance remains the same, but the active power density scales down by a factor of 8. A particular application of this concept, even at the level of a single processor core is that of SIMD acceleration as described in the next sub-section. Figure 7 depicts the current trend [28] of growth on number of cores over technology generation in the current regime of power density constrained microprocessor design.

### Vector/SIMD Processing and Hybrid Architectures

Vector/SIMD modes of parallelism present in current architectures afford a power-efficient method of extending performance for vectorizable codes. Fundamentally, this is because: for doing the work of fetching and processing a single (vector) instruction, a large amount of data is processed in a parallel or pipelined manner. If we consider a SIMD machine, with  $p$   $k$ -stage functional pipelines then looking at the pipelines alone, one sees a  $p$ -fold increase of performance, with a  $p$ -fold increase in power, assuming full utilization and hazard-free flow, as before. Thus, an SIMD pipeline unit offers the potential of scalable growth in performance, with commensurate growth in power, i.e., at constant power-performance efficiency. If, however, one includes the front-end instruction cache and fetch/dispatch unit



**Power Wall.** Fig. 6 The classic argument of how parallelism can be exploited to reduce power



**Power Wall.** Fig. 7 Microprocessor core count over time [28]

that are shared across the  $p$  SIMD pipelines, then power-performance efficiency actually grows with  $p$ . This is because, the power dissipation behavior of the instruction cache (memory) and the fetch/decode path remains essentially invariant with  $p$ , while net performance grows linearly with  $p$ . In terms of net energy, the front-end consumption actually decreases significantly in SIMD mode, since the number of instructions executed is much less than in scalar mode; and overall, since the execution time decreases, the net savings in leakage energy usually results in a significant net positive benefit for the full machine.

The SIMD extension is actually an example of the general concept of using power-efficient specialized hardware (or accelerators) as part of processor design. Such accelerators can be turned off (e.g., power-gated off) when not in use, while significant pieces of the general purpose (“scalar”) core can be switched off when the program encounters a long burst of a code that can be offloaded to an active accelerator. Such accelerators can be fixed function, “table-lookup”-type logic at one extreme, all the way through to programmable sub-cores at the other end of the spectrum. The concept

of hybrid or heterogeneous core architectures [41] is an example of the latter extreme. In a super scalar machine with a vector/SIMD (or other accelerator) extension, the overall power-efficiency increase is limited by the fraction of code that runs in vector/SIMD (or other accelerator) mode (per Amdahl’s Law).

#### Clock-Gating and Microarchitectural Support

Clock-gating refers to circuit-level control (e.g., see [23, 63]) for disabling the clock to a given set of latches, a macro, a bus, to a cache or register file access path, or an entire unit, on a particular machine cycle. In current generation server-class microprocessors, about 50–70% of the active (switching) power is consumed by the clock distribution network and its latch load alone. As reported in [9], the major part of the clock power is dissipated close to the leaf nodes of the clock tree that drive latch banks. Since a clock-gated latch keeps its current data value stable, clock gating prevents signal transitions of invalid data from propagating down the pipeline thereby reducing switching power in the combinational logic between latches. In addition to reducing dynamic power, clock gating can also reduce static (leakage) power. As already explained, leakage current in CMOS devices is exponentially dependent on temperature. The temperature reduction brought on by clock-gating can therefore significantly reduce the leakage power as well.

(Micro)architectural support for conventional clock-gating can be provided in at least three ways: (a) dynamic detection of idle modes in various clocked units or regions within a processor or system; (b) static or dynamic prediction of such idle modes; (c) using “data valid” bits within a pipeline flow path to selectively enable/disable the clock applied to the pipeline stage latches. If static prediction is used, the compiler inserts special “nap/doze/sleep/wake” type instructions

where appropriate, to aid the hardware in generating the necessary gating signals. Methods (a) and (b) result in coarse-grain clock-gating, where entire units, macros or regions can be gated off to save power; while, method (c) results in fine-grain clock-gating, where unutilized pipe segments can be gated off during normal execution within a particular unit, like the FPU. The detailed circuit-level implementation of gated-clocks, the potential performance degradation, inductive noise problems, etc., are not discussed in this entry. However, these are very important issues that must be dealt with adequately in today's power-constrained designs.

Referring back to section "A Review of Key Ideas in Power-Aware Architectures" and Fig. 5, note that since (fine-grain) clock-gating effectively causes a fraction of the latches to be "gated off," we may model this by assuming that the effective value of  $L$  decreases when such clock-gating is applied. This has the effect of increasing  $k$  (opt.), i.e., the operating frequency for the most power-efficient pipeline operation can be increased in the presence of clock-gating. This is an added benefit.

In recently reported work [30], the limits of clock-gating efficiency has been examined and then stretched by adding a couple of new advances: transparent pipeline clock-gating (TCG) [31] and elastic pipeline clock-gating (ECG) [32]. TCG introduces a new way of clock-gating pipelines. In traditional clock-gating, latches are held opaque to avoid data races between adjacent latch stages; thus,  $N$  clock pulses are needed to propagate a single data item through an  $N$ -stage pipeline, even if at a given clock cycle all other (i.e.,  $N - 1$ ) stages have invalid input data. In a transparent clock-gated pipeline, latches are held *transparent* by default. TCG is based on the concept of *data separation*. Assume that a pair of data items A and B simultaneously move through a TCG pipeline. A data race between A and B is avoided by separating the two data items by clocking or gating a latch stage opaque, such that the opaque latch stage acts as a barrier separating the two data items from each other. The number of clock pulses required for a data item A to move through an  $N$ -stage pipeline is no longer only dependent on  $N$ , but also on the number of clock cycles that separate A from the closest upstream data item B. For an  $N$ -stage pipeline, where B follows  $n$  clock cycles behind A, only floor ( $N/n$ ) clock pulses have to be generated to move A safely through

the pipeline. Elastic pipeline clock gating (ECG) is a different technique that achieves further efficiency by exploiting the inherent storage redundancy afforded by a traditional master-slave latch pair. ECG allows the designer to allow stall signals to propagate backward in pipeline flow logic in a stage-by-stage fashion, without incurring the leakage power and area overhead of explicitly inserted stall buffers. Logic-level details of TCG and ECG are available in the originally published papers [30–32]. As reported there, TCG enables clock power reduction to the tune of 50% over traditional stage-level clock gating under commercial (TPC-C) class workloads. Even under heavy floating point workloads where fewer bubbles are available in the pipeline, the clock power in the floating point pipeline can be reduced by 34%. The significant reduction in dynamic stall power (27%) and leakage power (44%) afforded by ECG in a floating point unit design have also been reported in the published literature.

### Predictive Power Gating

As previously indicated, leakage power is a major (if not dominant) component of total power dissipation in current and future CMOS microprocessors. Cutting off the power supply (Vdd) to major circuit blocks to conserve idle power ("sleep mode") is not a new concept, especially for batter-powered mobile systems. However, dynamically effecting such gating, on a unit-by-unit basis as function of input workload demand is not a design technique that has seen widespread usage yet, especially in server-class microprocessors. The main reasons have been the perceived risks or negative effects arising from: (a) performance and area overheads; (b) inductive noise on the power supply grid; and (c) potential design tools and verification concerns. Advances in circuit design have minimized the area and cycle-time delay overhead concerns in recent industrial practices. Microarchitectural predictive techniques (e.g., [18, 27, 38, 44]) have recently advanced to the point where now they equip designers with the tools needed to minimize any architectural performance overheads as well. The inductive noise concerns do persist, but there are known solution approaches in the realm of power distribution networks and package design that will no doubt mature to help mitigate those concerns. The design tools and verification challenge, of course

will prevail as a difficult roadblock – but again, solutions will eventually emerge to get rid of that concern. Per-core power gating, within a multi-core setting has recently gained acceptance within the chip design community (e.g., see Intel's Nehalem processor design [54]). The power-efficiency benefits of per-core power gating have been quantified recently for server and data center class workloads [43, 45].

### Variable Bit-Width Operands

One of the techniques proposed for reducing dynamic power consists of exploiting the behavior of data in programs, which is characterized by the frequent presence of small values. Such values can be represented as and operated upon as short bit-vectors. Thus, by using only a small part of the processing datapath, power can be reduced without loss of performance. Brooks and Martonosi [10] analyzed the potential of this approach in the context of 64-bit processor implementations (e.g., the Compaq Alpha™ architecture). Their results show that roughly 50% of the instructions executed had both operands whose length was less than or equal to 16 bits. Brooks and Martonosi proposed an implementation that exploits this by dynamically detecting the presence of narrow-width operands on a cycle-by-cycle basis.

### Adaptive Microarchitectures

Another method of reducing power is to adjust the size of various storage resources within a processor or system, with changing needs of the workload. Albonesi [2] proposed a dynamically reconfigurable caching mechanism, that reduces the cache size (and hence power) when the workload is in a phase that exhibits reduced cache footprint. Such downsizing also results in improved latency, which can be exploited (from a performance viewpoint) by increasing the cache cycling frequency on a local clocking or self-timed basis. Maro et al. [47] have suggested the use of adapting the functional unit configuration within a processor in tune with changing workload requirements. Reconfiguration is limited to “shutting down” certain functional pipes or clusters, based on utilization history or IPC performance. In that sense, the work by Maro et al. is not too different from coarse-grain clock-gating support, as discussed earlier. In early work done at IBM Watson, Buyuktosunoglu et al. [13] designed an adaptive issue

queue that can result in (up to) 75% power reduction when the queue is sized down to its minimum size. This is achieved with a very small IPC performance hit. Another example is the idea of adaptive register files (e.g., see [14]) where the size and configuration of the active size of the storage is changed via a banked design, or through hierarchical partitioning techniques. A recent tutorial article by Albonesi et al. [3] provides an excellent coverage of advances in the field of adaptive architectures.

### Dynamic Thermal Management

Most clock-gating techniques are geared toward the goal of reducing *average* chip power. As such, these methods do not guarantee that the worst-case (maximum) power consumption will not exceed safe limits. The processor's maximum power consumption dictates the choice of its packaging and cooling solution. In fact, as discussed in [23], the net cooling solution cost increases in a piecewise linear manner with respect to the maximum power, and the cost gradient increases rather sharply in the higher power regimes. This necessitates the use of mechanisms to limit the maximum power to a controllable ceiling, one defined by the cost profile of the market for which the processor is targeted. Most recently, in the high performance world, Intel's Pentium 4 processor is reported to use an elaborate on-chip thermal management system to ensure reliable operation [23]. At the lower end, the G3 and G4 PowerPC microprocessors [50, 52] include a Thermal Assist Unit (TAU) to provide dynamic thermal management. In recently reported academic work, Brooks and Martonosi [11] discuss and analyze the potential reduction in “maximum power” ratings without significant loss of performance, by the use of specific dynamic thermal management (DTM) schemes. The use of DTM requires the inclusion of on-chip sensors to monitor actual temperature, or proxies of temperature [11] estimated from on-chip counters of various events and rates.

### Dynamic Throttling of Communication Bandwidths

This idea has to do with reducing the width of a communication bus dynamically, in response to reduced needs or in response to temperature overruns. Examples of on-chip buses that can be throttled are:

instruction fetch bandwidth, instruction dispatch/issue bandwidths, register renaming bandwidth, instruction completion bandwidths, memory address bandwidth, etc. In the G3 and G4 PowerPC microprocessors [50, 52], the TAU invokes a form of instruction cache throttling as a means to lower the temperature when a thermal emergency is encountered.

### Speculation Control

In current generation, high performance microprocessors, branch mispredictions and mis-speculative prefetches end up wasting a lots of power. Manne et al. [22, 46] and Karkhanis et al. [37] have described means of detecting or anticipating an impending mispredict and using that information to prevent mis-speculated instructions from entering the pipeline. These methods have been shown to reduce energy waste, with minimal impact on performance.

### Power-Efficient Microarchitecture Paradigms

Now that we have examined specific microarchitectural constructs that aid power-efficient design, let us briefly examine the inherent power-performance scalability and efficiency of selected paradigms that are currently established or are emerging in the high-end processor roadmap. In particular, we consider: (a) wide-issue, speculative super scalar processors; (b) multi-cluster superscalars; (c) simultaneously multithreaded (SMT) processors; and (d) chip multiprocessors (CMP); those that use single program speculative multithreading, as well as those that are general multi-core SMP or throughput engines.

### Single-Core Superscalar Processor Paradigm

One school of thought anticipates a continued progression along the path of wider, aggressively superscalar paradigms. Researchers continue to innovate in an attempt to extract the last “ounce” of IPC-level performance from a single-thread instruction-level parallelism (ILP) model. Value prediction advances (pioneered by Lipasti et al. [62]) promise to break the limits imposed by true data dependencies. Trace caches (Smith et al. [62]) ease the fetch bandwidth bottleneck, which can otherwise impede scalability. However, increasing the superscalar width beyond a certain limit tends to yield diminishing gains in *net* performance. At

the same time, the power-performance efficiency metric (e.g., performance per watt or  $(\text{performance})^2/\text{watt}$ , etc) tends to degrade beyond a certain complexity point in the single-core superscalar design paradigm.

The microarchitectural trends beyond the current superscalar regime are effectively targeted toward the goal of extending the power-performance efficiency factors. Whenever we reach a maximum in the power-performance efficiency curve, it is time to invoke the next paradigm shift.

Next, we examine some of the promising new trends in microarchitecture that can serve as the next platform for designing power-performance scalable machines.

### Multicluseter Superscalar Processors

Zyuban et al. [66, 69] studied the class of multicluseter superscalar processors as a means of extending the power-efficient growth of the basic super scalar paradigm. One way to address the energy growth problem at the microarchitectural level is to replace a classical superscalar CPU with a set of clusters, so that all key energy consumers are split among clusters. Then, instead of accessing centralized structures in the traditional superscalar design, instructions scheduled to an individual cluster would access local structures most of the time. The main advantage of accessing a collection of local structures instead of a centralized one is that the number of ports and entries in each local structure is much smaller. This reduces the latency and energy per access. If the non-accessed sub-structures of a resource can be “gated off” (e.g., in terms of the clock), then, the net energy savings can be substantial.

According to the results obtained in Zyuban’s work, the energy dissipated per cycle in every unit or sub-unit within a superscalar processor can be modeled to vary (approximately) as  $\text{IPC}_{\text{unit}} * (\text{IW})^g$ , where IW is the issue width,  $\text{IPC}_{\text{unit}}$  is the average IPC performance at the level of the unit or structure under consideration, and g is the energy growth parameter for that unit. Then, the energy-delay product (EDP) for the particular unit would vary as:

$$\text{EDP}_{\text{unit}} \frac{\text{IPC}_{\text{unit}} * (\text{IW})^g}{\text{IPC}_{\text{overall}}} \quad (12)$$

Zyuban shows that for real machines, where the overall IPC always increases with issue width in a sub-linear manner, the overall EDP of the processor can be bounded as:

$$(IPC)^{2g-1} \leq EDP \leq (IPC)^{2g} \quad (13)$$

where  $g$  is the energy-growth factor of a given unit and IPC refers to the overall IPC of the processor; and IPC is assumed to vary as  $(IW)^{0.5}$ . Thus, according to this formulation, superscalar implementations that minimize  $g$  for each unit or structure will result in energy-efficient designs. The eliot/elpaso tool does not model the effects of multi-clustering in detail; however, from Zyuban's work, we can infer that a carefully designed multicluster architecture has the potential of extending the power-performance efficiency scaling beyond what is possible using the classical superscalar paradigm. Of course, such extended scalability is achieved at the expense of reduced IPC performance for a given superscalar machine width. This IPC degradation is caused by the added inter-cluster communication delays and other power management overhead in a real design. Some of the IPC loss (if not all) can be offset by a clock frequency boost which may be possible in such a design, due to the reduced resource latencies and bandwidths.

High-performance processors (e.g., the Compaq Alpha 21264 and the IBM POWER4/5) certainly have elements of multi-clustering, especially in terms of duplicated register files and distributed issue queues. Zyuban proposed and modeled a specific multicluster organization in his work. This simulation-based study determined the optimal number of clusters and their configurations, for the EDP metric.

### Simultaneous Multithreading (SMT)

Let us examine the SMT paradigm [64] to understand how this may affect our notion of power-performance efficiency. With SMT, assume that we can fetch from two threads (simultaneously, if the icache is dual-ported, or in alternate cycles if the icache remains single-ported). The back-end execution engine is shared across the two threads, although each thread has its own architected register space. This facility allows the utilization factors, and the net throughput performance to go up, without a commensurate increase in the maximum clocked power. This is because, the issue width  $W$  is not increased, but the execution and resource stages or

slots can be filled up simultaneously from both threads. The added complexity in the front-end, of maintaining two program counters (fetch streams) and the global register space increase adds to the power a bit, but the throughput gain is significantly higher. SMT is clearly a very area-efficient (and hence leakage-efficient) microarchitectural paradigm. IBM's POWER processor family has progressed from 2-way SMT per core in POWER5 [35] to 4-way SMT per core in POWER7 [34].

Seng and Tullsen [53] presented analysis to show that using a suitably architected SMT processor, the per-thread speculative waste can be reduced, while increasing the utilization of the machine resources by executing simultaneously from multiple threads. This was shown to reduce the average energy per instruction by 22%.

### Chip Multiprocessing

In a multiscalar-like speculative-multi-threading machine [55], different iterations of a single loop program could be initiated as separate tasks or threads on different core processors on the same chip. Iterations beyond the first one are speculatively executed, assuming no loop-carried dependencies. Register values set in one task are forwarded in sequence to dependent instructions in subsequent tasks. Execution on each processor proceeds speculatively, assuming the absence of load-store address conflicts between tasks; dynamic memory address disambiguation hardware is required to detect violations and restart task executions as needed. In this paradigm, if the performance can be shown to scale well with the number of tasks, and if each processor is designed as a limited-issue, limited-speculation (low complexity) core, it is possible to achieve better overall scalability of performance-power efficiency.

A key real trend in high-end microprocessors is classical chip multiprocessing (CMP), where multiple user programs (or sub-tasks of a single program) execute separately (and non-speculatively) on different processors on the same chip. A commonly used paradigm in this case is that of (shared memory) symmetric multiprocessing (SMP) on a chip (see Hammond et al. in [62]). Larger SMP server nodes can be built from such chips. Server system vendors like IBM have relied on such CMP paradigms as the scalable paradigm for the immediate future. IBM's POWER4 and POWER5 designs [16, 35, 60] were the first examples of this industry-wide trend; later examples being:

Intel's Montecito [48] and Sun's Niagara [40]. Most recent server-class multi-core/multi-threaded processors include Intel's Nehalem [54] and IBM's POWER7 [34]. Such CMP designs offer the potential of convenient coarse-grain clock-gating and “power-down” modes, where one or more processors on the chip may be “turned off” or “slowed down” to save power when needed.

In general, in a design era where technological constraints like power dissipation have caused a significant slowdown in the growth of single-thread execution clock frequency (and performance), parallelism via multiple cores on a die, each operating at lower than achievable single-core frequency (i.e., at larger FO4 per stage than prior generation processors), is the natural paradigm of choice for power-efficient, scalable performance growth through the next several generations of processor design. Heterogeneity in the form of different types of cores, accelerators and mixed signal components [20, 33, 41] or variable per-core frequency support are already prevalent concepts. Such heterogeneity is needed to support the diverse set of workloads that will enable the next generation of computing systems in a power-constrained design era.

## Conclusions

In this entry, we first defined the “power wall” and examined the root technological causes behind the onset of the current power-constrained design era. We then discussed issues related to power-performance efficiency and metrics from an architect’s viewpoint. We showed that depending on the application domain (or market), it may be necessary to adopt one metric over another in comparing processors within that segment. Next, we described some of the promising new ideas in power-aware microarchitecture design. This discussion included circuit-centric solutions like clock-gating, and power-gating where microarchitectural support is needed to make the right decisions at run time. We then looked at the trends in core- and chip-level microarchitecture design paradigms at a high-level, given the reality of the power wall. We limited our focus to a few key ideas and paradigms of interest in future power-aware processor design. Many new ideas to address various aspects of power reduction have been presented in recent research papers. All of these could not be discussed in this entry; but the

interested reader should certainly refer to the cited references and recent conference publications for further detailed study.

## Bibliography

1. Agrawal A, Mukhopadhyay S, Raychowdhury A, Roy K, Kim CH (2006) Leakage power analysis and reduction in nanoscale circuits. *IEEE Micro* 26(2):68–80
2. Albonesi D (1998) Dynamic IPC/clock rate optimization. In: Proceedings of the 25th annual international symposium on computer architecture (ISCA), ACM/IEEE Computer Society, Barcelona, pp 282–292
3. Albonesi DH, Balasubramonian R, Dropsho SG, Dwarkadas S, Friedman EG, Huang MC, Kursun V, Magklis G, Scott ML, Semeraro G, Bose P, Buyuktosunoglu A, Cook PW, Schuster SE (2003) Dynamically tuning processor resources with adaptive processing. *IEEE Comput* 36(12):49–58, Special Issue on Power-Aware Computing
4. Barroso L, Holzle U (2007) The case for energy proportional computing. *IEEE Comput* 40(12):33–37
5. Bohr M (2007) A 30 year retrospective on dennard’s MOSFET scaling paper. *P Solid St Circ Soc* 12(1):11–13
6. Borkar S (1999) Design challenges of technology scaling. *P IEEE Micro* 19(4):23–29
7. Bose P, Kim S, O’Connell FP (1999) Ciarfella WA Bounds modeling and compiler optimizations for superscalar performance tuning. *J Syst Architect* 45:1111–1137; Elsevier
8. Brews JR (1979) Subthreshold behavior of uniformly and non-uniformly doped long-channel MOSFET. *IEEE T Electron Dev* 26:1282–1291
9. Brooks D, Bose P, Srinivasan V, Gschwind MK, Emma PG, Rosenfield MG (2003) New methodology for early-stage, microarchitecture-level power-performance analysis of microprocessors. *IBM J Res Dev* 47(5/6):653–662
10. Brooks D, Martonosi M (1999) Dynamically exploiting narrow width operands to improve processor power and performance. In: Proceedings of the 5th international symposium on high-performance computer architecture (HPCA-5), IEEE Computer Society, Orlando
11. Brooks D, Martonosi M (2001) Dynamic thermal management for high-performance microprocessors. In: Proceedings of the 7th international symposium On high performance computer architecture, IEEE Computer Society, Nuevo Leone, pp 20–24
12. Brooks DM, Bose P, Schuster SE, Jacobson H, Kudva PN, Buyuktosunoglu A, Wellman J-D, Zyuban V, Gupta M, Cook PW (2000) Power-aware microarchitecture: design and modeling challenges for next-generation microprocessors. *P IEEE Micro* 20(6):26–44
13. Buyuktosunoglu A et al (2000) An adaptive issue queue for reduced power at high performance. In: Proceedings ISCA Workshop on complexity-effective design (WCED), Vancouver
14. Cruz J-L, Gonzalez A, Valero M, Topham NP (2000) Multiple-banked register file architectures. In: Proceedings of the international symposium on computer architecture (ISCA), Vancouver, pp 316–325

15. Dennard R et al (1974) Design of ion-implanted MOSFETs with very small physical dimensions. *IEEE J Solid St Circ* SC-9(5):256–268
16. Diefendorff K (1999) POWER4 focuses on memory bandwidth. *Microprocessor Report* 13(13):11–17
17. Dubey PK, Flynn MJ (1990) Optimal pipelining. *J Parallel Distr Com* 8(1):10–19
18. Flautner K, Kim NS, Martin S, Blaauw D, Mudge T (2002) Drowsy Caches: simple techniques for reducing leakage power. In: Proceedings of the international symposium on computer architecture (ISCA), IEEE Computer Society, Anchorage
19. Flynn MJ, Hung P, Rudd K (1999) Deep-submicron microprocessor design issues. *P IEEE Micro* 19(4):11–22
20. Gara A et al (2005) Overview of the blue gene/L system architecture. *IBM J Res Dev* 49(2/3):195–212
21. Gonzalez R, Horowitz M (1996) Energy dissipation in general purpose microprocessors. *IEEE J Solid-St Circ* 31(9):1277–1284
22. Grunwald D, Klauser A, Manne S, Pleszkun, A (1998) Confidence estimation for speculation control. In: Proceedings 25th annual international symposium on computer architecture (ISCA), ACM/IEEE Computer Society, Barcelona, pp 122–131
23. Gunther SH, Binns F, Carmean DM, Hall JC (2000) Managing the impact of increasing microprocessor power consumption. In: Proceedings of the Intel Technology Journal
24. Hartstein A, Puzak TR (2002) The optimum pipeline depth for a microprocessor. Proceedings of the 29th international symposium on computer architecture (ISCA-29), IEEE Computer Society, Anchorage
25. <http://www.specbench.org>
26. Hu C (1996) Device and technology impact on low power electronics. In: Rabaey J (ed) Low power design methodologies. Kluwer, Boston, pp 21–35
27. Hu Z, Buyuktosunoglu A, Srinivasan V, Zyuban V, Jacobson H, Bose P (2004) Microarchitectural techniques for power gating of execution units. In: Proceedings of the international symposium on low power electronics and design (ISLPED), ACM, Newport Beach
28. ISSCC (International Solid State Circuits Conference) (2010) Trends Report, [http://isscc.org/doc/2010/ISSCC2010\\_TechTrends.pdf](http://isscc.org/doc/2010/ISSCC2010_TechTrends.pdf)
29. Iyer A, Marculescu D (2002) Power-performance evaluation of globally asynchronous, locally synchronous processors. In: Proceedings of the international symposium on computer architecture (ISCA), IEEE Computer Society, Anchorage
30. Jacobson H, Bose P, Hu Z, Eickemeyer R, Eisen L, Griswell J (2005) Stretching the limits of clock-gating efficiency in server-class processors. In: Proceedings of the international symposium on high performance computer architecture (HPCA), IEEE Computer Society, San Francisco
31. Jacobson HM (2004) Improved clock-gating through transparent pipelining. In: Proceedings of the international symposium on low Power electronics and design (ISLPED), ACM, California
32. Jacobson HM et al (2002) Synchronous interlocked pipelines. In: Proceedings of the international symposium on advanced research in asynchronous circuits and systems, IEEE Computer Society, Manchester
33. Kahle JA et al (2005) Introduction to the Cell multiprocessor. *IBM J Res Dev* 49(4/5):589–604
34. Kalla R, Sinharoy B, Starke WJ, Floyd MJ (2010) Power7: IBM's next generation server processor. *IEEE Micro* 30(2):7–15
35. Kalla R, Sinharoy B, Tandler J (2004) IBM POWER5 chip: a dual-core multithreaded processor, *IEEE Micro* 24(2):40–47
36. Kanda K et al (2001) Design impact of positive temperature dependence on drain current in sub-1-V CMOS VLSIs. *IEEE JSSC*, 36(10):1559–1564
37. Karkhanis T, Bose P, Smith J (2002) Saving energy with just in time instruction delivery. In: Proceedings of the international symposium on low power electronics and design (ISLPED), ACM, Monterey
38. Kaxiras S, Hu Z, Martonosi M (2001) Cache Decay: exploiting generational behavior to reduce cache leakage power. In: Proceedings of the international symposium on computer architecture (ISCA), Goteborg
39. Kogge PM (1981) The architecture of pipelined computers. Hemisphere Publishing Corporation, New York
40. Kongetira P (2004) A 32-way multithreaded SPARC® processor. Presented at Hot Chips
41. Kumar R, Tullsen D, Jouppi N, Ranganathan P (2005) Heterogeneous chip multiprocessors. *IEEE Comput* 38(11):32–38
42. Larson AG (1973) Cost-effective processor design with an application to fast fourier transform computers. Digital systems laboratory report SU-SEL-73-037, Stanford University, Stanford; see also, Larson and Davidson (1973) Cost-effective design of special purpose processors: a fast fourier transform case study. In: Proceedings 11th annual allerton conference on circuits and system theory. University of Illinois, Champaign-Urbana, pp 547–557
43. Leverich J, Monchiero M, Talwar V, Ranganathan P, Kozyrakis C (2009) Power management of datacenter workloads using per-core power gating. *IEEE Comput Archit Lett* 8(2):48–51
44. Lungu A, Bose P, Buyuktosunoglu A, Sorin D (2009) Dynamic power gating with quality guarantees. In: Proceedings of the international symposium on low power electronics and design (ISLPED), ACM, New York
45. Madan NS, Buyuktosunoglu A, Bose P, Annavaram M (2011) Guarded power gating in a multi-core setting, presented at ISCA workshop on energy-efficient design (WEED), June 2010; to appear as a Lecture notes on computer science (LNCS) volume in 2010; see also full paper in Proceedings of the 17th international Symposium on high performance computer architecture (HPCA)
46. Manne S, Klauser A, Grunwald D (1998) Pipeline gating: speculation control for energy reduction. In: Proceedings of the 25th annual international symposium on computer architecture (ISCA), ACM/IEEE Computer Society, Barcelona, pp 132–141
47. Maro R, Bai Y, Bahar RI (2000) Dynamically reconfiguring processor resources to reduce power consumption in high-performance processors. In: Proceedings of power aware computer systems (PACS) Workshop, held in conjunction with ASPLOS, Cambridge
48. McNairy C, Bhatia R (2005) Montecito: a dual-core, dual-thread Itanium Processor. *IEEE Micro* 25(2):10–20 (see also, Hot Chips 2004)

49. Oklobdzija VG (1998) Architectural tradeoffs for low power. In: Proceedings of the ISCA Workshop on Power-Driven Microarchitectures, Barcelona
50. Reed P et al (1997) 250 MHz 5 W RISC microprocessor with on-chip L2 cache controller. Dig Tech Pap IEEE Int Solid St Circ Conf 40:412
51. Sakurai T, Newton R (1990) Alpha-power law MOSFET model and its applications to CMOS inverter delay and other formulas. IEEE JSSC 25(2):584–594
52. Sanchez H et al (1997) Thermal management system for high performance PowerPC microprocessors. In: Digest of papers, IEEE COMPON, p 325
53. Seng JS, Tullsen DM, Cai G (2000) The power efficiency of multithreaded architectures. In: Invited talk presented at: ISCA Workshop on Complexity-Effective Design (WCED), Vancouver
54. Singhal R (2008) Inside intel core microarchitecture (Nehalem). Presented at Hot Chips-20, Stanford
55. Sohi G, Breach SE, Vijaykumar TN (1995) Multiscalar Processors. In: Proceedings of the 22nd annual international symposium on computer architecture, IEEE CS Press, Los Alamitos, pp 414–425
56. Srinivasan V, Brooks D, Gschwind M, Bose P, Zyuban V, Strenski PN, Emma PG (2002) Optimizing pipelines for power and performance. In: Proceedings of the 35th annual IEEE/ACM symposium on microarchitecture (MICRO-35), ACM/IEEE, Istanbul
57. Talks presented at the ISCA workshops on complexity effective design (WCED-2000 through WCED-2006), <http://www.csl.cornell.edu/~albonesi/wced.html>
58. Talks presented at the kool chips workshops (1998) <http://www.cs.colorado.edu/~grunwald/LowPowerWorkshop>
59. Talks presented at the power aware computer systems (PACS) Workshops; e.g. the 2004 offering: <http://www.ece.cmu.edu/~pacs04/>
60. Tendler JM, Dodson JS, Fields JS Jr, Le H, Sinharoy B (2002) POWER4 system microarchitecture. IBM J Res Dev 46(1):1–116
61. The International Technology Roadmap for Semiconductors. <http://www.itrs.net/reports.html>
62. Theme issue (1997) The future of processors. IEEE Comput 30(9):97–93
63. Tiwari V et al (1998) Reducing power in high-performance microprocessors. In: Proceedings of the IEEE/ACM Design Automation Conference. ACM, New York, pp 732–737
64. Tullsen DM, Eggers SJ, Levy HM (1995) Simultaneous Multithreading: Maximizing On-Chip parallelism. In: Proceedings of the 22nd annual international symposium on computer architecture, Santa Margherita Ligure, pp 292–403
65. Yen D (2005) Chip multithreading processors enable reliable high throughput computing. Keynote speech at international symposium on reliability physics (IRPS)
66. Zyuban V (2000) Inherently lower-power high performance super scalar architectures. PhD thesis, Department of Computer Science and Engineering, University of Notre Dame
67. Papers in the special issue of IBM Journal of Research and Development, March/May 2002
68. Zyuban V, Brooks D, Srinivasan V, Gschwind M, Bose P, Strenski P, Emma P (2004) Integrated analysis of power and performance for pipelined microprocessors. IEEE T Comput 53(8):1004–1016
69. Zyuban V, Kogge P (2000) Optimization of high-performance superscalar architectures for energy efficiency. In: Proceedings of the IEEE symposium on low power electronics and design, ACM, New York

## PRAM (Parallel Random Access Machines)

JOSEPH F. JAJA

University of Maryland, College Park, MD, USA

### Definition

The Parallel Random Access Machine (PRAM) is an abstract model for parallel computation which assumes that all the processors operate synchronously under a single clock and are able to randomly access a large shared memory. In particular, a processor can execute an arithmetic, logic, or memory access operation within a single clock cycle.

### Discussion

#### Introduction

Parallel Random Access Machines (PRAMs) were introduced in the late 1970s as a natural generalization to parallel computation of the *Random Access Machine* (RAM) model. The RAM model is widely used as the basis for designing and analyzing sequential algorithms. The PRAM model assumes the presence of a number of processors, each identified by a unique id, which have access to a single unbounded shared memory. The processors operate synchronously under a single clock such that each processor can execute an arithmetic or logic operation or a memory access operation within a single clock cycle. In general, each processor under the PRAM model can execute its own program, which can be stored in some type of a private program memory. However, almost all the known PRAM algorithms are of the SPMD (Single Program Multiple Data) type in which a single program is executed by all the processors such that an instruction can refer to the id of the processor, which is responsible for executing the corresponding instruction. A processor may or may not be active during any given clock cycle. In fact, most of the

PRAM algorithms are of the SIMD (Single Instruction Multiple Data) type in which, during each cycle, a processor is either idle or executing the same operation as the remaining active processors.

As a simple example of a PRAM algorithm, consider the computation of the sum  $S$  of the elements of an array  $A[1 : n]$  using an  $n$ -processor PRAM, where the processors are indexed by  $pid = 1, 2, \dots, n$ . A PRAM algorithm can be organized as a balanced binary tree with  $n$  leaves such that each internal node represents the sum computation applied to the values available at the children. The PRAM algorithm proceeds level by level, executing all the computations at each level in a single parallel step. Hence the processors complete the process in  $\lceil \log n \rceil$  parallel steps. The algorithm is illustrated in Fig. 1.

The PRAM algorithm written in the SPMD style is given next where for simplicity  $n = 2^k$  for some positive integer  $k$ . The array  $B[1 : n]$  is used to store the intermediate results.

---

**Algorithm 1** PRAM SUM Algorithm

---

**Input:** An array  $A$  of size  $n = 2^k$  stored in shared memory.

**Output:** The sum of the elements of  $A$ .

**begin**

```

 $B[pid] = A[pid]$
 $d = n$
for $h = 1$ to k do
 $d = \frac{d}{2}$
 if $pid \leq d$ then
 $B[pid] = B[2pid - 1] + B[2pid]$
 end if
 end for
 return($B[1]$)
end

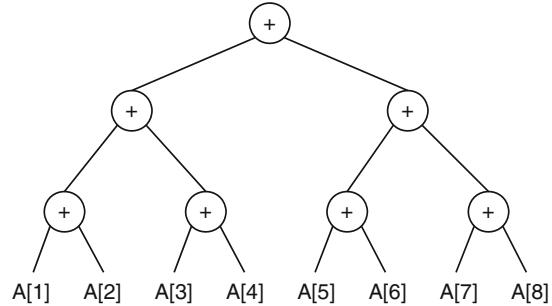
```

---

Given that the processors can simultaneously access the shared memory within the same clock cycle, several variants of the PRAM model exist depending on the assumptions made for simultaneous access to the same memory location.

- **Exclusive Read Exclusive Write (EREW) PRAM.**

No simultaneous accesses to the same location are allowed under the EREW PRAM model.



**PRAM (Parallel Random Access Machines).** Fig. 1 A Balanced Binary Tree for the Sum Computation. The PRAM algorithm proceeds from the leaves to the root while executing the operations at each level within a single clock cycle

- **Concurrent Read Exclusive Write (CREW) PRAM.** Simultaneous read accesses to the same location are allowed but no concurrent write operations to the same location are permitted.
- **Concurrent Read Concurrent Write (CRCW) PRAM.** Simultaneous read or write operations to the same memory location are allowed. This model has several subversions depending on how the concurrent write operations to the same location are resolved. The **Common CRCW PRAM** model assumes that the processors are writing the same value, while the **Arbitrary CRCW PRAM** model assumes that one of the processors attempting a concurrent write into the same location succeeds. The **Priority CRCW** assumes that the indices of the processors are linearly ordered, and allows the processor with the highest priority to succeed in case of a concurrent write.

The different assumptions on the concurrent accesses to a single location lead to parallel algorithms that can differ significantly in performance. Consider, for example, the problem of determining whether all the entries of a binary array  $M[1 : n]$  are all equal to 1. The output  $A$  of this computation is in fact the logical AND of all the entries in  $M$ . An  $n$ -processor Common CRCW PRAM algorithm can perform this computation as follows. Initially,  $A$  is set equal to 1. For all  $1 \leq pid \leq n$ , processor  $pid$  tests whether the entry  $M[pid] = 0$ , and in the affirmative, executes the operation  $A = 0$ . Clearly this algorithm correctly computes the AND of the  $n$  elements

of  $M$  in constant time on the Common CRCW PRAM. However, it can be shown, under a very general model, that a CREW PRAM will require  $\Omega(\log n)$  parallel steps regardless of the number of processors available. On the other hand, the computational gap between the various versions of the PRAM model is limited in the sense that the strongest  $p$ -processor PRAM model, namely the priority CRCW from our list above, can be simulated by a  $p$ -processor EREW (weakest PRAM model) with a slowdown of a factor of  $O(\log p)$ .

From a theoretical perspective, the design of a PRAM algorithm amounts to the development of a strategy that achieves the highest level of parallelism, that is, the smallest number of parallel steps, using a “reasonable” number of processors. Note that the number of processors can depend on the input size. In particular, the class of “well-parallelizable” problems under the PRAM model can be defined as the class of the problems that can be solved in polylogarithmic number of parallel steps (i.e.,  $O(\log^k n)$  for some fixed constant  $k$ ), using a polynomial number of processors. Such a class is referred to as the NC complexity class, which has been related to the circuits model used in traditional computational complexity. See the bibliographic notes for references to related work.

## Complexity Measures and Work-Time Framework

A PRAM algorithm can be evaluated by several complexity measures, where a complexity measure is a worst case asymptotic estimate of the performance as a function of the input length  $n$ . Given a  $p$ -processor PRAM algorithm to solve a problem with input size  $n$ , let  $T_P(p, n)$  be the number of parallel steps used by the algorithm. The optimal (or often the best known) sequential complexity is denoted by  $T_S(n)$ . The **speedup** is defined to be

$$\frac{T_S(n)}{T_P(p, n)},$$

which represents the speedup of the parallel algorithm *relative to the best sequential algorithm*. Clearly the best possible speedup is  $\Theta(p)$ . On the other hand, the **relative speedup** is defined to be

$$\frac{T_P(1, n)}{T_P(p, n)},$$

which refers to the speedup achieved by the algorithm with  $p$  processors relative to the same algorithm running with one processor.

Another related measure is the **relative efficiency** of a PRAM algorithm defined as the ratio

$$\frac{T_P(1, n)}{p T_P(p, n)}.$$

The two PRAM algorithms presented earlier assume the presence of  $n$  processors, where  $n$  is the input size. To derive the complexity measures defined above, these algorithms can be mapped into  $p$ -processor ( $p < n$ ) PRAM algorithms by distributing the concurrent operations carried out at each parallel step as evenly among the  $p$  processors as possible. In particular, the SUM algorithm can be executed in

$$\left\lceil \frac{n}{p} \right\rceil + \left\lceil \frac{n/2}{p} \right\rceil + \dots + 1 = O\left(\frac{n}{p} + \log n\right)$$

parallel time on a  $p$ -processor PRAM algorithm. Hence the speedup is  $\Theta(p)$  whenever  $p = O(n/\log n)$ . However, writing the corresponding algorithm for each processor (using processor ids) is in general a tedious process that distracts from the simplicity of the PRAM model.

An alternative is to describe a PRAM algorithm in the so-called **Work-Time** or simply **WT** framework, which is also related to the **data parallel** paradigm. The algorithm does not assume any specific number of processors, nor does it refer to any processor index, but rather it is expressed as a sequence of steps, where each step is either a typical sequential operation or a set of concurrent operations that can be executed in parallel. A set of parallel operations is specified by the pseudo-instruction **par do** or **forall**. For example, the previous PRAM SUM algorithm can be expressed in this framework as shown in Algorithm 2.

Under the WT framework, the complexity of a PRAM algorithm can be measured by two parameters – **work**  $W(n)$  and **time**  $T_P(n)$ . The work  $W(n)$  is the *total* number of operations required by the algorithm as a function of  $n$  and the time  $T_P(n)$  is defined as the number of parallel steps needed by the algorithm assuming an unlimited number of processors. The corresponding functions for the SUM algorithm are  $W(n) = O(n)$  and  $T_P(n) = O(\log n)$ , respectively.

**Algorithm 2** Parallel SUM Algorithm in the WT Framework

**Input:** An array  $A$  of size  $n = 2^k$  residing in memory.

**Output:** The sum of the elements of  $A$ .

**begin**
 $d = n$ 
**for all**  $1 \leq i \leq n$  **pardo**
 $B[i] = A[i]$ 
**end for**
**for**  $h = 1$  to  $k$  **do**
 $d = \frac{d}{2}$ 
**for all**  $1 \leq i \leq d$  **pardo**
 $B[i] = B[2i - 1] + B[2i]$ 
**end for**
**end for**
**return**( $B[1]$ )

**end**

In general, a WT-parallel algorithm with complexity  $W(n)$  and  $T_P(n)$  can be simulated on a  $p$ -processor PRAM using Brent's scheduling principle (see bibliographic notes) in time

$$T_P(p, n) = O\left(\frac{W(n)}{p} + T_P(n)\right)$$

The algorithm is called **work optimal** if  $W(n) = \Theta(T_S(n))$ . In this case, the corresponding  $p$ -processor PRAM algorithm achieves optimal speedup as long as  $p = O\left(\frac{T_S(n)}{T_P(n)}\right)$ .

Under the WT framework, a typical goal is to develop an algorithm that achieves the fastest parallel time among the work-optimal algorithms. That is, the main goal to develop a work-optimal algorithm that is as fast as possible.

## Basic PRAM Techniques

Basic PRAM techniques are illustrated through the description of PRAM algorithms for the following computations: matrix multiplication, prefix sums or scan, list ranking, fractional independent set, and computing the maximum. The WT framework is used to express these algorithms.

### Matrix Multiplication

Given two matrices  $A$  and  $B$  of dimensions  $m \times n$  and  $n \times q$ , respectively, the product  $C = AB$  is a matrix of

dimension  $m \times q$  such that

$$C[i, j] = \sum_{k=1}^n A[i, k]B[k, j], 1 \leq i \leq m, 1 \leq j \leq q.$$

Computing the matrix  $C$  on the PRAM model is quite simple since both matrices  $A$  and  $B$  are initially stored in shared memory, and their entries can each be accessed randomly within a single clock cycle. The algorithm in the WT framework amounts to computing in parallel all the products  $A[i, k]B[k, j]$ , for  $1 \leq i \leq m$ ,  $1 \leq k \leq n$ , and  $1 \leq j \leq q$ . The PRAM SUM algorithm can then be used to compute all the entries  $C[i, j]$  in parallel,  $1 \leq i \leq m$  and  $1 \leq j \leq q$ . Thus, the parallel complexity of the resulting algorithm is  $T_P = O(\log n)$  and the total work is  $W = O(nmq)$ .

This algorithm is based on the standard sequential algorithm. Should the initial sequential algorithm be one of the faster sequential matrix multiplication algorithm, the corresponding PRAM algorithm will then run in logarithmic parallel time using the same number of operations as the initial algorithm.

### Prefix Sums or Scan

Given a set of elements stored in an array  $A[1 : n]$  and a binary associative operation  $\otimes$ , the prefix sums of  $A$  consist of the  $n$  partial sums defined by

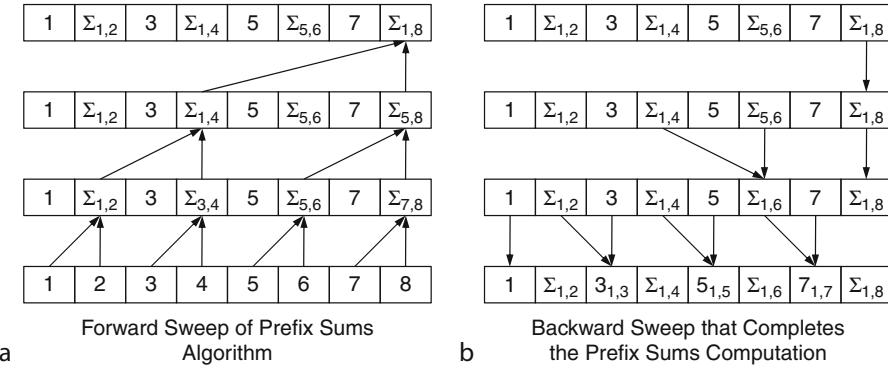
$$PS[i] = A[1] \otimes A[2] \otimes \dots \otimes A[i], 1 \leq i \leq n$$

The straightforward sequential algorithm computes  $PS[i] = PS[i - 1] \otimes A[i]$  for  $2 \leq i \leq n$  starting with  $PS[1] = A[1]$ , and hence the sequential complexity is  $T_S(n) = \Theta(n)$ .

A simple fast PRAM algorithm can be designed using a balanced binary tree built on top of the  $n$  elements of  $A$ . The algorithm consists of a forward sweep through the tree in a way similar to that carried out by the SUM algorithm. A backward sweep will then compute the prefix sums of the set of elements available at each level. A recursive version is described in Algorithm 3.

The algorithm is illustrated in Fig. 2, where the left part illustrates the forward sweep and the right part illustrates the backward sweep.

Since the algorithm involves a forward and a backward sweep through a balanced binary tree of height  $O(\log n)$ , the parallel complexity of the algorithm satisfies  $T_P(n) = O(\log n)$ . Also, since the number of



**PRAM (Parallel Random Access Machines).** Fig. 2 Computation of prefix sums through a forward sweep and a backward sweep of the balanced binary tree algorithm. The notation  $\Sigma_{ij}$  represents the sum of elements from  $A[i]$  through  $A[j]$ , and at each level an entry with two arrows pointing to it represent a sum operation

---

### Algorithm 3 Prefix Sums Algorithm

---

**Input:** An array  $A$  of size  $n = 2^k$  stored in shared memory.

**Output:** The prefix sums of  $A$  stored in the array  $PS$ .

**begin**

**if**  $n = 1$  **then**

**return**( $PS[1] = A[1]$ )

**end if**

**for all**  $1 \leq i \leq n/2$  **par do**

$Y[i] = A[2i - 1] \otimes A[2i]$

**end for**

  Recursively compute the prefix sums of  $Y[1 : n/2]$  in place

**for all**  $1 \leq i \leq n$  **par do**

*i even*: set  $PS[i] = Y[i/2]$

*i = 1*: set  $PS[1] = A[1]$

*else*: set  $PS[i] = Y[(i - 1)/2] \otimes A[i]$

**end for**

**end**

---

internal nodes of a binary tree on  $n$  leaves is  $n - 1$ , the total work is clearly  $W(n) = O(n)$ , and hence this algorithm is work optimal. It follows that a  $p$  processor PRAM can compute the prefix sums of  $n$  elements in time  $T_P(p, n) = O\left(\frac{n}{p} + \log n\right)$ .

### List Ranking

Consider a linked list  $L$  of  $n$  nodes whose successor relationship is represented by an array  $S[1 : n]$  such that  $S[i]$  is equal to the index of the successor node of  $i$ .

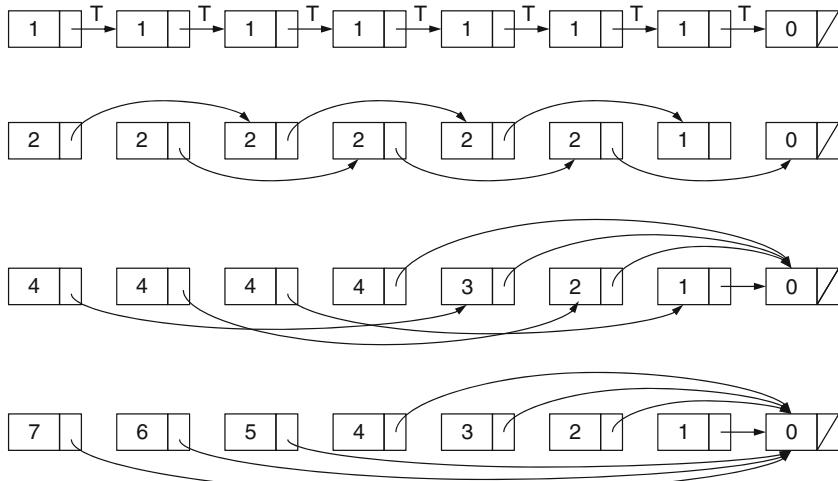
For the last node  $k$ , its successor is denoted by  $S[k] = 0$ . The list ranking problem is to determine for each node  $i$  the distance  $R[i]$  of the node  $i$  to the end of the list. While an optimal sequential algorithm is relatively straightforward – start by inverting the list and proceed to compute the ranks incrementally following the predecessor links – the problem may seem at first sight to be quite difficult to parallelize. It turns out that a fast PRAM algorithm can be developed through the introduction of the *pointer jumping* technique as illustrated in Algorithm 4. Note that  $T$  is used for the intermediate manipulation of the pointers, while the initial pointers stored in  $S$  remain intact.

The list ranking algorithm is illustrated in Fig. 3, where the pointer jumping technique is applied on each node  $i$  satisfying  $T[i] \neq 0$  and  $T[T[i]] \neq 0$ .

Since the execution of the last parallel loop involves replacing the successor by its successor, the distance between a node and its successor doubles after each iteration and hence the  $T$  pointer of each node will point to the last node after  $\lceil \log n \rceil$  iterations. Therefore, the algorithm terminates correctly after  $O(\log n)$  iterations, and each iteration involves  $O(n)$  operations. The algorithm has parallel complexity  $O(\log n)$  using  $O(n \log n)$  total number of operations. The work can be made optimal but the corresponding PRAM algorithm is more complex.

### Fractional Independent Set

The purpose of introducing the next problem is to illustrate the use of randomization in the design of PRAM



**PRAM (Parallel Random Access Machines).** Fig. 3 Application of the list ranking algorithm on a list with eight elements

---

#### Algorithm 4 List Ranking Algorithm

---

**Input:** An array  $S$  of size  $n$  representing the successor relationship of a linked list.

**Output:** The distance  $R[i]$  of node  $i$  to the end of the list,  $1 \leq i \leq n$ .

**begin**

**for all**  $1 \leq i \leq n$  **pardo**

**if**  $S[i] \neq 0$  **then**  $R[i] = 1$  **else**  $R[i] = 0$

**end for**

**for all**  $1 \leq i \leq n$  **pardo**

$T[i] = S[i]$

**end for**

**repeat**  $\lceil \log n \rceil$  **times do**

        {

**for all**  $1 \leq i \leq n$  **pardo**

**if**  $T[i] \neq 0$  **and**  $T[T[i]] \neq 0$  **then**

                {  $R[i] = R[i] + R[T[i]]$ ;  $T[i] = T[T[i]]$  }

**end for**

        }

**end**

---

algorithms to break symmetry. Given a directed cycle  $C = \langle v_1, v_2, \dots, v_n \rangle$ , a fractional independent set is a subset  $U$  of the vertices such that: (i)  $U$  is an independent set, that is, no two vertices in  $U$  are connected by a directed edge; and (ii) the size of  $U$  is a constant fraction of the size of  $V$ . The fractional independent set problem is to determine such a set.

It is trivial to develop a sequential algorithm to solve this problem. Starting from any vertex, place every other vertex in  $U$ . A simple and fast PRAM algorithm can be designed using randomization as follows. Each vertex, in parallel, is randomly assigned a label 1 or 0 with equal probability. Clearly, the expected number of vertices of each label is  $n/2$ . However, this does not necessarily solve the problem since there is no guarantee that the vertices with the same label form an independent set. To ensure that this is indeed the case, another parallel step is carried out which involves changing the label of a vertex to 0 if the labels of this vertex and its successor are both equal to 1. The remaining vertices of label 1 are now guaranteed to form an independent set. It can be shown that, with high probability, the size of such an independent set is a constant fraction of the size of the original vertex set  $V$ .

On the PRAM model, this algorithm runs in  $O(1)$  parallel time using  $O(n)$  operations. It is worth noticing that this algorithm can in particular be used to make the list ranking algorithm more efficient (i.e., using a total number of operations which is asymptotically less than  $n \log n$ ).

#### Superfast Maximum Algorithm

The algorithm presented in this section illustrates an important PRAM technique, called *accelerated cascading*, in combining two strategies. The first amounts to a work optimal algorithm (possibly a sequential

**Algorithm 5** Randomized Fractional Independent Set Algorithm

**Input:** A directed cycle with a vertex set  $V$  whose arcs are specified by an array  $S[1 : n]$ , i.e.,  $\langle i, S[i] \rangle$  is an arc.

**Output:** A fractional independent set  $U \subset V$ .

**begin**

**for all**  $v \in V$  **pardo**

Randomly assign  $label(v) = 1$  or  $0$  with equal probability

**if**  $label(v) = 1$  and  $label(S[v]) = 1$  **then**  $label(v) = 0$

**end for**

**return**( $U = \{v | label(v) = 1\}$ )

**end**

algorithm) to reduce the size of the problem below a certain threshold. The second strategy uses a very fast PRAM algorithm that involves a nonoptimal number of operations. The maximum algorithm will be used to illustrate such a technique and to also demonstrate the extra power of the PRAM model when concurrent write operations are allowed.

The PRAM strategy introduced earlier to compute the sum of  $n$  elements can be used to compute the maximum of  $n$  elements, resulting in the parallel time complexity  $T_P(n) = O(\log n)$  and total work of  $W(n) = O(n)$ . However, it is possible to develop a *constant time* algorithm on the Common CRCW PRAM model as illustrated in **Algorithm 6**.

**Algorithm 6** Constant Time Maximum Algorithm

**Input:** An array  $A[1 : n]$  consisting of  $n$  distinct elements

**Output:** A Boolean array  $M[1 : n]$  such that  $M[i] = 1$  if, and only if,  $A[i]$  is the maximum element

**begin**

**for all**  $1 \leq i, j \leq n$  **pardo**

**if**  $A[i] \geq A[j]$  **then**  $B[i, j] = 1$  **else**  $B[i, j] = 0$

**end for**

**for all**  $1 \leq i \leq n$  **pardo**

$M[i] = \bigwedge_{j=1}^n B[i, j]$

**end for**

**end**

The Boolean AND of  $n$  binary variables can be performed in  $O(1)$  parallel steps using  $O(n)$  operations on

the Common CRCW PRAM. Therefore, the above algorithm achieves constant parallel time but uses  $O(n^2)$  operations, and thus it is extremely inefficient. To remedy this problem, and still achieve faster than  $O(\log n)$  parallel time, a doubly logarithmic depth tree is used instead of the balanced binary tree. Essentially, the root of a doubly logarithmic depth tree has  $\Omega(\sqrt{n})$  children, and each subtree is defined similarly. It is not hard to see that such a tree has height  $O(\log \log n)$ , where  $n$  is the number of leaves. Using the doubly logarithmic depth tree and the constant time maximum algorithm at each node of the tree, the maximum can be computed in  $O(\log \log n)$  time using  $O(n \log \log n)$  operations.

The accelerated cascading strategy is now used to turn this fast but nonoptimal work algorithm into a fast and work optimal algorithm as follows:

- Partition the array  $A$  into  $n/\log \log n$  blocks, such that each block contains approximately  $\log \log n$  elements and use the sequential algorithm to compute the maximum element in each block.
- Use the doubly logarithmic depth tree on the maxima computed in the first step.

The resulting algorithm has a parallel time complexity of  $O(\log \log n)$  using only  $O(n)$  operations, and hence is work optimal. Therefore, the maximum can be computed in  $O(\log \log n)$  parallel time using a work optimal strategy on the Common CRCW PRAM.

## Bibliographic Notes and Further Reading

The PRAM model was initially introduced through a number of papers, most notably in the papers by Fortune and Wyllie [2], Goldschlager [3], and Ladner and Fisher [9]. The Work–Time framework, which ties Brent’s scheduling principle [1] and the PRAM model, was first observed by Shiloach and Vishkin [11] and used extensively in the book by JaJa [6]. Related data parallel algorithms were introduced by Hillis and Steele [5]. Substantial work dealing with the design and analysis of PRAM algorithms and the theoretical underpinnings of the model has been carried out in the 1980s and 1990s. An early survey is the paper by Karp and Ramachandran [7], and an extensive coverage of the topic can be found in JaJa’s book [6].

An intriguing relationship exists between the PRAM model and the circuits model used in traditional computational complexity. The NC class was formally introduced by Pippenger [10] for circuits. An important related theoretical direction is based on the notion of P-completeness, which tries to shed light on problems that do not seem to be highly parallelizable under the PRAM model with polynomial numbers of processors. Interested reader can consult the reference [4] for a good overview of this topic.

Some recent efforts have been devoted to advocate the PRAM as a practical parallel computation model, both in terms of developing prototype hardware that supports the model and software that enables the writing of PRAM programs. The book of Keller, Kessler, and Traff [8] and the recent paper by Wen and Vishkin [12] are illustrative of such efforts.

## Bibliography

1. Brent R (1974) The parallel evaluation of general arithmetic expressions. *JACM* 21(2):201–208
2. Fortune S, Wyllie J (1978) Parallelism in random access machines. In: Proceedings of the tenth ACM symposium on theory of computing. San Diego, CA, pp 114–118
3. Goldschlager L (1978) A unified approach to models of synchronous parallel machines. In: Proceedings of the tenth ACM symposium on theory of computing. San Diego, CA, pp 89–94
4. Greenlaw R, Hoover HJ, Ruzzo WL (1995) Limits to Parallel Computation: P-Completeness Theory. In: Topics in parallel computation. Oxford University Press, Oxford
5. Hillis WD, Steele GL (1986) Data parallel algorithms. *Commun ACM* 29(12):1170–1183
6. JaJa J (1992) An introduction to parallel algorithms. Addison Wesley Publishing Co., Reading, MA
7. Karp RM, Ramachandran V (1990) Parallel algorithms for shared-memory machines. In: van Leeuwen J (ed) Handbook of theoretical computer science, North Holland, Amsterdam, The Netherlands, Chapter 17, pp 869–942
8. Keller J, Kessler C, Traff J (2001) Practical PRAM programming. Wiley, New York
9. Ladner R, Fisher M (1980) Parallel prefix computations. *JACM* 27(4):831–838
10. Pippenger N (1979) On simultaneous resource bounds. In: Proceedings twentieth annual IEEE symposium on foundations of computer science. San Juan, Puerto Rico, pp 307–311
11. Shiloach Y, Vishkin U (1982) An  $O(n^2 \log n)$  parallel max-flow algorithm. *J Algorithms* 3(2):128–146
12. Wen X, Vishkin U (2008) FPGA-based prototype of a PRAM-on-chip processor. In: Proceedings of the 2008 ACM conference on computing frontiers. Ischia, Italy, pp 55–66

## Preconditioners for Sparse Iterative Methods

ANSHUL GUPTA

IBM T.J. Watson Research Center, Yorktown Heights, NY, USA

### Synonyms

Linear equations solvers

### Definition

Iterative methods for solving sparse systems of linear equations are potentially less memory and computation intensive than direct methods, but often experience slow convergence or fail to converge at all. The robustness and the speed of Krylov subspace iterative methods is improved, often dramatically, by *preconditioning*. Preconditioning is a technique for transforming the original system of equations into one with an improved distribution (clustering) of eigenvalues so that the transformed system can be solved in fewer iterations. A key step in preconditioning a linear system  $Ax = b$  is to find a nonsingular *preconditioner* matrix  $M$  such that the inverse of  $M$  is as close to the inverse of  $A$  as possible and solving a system of the form  $Mz = r$  is significantly less expensive than solving  $Ax = b$ . The system is then solved by solving  $(M^{-1}A)x = M^{-1}b$ . This particular example shows what is known as *left preconditioning*. There are two other formulations, known as *right preconditioning* and *split preconditioning*. The basic concept, however, is the same. Other practical requirements for successful preconditioning are that the cost of computing  $M$  itself must be low and the memory required to compute and apply  $M$  must be significantly less than that for solving  $Ax = b$  via direct factorization.

### Discussion

Preconditioning methods are being actively researched and have been for a number of years. There are several classes of preconditioners; some are more amenable to being computed and applied in parallel than others. This chapter gives an overview of the generation (i.e., computing  $M$  in parallel) and application (i.e., solving a system of the form  $Mz = r$  in parallel) of the most commonly used parallel preconditioners.

While solving a sparse linear system  $Ax = b$  in parallel, the matrix  $A$  and the vectors  $x$  and  $b$  are typically partitioned. The partitions are assigned to tasks that are executed by individual processes or threads in a parallel processing environment. Both the creation and the application of a preconditioner in parallel is affected by the underlying partitioning of the data. A commonly used effective and natural way of partitioning the data involves partitioning the graph, of which the coefficient matrix is an adjacency matrix. Other than partitioning the problem for parallelization, the graph view of the matrix plays a useful role in many aspects of solving sparse linear systems. [Figure 1](#) illustrates a partitioning of the rows of a matrix among four tasks based on a four-way partitioning of its graph.

### Simple Preconditioners Based on Stationary Methods

Stationary iterative methods are relatively simple algorithms that start with an initial guess of the solution (like all iterative methods) and attempt to converge toward the actual solution by repeated application of a correction equation. The correction equation uses the current residual and a fixed (stationary) operator or matrix,

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| X |   |   |   |   |   |   |   |   | X | X  |    |    |    |    | X  |
|   | X | X |   |   |   |   | X |   |   |    | X  |    |    |    |    |
|   | X | X | X |   |   |   |   |   |   |    |    | X  |    |    |    |
|   |   | X |   |   |   |   |   | X | X |    |    |    |    |    | X  |
|   |   |   | X | X | X |   |   |   |   | X  |    |    |    |    |    |
|   |   |   | X | X | X |   |   |   |   |    | X  |    |    |    |    |
|   |   |   |   | X | X | X |   |   |   |    |    | X  | X  | X  |    |
|   |   |   |   | X |   |   | X |   | X |    |    |    |    |    |    |
|   |   |   |   |   | X |   |   | X |   |    |    | X  | X  |    |    |
|   |   |   |   |   |   | X |   |   | X |    |    |    | X  |    |    |
|   |   |   |   |   |   |   | X | X |   |    |    |    | X  |    |    |
|   |   |   |   |   |   |   |   | X | X | X  |    |    |    |    | X  |
|   |   |   |   |   |   |   |   |   | X |    |    |    |    |    | X  |
|   |   |   |   |   |   |   |   |   |   | X  |    |    |    |    | X  |
|   |   |   |   |   |   |   |   |   |   |    | X  |    |    |    | X  |
|   |   |   |   |   |   |   |   |   |   |    |    | X  |    |    | X  |

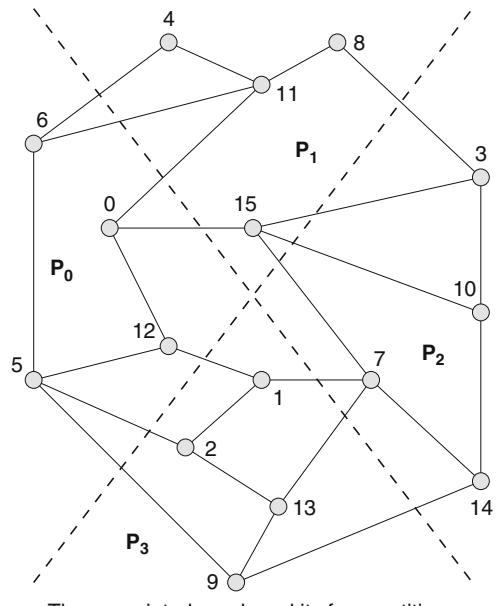
a A  $16 \times 16$  symmetric sparse matrix

which is an approximation of the original coefficient matrix. While stationary iterative methods themselves have poor convergence properties, the approximating matrix can serve as a preconditioner for Krylov subspace methods.

### Jacobi and Block-Jacobi Preconditioners

One of the simplest preconditioners is the point-Jacobi preconditioner, which is nothing but the diagonal  $D$  of the matrix  $A$  of coefficients. Applying the preconditioner in parallel is straightforward. It simply involves division with the entries (or multiplication with their inverses) of the part of the diagonal corresponding to the portion of the matrix that each thread or process is responsible for. In fact, scaling the coefficient matrix by the diagonal so that the scaled matrix has all 1's on the diagonal is equivalent to Jacobi preconditioning.

A block-Jacobi preconditioner is made up of nonoverlapping square diagonal blocks of the coefficient matrix. These blocks may be of the same or different sizes. These blocks are usually factored or inverted (independently, in parallel) during the preconditioner construction phase so that the preconditioner can be applied inexpensively during the Krylov solver's iterations.



b The associated graph and its four partitions

**Preconditioners for Sparse Iterative Methods.** [Fig. 1](#) A  $16 \times 16$  sparse matrix with symmetric structure and its associated graph partitioned among four tasks

### Gauss–Seidel Preconditioner

Let the coefficient matrix  $A$  be represented by a three-way splitting as  $L + D + U$ , where  $L$  is the strictly lower triangular part of  $A$ ,  $D$  is a diagonal matrix that consists of the principal diagonal of  $A$ , and  $U$  is the strictly upper triangular part of  $A$ . The Gauss–Seidel preconditioner is defined by

$$M = (D + L)D^{-1}(D + U). \quad (1)$$

A system  $Mz = r$  is then trivially solved as  $y = (D + L)^{-1}r$ ,  $w = Dy$ , and  $z = (D + U)^{-1}w$ . Thus, applying the Gauss–Seidel preconditioner in parallel involves solving a lower and an upper triangular system in parallel. In general, equation  $i$  of a lower triangular system can be solved for the  $i$ -th unknown when equations  $1 \dots i - 1$  have been solved. Similarly, equation  $i$  of an  $N \times N$  upper triangular system can be solved after equations  $i + 1 \dots N$  have been solved. Since the matrices  $L$  and  $U$  in our case are sparse, the  $i$ -th equation while computing  $y = (D + L)^{-1}r$  depends only on those unknowns that have nonzero coefficients in the  $i$ -th row of  $L$ . Similarly, the  $i$ -th equation while computing  $x = (D + U)^{-1}w$  depends only on those unknowns that have nonzero coefficients in the  $i$ -th row of  $U$ . As a result, while solving both these systems, multiple unknowns may be computed simultaneously – those that do not depend on any unsolved unknowns. This is an obvious source of parallelism. This parallelism can be maximized by reordering the rows and columns of  $A$  (and hence those of  $L$  and  $U$ ) in a way that maximizes the number of independent equations at each stage of the solution process.

[Figure 2](#) illustrates one such ordering, known as red–black ordering, that can be used to parallelize the application of the Gauss–Seidel preconditioner for a matrix arising from a finite-difference discretization. The vertices of the graph corresponding to the matrix are assigned colors such that no two neighboring vertices have the same color. All vertices and the corresponding rows and columns of the matrix are numbered first, followed by those of the other color. Assignment of matrix rows to tasks is based on a partitioning of the graph. With red–black ordering, each triangular solve is performed in two phases. During the lower triangular solve, first, all red unknowns are computed in parallel because they are all independent. After this step, all black unknowns can be computed. The order

is reversed during the upper triangular solve. On a distributed memory platform, each computation phase is followed by a communication phase. During a communication phase, each process communicates the values of the unknowns corresponding to graph vertices on the partition boundaries with its neighboring processes.

The idea of red–black ordering can be extended to general sparse matrices and their graphs, for which more than two colors may be required to ensure that no neighboring vertices have the same color. The triangular solves are then performed in as many parallel phases as the number of colors. A multicolored ordering with four colors is illustrated in [Fig. 3](#). For improved cache performance, block variants of multicolored ordering can be constructed by assigning colors to clusters of graph vertices and ensuring that no two clusters that have an edge connecting them are assigned the same color.

The reader is cautioned that often the convergence of a Krylov subspace method is sensitive to the ordering of matrix rows and columns. While red–black and multicolor orderings enhance parallelism, they may result in a deterioration of the convergence rate in some cases.

### SOR Preconditioner

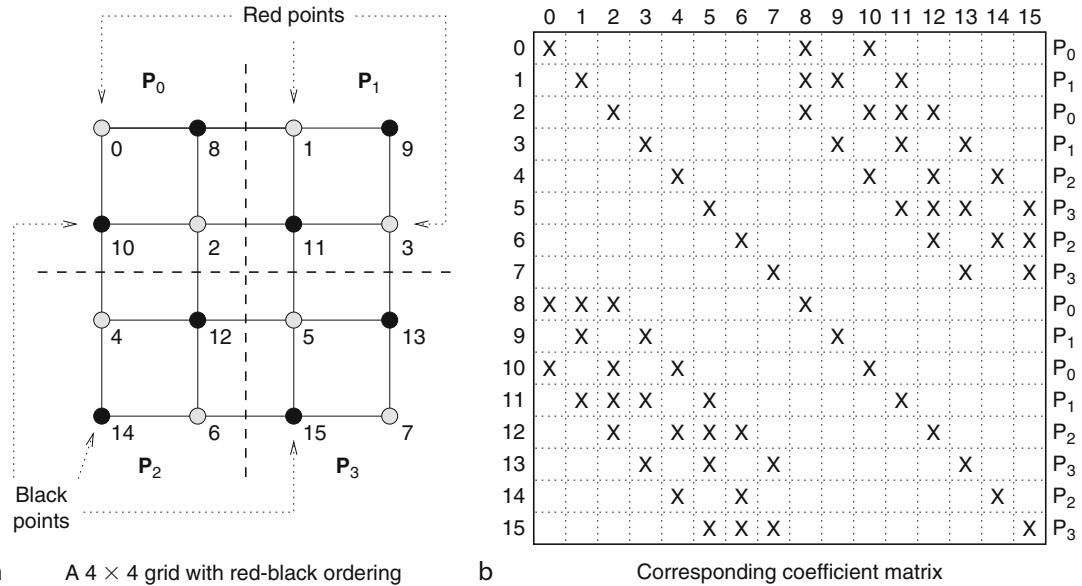
A significant increase in convergence rate may be obtained by a modification of [Eq. 1](#) as follows, with  $0 < \omega < 2$ :

$$M = \left( \frac{D}{\omega} + L \right) \frac{\omega D^{-1}}{2 - \omega} \left( \frac{D}{\omega} + U \right), \quad (2)$$

although determining an optimal value of  $\omega$  can be expensive. The preconditioner specified by [Eq. 2](#) is known as successive overrelaxation or SOR preconditioner. Its symmetric formulation, when  $L = U$ , is known as symmetric SOR or SSOR preconditioner. The issues in parallel application of the SOR preconditioner are identical to those for the Gauss–Seidel preconditioner.

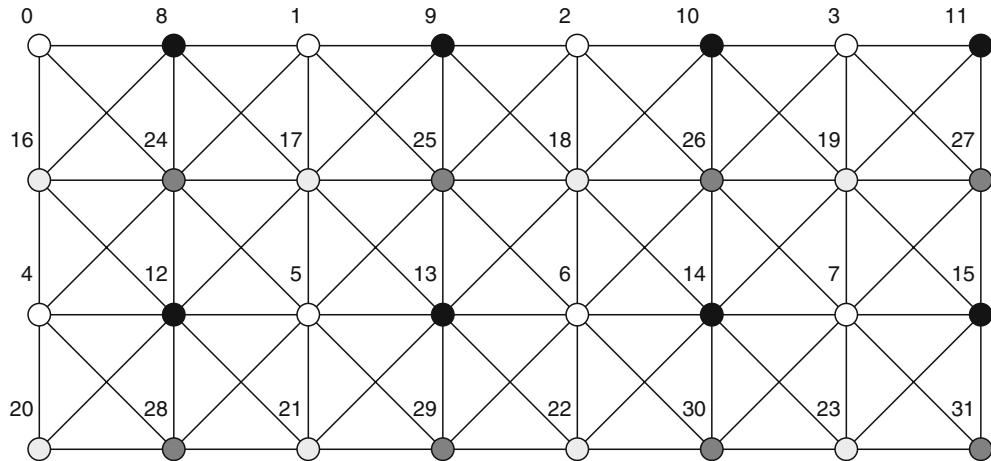
### Preconditioners Based on Incomplete Factorization

A class of highly effective but conceptually simple preconditioners is based on incomplete factorization methods. Recall that iterative methods are used in applications where a factorization of the form  $A = LU$  is not feasible because the triangular factor matrices  $L$  and  $U$

a A  $4 \times 4$  grid with red-black ordering

b Corresponding coefficient matrix

**Preconditioners for Sparse Iterative Methods.** Fig. 2 The sparse matrix corresponding to a  $4 \times 4$  finite-difference grid with red-black ordering, partitioned among four parallel tasks



**Preconditioners for Sparse Iterative Methods.** Fig. 3 Multicolored ordering of a hypothetical finite-element graph using four colors

are much denser than  $A$ , and therefore, too expensive to compute and store. The idea behind incomplete factorization is to perform a factorization of  $A$  along the lines of a regular Gaussian elimination or Cholesky factorization, but dropping a large proportion of nonzero entries from the triangular factors along the way. Depending on the underlying factorization method, incomplete factorization is referred to as ILU (incomplete LU) or IC (incomplete Cholesky) factorization. Due to

dropping, the resulting triangular factors  $\tilde{L}$  and  $\tilde{U}$  are much sparser than  $L$  and  $U$  and are computed with significantly less computing effort. Entries are chosen for dropping using some criteria that strive to keep the inverse of  $M = \tilde{L}\tilde{U}$  as close to the inverse of  $A$  as possible. Devising effective dropping criteria has been an active area of research. Incomplete factorization methods can be broadly classified as follows based on the dropping criteria.

### Static-Pattern Incomplete Factorization

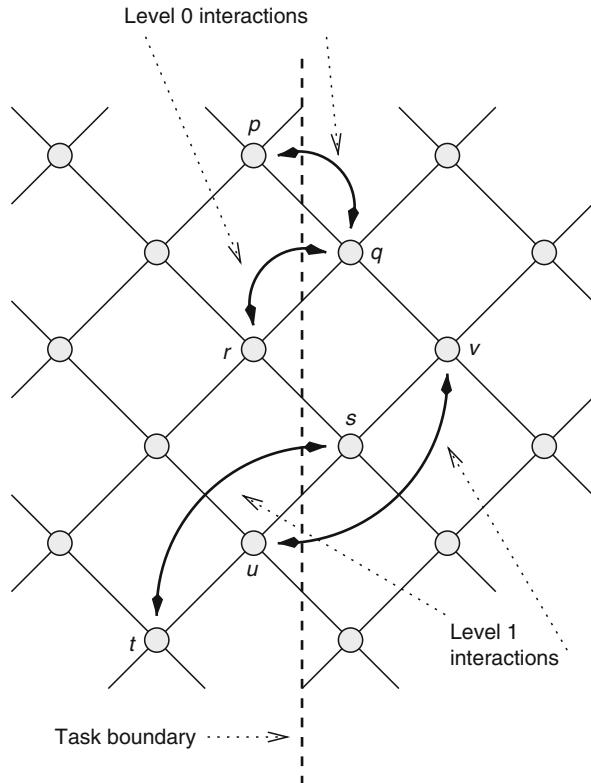
A static-pattern incomplete factorization method is one in which the locations of the entries that are kept in the factors and those that are dropped are determined a priori, based only on the structure of  $A$ . The simplest form of incomplete factorizations are ILU(0) and IC(0), in which the structure of  $\tilde{L} + \tilde{U}$  is identical to structure of  $A$ ; i.e., only those factor entries whose locations coincide with those of nonzero entries in the original matrix are saved. A generalization of static-pattern incomplete factorization is a level- $\kappa$  incomplete factorization, referred to as ILU( $\kappa$ ) or IC( $\kappa$ ) factorization in the literature. The structure of a level- $\kappa$  incomplete factorization is computed symbolically as follows. Initially,  $level(i,j) = 0$  if  $a_{ij} \neq 0$ ; otherwise,  $level(i,j) = \infty$ . This is followed by an emulation of factorization where each numerical update step of the form  $a_{ij} = a_{ij} - a_{ik} \cdot a_{kj}$  is replaced by updating  $level(i,j) = \min(level(i,j), level(i,k) + level(k,j) + 1)$ . During the subsequent numerical factorization phases, entries in locations with a level greater than  $\kappa$  are dropped.

In graph terms, upon the completion of symbolic factorization,  $level(i,j)$  is the length of the shortest path between vertices  $i$  and  $j - 1$ . During the computation and application of an ILU( $\kappa$ ) or IC( $\kappa$ ) preconditioner, the computation corresponding to a vertex in the graph requires data associated with vertices that are up to  $\kappa + 1$  edges away from it. For example, in the graph shown in Fig. 4, data exchange is required among vertex pairs  $(p,q)$  and  $(q,r)$  for  $\kappa = 0$ . For  $\kappa = 1$ , additional exchanges among vertex pairs such as  $(s,t)$  and  $(u,v)$  are required. The figure also illustrates that in a parallel environment, data associated with  $\kappa+1$  layers of vertices adjacent to a partition boundary needs to be exchanged with a neighboring task.

Both the computation per vertex of the graph and the data exchange overhead per task in each iteration of a Krylov solver using a level- $\kappa$  incomplete factorization preconditioner increase as  $\kappa$  increases. On the other hand, the overall number of iterations typically declines. The optimum value of  $\kappa$  is problem dependent.

### Threshold-Based Incomplete Factorization

Although it permits a relatively easy and fast parallel implementation, static-pattern incomplete factorization is robust for a few classes of problems only, including those with diagonally dominant coefficient



**Preconditioners for Sparse Iterative Methods. Fig. 4**

Illustration of data exchange across a task boundary when level 0 and level 1 fill is permitted in incomplete factorization

matrices. It can, and often does delete fill entries of large magnitudes that do not happen to be located in the predetermined locations. The resulting large error can make the preconditioner ineffective. Threshold-based incomplete factorization rectifies this problem by dropping entries from the factors as they are computed. Regardless of their locations, entries greater in magnitude than a user-defined threshold  $\tau$  are kept and the others are dropped. Typically, a second threshold  $\gamma$  is also used to limit the factors to a predetermined size. If  $n_i$  is the number of nonzeros in row (or column)  $i$  of the coefficient matrix, then at most  $\gamma n_i$  entries (those with the largest magnitudes) are permitted in row (or column)  $i$  of the incomplete factor.

Successful and scalable parallel implementations of threshold-based incomplete factorization preconditioning use graph partitioning and graph coloring for balancing computation and minimizing communication

among parallel tasks. The use of these two techniques in the context of sparse matrix computations has already been discussed earlier. Graph partitioning enables parallel tasks to independently compute and apply (i.e., perform forward and back substitution) the preconditioner independently for matrix rows and columns corresponding to the internal vertices of the graph. An internal vertex and all its neighbors belong to the same partition. Graph coloring permits parallel incomplete factorization and forward and back substitution of matrix rows and columns corresponding to the boundary vertices. In the context of incomplete factorization, coloring is applied to a graph that includes only the boundary vertices (i.e., after the internal vertices have been eliminated) but includes the additional edges (fill-in) created as a result of the elimination of the internal vertices. The reason is that the dependencies among the rows and columns of the matrix are determined by all the nonzeros in the incomplete factors, both original and those created as a result of fill-in.

### Incomplete Factorization Based on Inverse-Norm Estimate

This is a relatively new class of incomplete factorization preconditioners in which the dropping criterion takes into account and seeks to minimize the growth of the norm of the inverse of the factors. These preconditioners have been shown to be more robust and effective than incomplete factorization with dropping based solely on the position or absolute value of the entries. The issues in the parallel generation and application of these preconditioners are very similar to those in threshold-based incomplete factorization – in both cases, the location of nonzeros in the factors cannot be determined a priori.

### Sparse Approximate Inverse Preconditioners

While the incomplete factorization preconditioners seek to compute  $\tilde{L}$  and  $\tilde{U}$  as approximations of the actual factors  $L$  and  $U$  of the coefficient matrix  $A$ , sparse approximate preconditioners seek to compute  $M^{-1}$  as an approximation to its inverse  $A^{-1}$ . The problem of computing  $M^{-1}$  is framed as the problem of minimizing the norm  $\|I - AM\|$  or  $\|I - MA\|$ . To support parallelism, these minimization problems can be reduced to

independent subproblems for computing the rows and columns of  $M^{-1}$ . Note that the actual inverse of a sparse  $A$  is dense, in general. It is therefore imperative that a number of entries be dropped in order to keep  $M^{-1}$  sparse. Just like incomplete factorization, the dropping can be structural (static), or based on values (dynamic), or both. In graph terms, while computing the rows or columns of  $M^{-1}$  in parallel, dropping is typically orchestrated in a way that confines the interaction to pairs of vertices that are either immediate neighbors or have short paths connecting them in the graph of  $A$ . Graph partitioning is used to facilitate load balance and minimize interaction among parallel tasks – both during the computation and the application of the preconditioner.

There are some important advantages to explicitly using an approximation of  $A^{-1}$  for preconditioning, rather than using  $A$ 's approximate factors. First, the preconditioner computation avoids the kind of breakdowns that are possible in incomplete factorization due to small or zero (or negative, in case of incomplete Cholesky) diagonals. Secondly, the application of the preconditioners involves a straightforward multiplication of a vector with the sparse matrix  $M^{-1}$ , which may be simpler and more easily parallelizable than the forward and back substitutions with  $\tilde{L}$  and  $\tilde{U}$ . However, just like  $\tilde{L}$  and  $\tilde{U}$ ,  $M^{-1}$  may be denser than  $A$ . Therefore, the graph of  $M^{-1}$  may have many more edges than the graph of  $A$  and multiplying a vector with  $M^{-1}$  may require more communication than multiplying a vector with  $A$ .

### Multigrid Preconditioners

Multigrid methods are a class of iterative algorithms for solving partial differential equations (PDEs) efficiently, often by exploiting more problem-specific information than a typical Krylov subspace method. Each iteration of a multigrid solver is a somewhat complex recursive procedure. In many applications, the effectiveness of a multigrid algorithm can be substantially enhanced by using it to precondition a Krylov subspace method rather than using it as the solver. This is done by replacing the preconditioning step of the Krylov subspace solver with one iteration of the multigrid algorithm; i.e., treating the approximate solution obtained by an iteration of the multigrid algorithm as the solution with respect to a hypothetical preconditioner matrix.

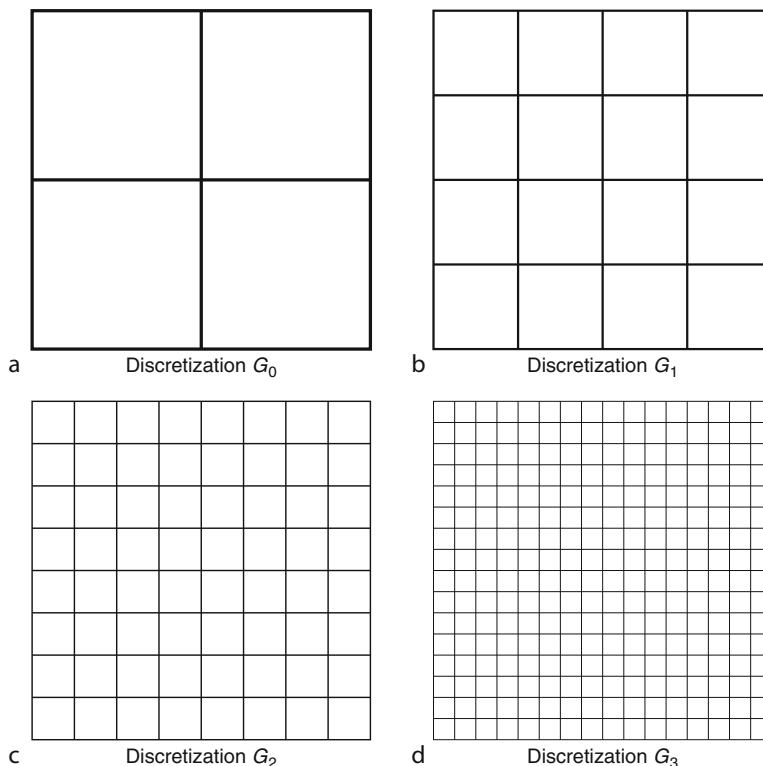
## Geometric Multigrid

Multigrid methods were originally designed for solving elliptic PDEs by discretizing them using a hierarchy of regular grids of varying degrees of fineness over the same domain. For example, consider a domain  $D$  and a sequence of successively finer discretizations  $G_0, G_1, \dots, G_m$ . Here  $G_0$  is the coarsest discretization and  $G_m$  is the finest discretization over which the eventual solution to the PDE is desired. Figure 5 shows a square domain with  $m = 3$ . As the figure shows, the grid points in  $G_i$  are a subset of the grid points in  $G_{i+1}$ . A simple formulation of multigrid would work as follows:

1. First, the linear system corresponding to discretization  $G_0$  is solved. Since  $G_0$  has a small number of points, the associated linear system is small and can be solved inexpensively by an appropriate direct or iterative method.
2. The solution at  $G_0$  is interpolated to obtain an initial guess of the solution of the system corresponding to  $G_1$ , which is four times larger. Among various ways of *interpolation* (also known as *prolongation*),

a simple one involves approximating the value of the solution at a point that is in  $G_1$  but not in  $G_0$  by the average of the values of its neighbors.

3. Starting with the initial guess obtained by interpolation, a few steps of *relaxation* (also referred to as *smoothing*) are used to refine the solution at  $G_1$ . Often, the relaxation steps are simply a few iterations of a relatively inexpensive stationary method such as Jacobi or Gauss–Siedel.
4. The process of relaxation and interpolation continues from  $G_i$  to  $G_{i+1}$ , until  $i + 1 = m$ . After the relaxation at  $G_m$ , a first approximation  $x_0^m$  to the solution  $x^m$  of the linear system  $A^m x^m = b^m$  corresponding to  $G_m$  is obtained. Successively more accurate approximations  $x_1^m, x_2^m, \dots$  are obtained by  $x_{i+1}^m = x_i^m + d_i^m$ , where  $d_i^m$  is obtained by solving  $Ad_i^m = r_i^m \equiv (b^m - Ax_i^m)$ .
5. The system  $Ad_i^m = r_i^m$  in the  $i$ -th multigrid iteration is solved by a recursive process, in which the residual  $r_i^m$  corresponding to  $G_m$  is projected on to  $G_{m-1}$  and so on. The process of *projection* (also known as *restriction*) is the reverse of interpolation. At the end



Preconditioners for Sparse Iterative Methods. Fig. 5 Successively finer discretizations of a domain

of the recursive projection steps, a relatively small system of equations  $Ad_i^0 = r_i^0$  corresponding to  $G_0$  is obtained, which is readily solved by an appropriate iterative or direct method.

6. The cycle of interpolation and relaxation is then repeated to obtain  $d_i^m$ .
7. The process is stopped after  $k$  multigrid iterations if  $r_k^m$  is smaller than a user-defined threshold.

When the multigrid method is used as a preconditioner, then instead of repeating the multigrid cycle  $k$  times to solve the problem, the approximate solution after one cycle is substituted as the solution with respect to the preconditioner inside a Krylov subspace algorithm. Whether multigrid is used as a solver or a preconditioner, the parallelization process is the same.

The first step in implementing a parallel multigrid procedure is to partition the domain among the tasks such that each task is assigned roughly the same number of points of the finest grid. For example, Fig. 6 shows the partitioning of the domain of Fig. 5 and its discretizations  $G_0-G_3$  into four parallel tasks. Unlike Figs. 5 and 6, the domain in a real problem may be irregular and the partitioning may not be trivial. The partitioning of the domain implicitly defines a partitioning of the grids at all levels of discretization. It is easily seen that

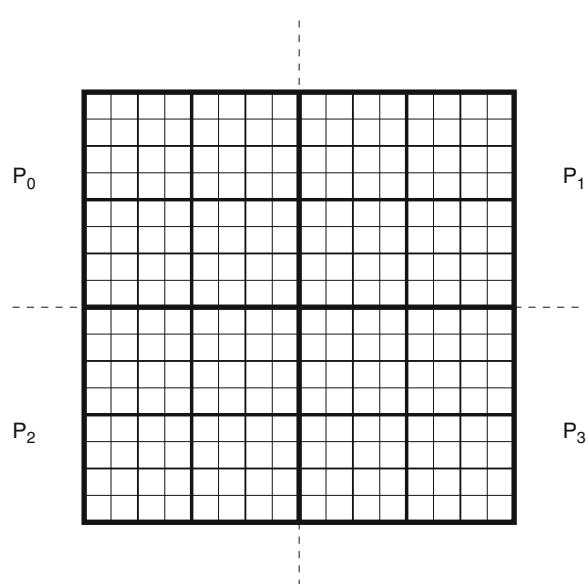
the parallel interpolation, relaxation, and projection at any grid level require a task to exchange information corresponding to the grid points along the partition boundary with its neighboring tasks. All computations corresponding to each partition's interior points can be performed independently.

### Algebraic Multigrid

The algebraic multigrid (AMG) method is a generalization of the hierarchical approach of the geometric multigrid method so that it is not dependent on the availability of the meshes used for discretizing a PDE, but can be used as a black-box solver for a given linear system of equations. In the geometric multigrid method, successively finer meshes are constructed by a geometric refining of the coarser meshes. On the other hand, the starting point for AMG is the final system of equations (analogous to the finest grid), from which successively smaller (coarser) systems are constructed. The coefficients of a coarse system in AMG are only algebraically related to the coefficients of the finer systems, which is in contrast to the geometric relationship between successive grids in geometric multigrid.

Just like geometric multigrid, an AMG method can be used either as a solver or a preconditioner. In either case, the method involves two phases. In the first set-up phase, the hierarchy of coarse systems is constructed from the original linear system and the prolongation and restriction operators are defined. The second solution phase consists of the prolongation, relaxation, and restriction cycles.

Efficient parallelization of AMG is much harder than that of geometric multigrid. As usual, the basis of parallelization is a good partitioning of the graph corresponding to the coefficient matrix and assigning the partitions to individual tasks. The solution phase, in principle, can then be parallelized with computation corresponding to the interior nodes remaining independent and that involving the nodes at or close to the boundary involving exchange of data with neighboring partitions. The boundary communication can be more involved than in the case of geometric multigrid because of the irregularity of the graph and the fact that the sets of interacting boundary nodes and the interaction pattern can be different for prolongation, relaxation, and restriction. However, the main difficulty in parallelizing AMG is in the set-up phase.



**Preconditioners for Sparse Iterative Methods. Fig. 6** The domain and discretizations  $G_0-G_3$  of Fig. 5 mapped onto four parallel tasks

Effective parallelization of the set-up phase is essential for the overall scalability of parallel AMG because this phase can account for up to one-fourth of the total execution time. The process of construction of a coarser linear system in the classical AMG approach relies on a notion of the “strength” of dependencies among coefficients. This process is not only sequential in nature, but also involves a highly nonlocal pattern of interaction between the vertices of the graph of the coefficient matrix. Therefore, the algorithm must be adapted for parallelization, which can make the convergence rate and per iteration cost of AMG dependent on the number of parallel tasks. Care must be taken to ensure that parallelization does not adversely affect the convergence rate or the iteration complexity of AMG. Typical approaches to parallelizing coarsening in AMG rely on decoupling the partitions, using parallel independent sets (similar to multicoloring described earlier), or performing subdomain blocking, which starts the coarsening at the partition boundaries and then proceeds to the interior nodes. Note that the coarsening scheme has an impact on the inter-task interaction during the prolongation and restriction steps of the solution phase.

When parallelizing AMG on large parallel machines, the number of parallel tasks may exceed the number of points in some of the grids at the coarsest levels. This situation requires special treatment. A commonly used work-around to this problem is agglomeration, in which neighboring domains are coalesced leaving some tasks idle during the processing of the coarsest levels. Another approach is to stop the coarsening when the number of coefficients per task becomes too small, which makes the behavior of the overall algorithm dependent on the number of parallel tasks.

### Stochastic Preconditioners

There has been a fair amount of research on algorithms for approximating the solution of linear systems based on random sampling of the coefficient matrix or on random walks in the graph corresponding to it. Most of these methods have been proven to work on limited classes of linear systems only, such as symmetric diagonally dominant systems. A few practical solvers have been developed recently by using some of these techniques for preconditioning Krylov subspace methods. An attractive property of these methods is that they are usually trivially parallelizable. The quest for scalable massively parallel sparse linear solvers may prompt

more active research into statistical techniques for preconditioning.

### Matrix-Free Methods and Physics-Based Preconditioners

Note that this chapter discusses preconditioners derived explicitly from the coefficient matrix  $A$  of the system  $Ax = b$  that needs to be solved. In some applications,  $A$  is never constructed explicitly to save time and storage. Instead, it is applied implicitly to compute the matrix-vector products required in the Krylov subspace solver. In some of these cases, preconditioning is also applied implicitly, or the knowledge of the physics of the application is utilized to construct the preconditioner, which cannot be derived from the coefficient matrix in a matrix-free method. Such preconditioners are called physics-based preconditioners. Due to the highly application-specific nature of matrix-free methods and physics-based preconditioners, these topics are not covered in further detail in this chapter.

### Related Entries

- [Graph Partitioning](#)
- [Linear Algebra Software](#)
- [Rapid Elliptic Solvers](#)

### Bibliographic Notes

The readers are referred to Saad’s book [11] and the survey by Benzi [1] for a fairly comprehensive introduction to various preconditioning techniques. These do not cover parallel preconditioners and may not included some of the most recent work in preconditioning. However, these are excellent resources for gaining an insight into the state of the art of preconditioning circa 2002.

Hysom and Pothen [7] and Karypis and Kumar [8] cover the fundamentals of scalable parallelization of incomplete factorization-based preconditioners. The work of Grote and Huckle [6] and Edmond Chow [3, 4] is the basis of modern parallel sparse approximate inverse preconditioners. Chow et al.’s survey [5] should give the readers a good overview of parallelization techniques for geometric and algebraic multigrid methods.

Last, but not the least, almost all parallel preconditioning techniques rely on effective parallel heuristics for two critical combinatorial problems – graph partitioning and graph coloring. The readers are referred

to papers by Karypis and Kumar [9, 10] and Bozdag et al. [2] for an overview of these.

## Bibliography

1. Benzi M (2002) Preconditioning techniques for large linear systems: a survey. *J Computat Phys* 182(2):418–477
2. Bozdag D, Gebremedhin AH, Manne F, Boman EG, Catalyurek UV (2008) A framework for scalable greedy coloring on distributed memory parallel computers. *J Parallel Distrib Comput* 68(4):515–535
3. Chow E (2000) A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM J Sci Comput* 21(5):1804–1822
4. Chow E (2001) Parallel implementation and practical use of sparse approximate inverse preconditioners with a priori sparsity patterns. *Int J High Perform Comput appl* 15(1):56–74
5. Chow E, Falgout RD, Hu JJ, Tuminaro RS, Yang UM (2006) A survey of parallelization techniques for multigrid solvers. In Heroux MA, Raghavan P, Simon HD (eds) *Parallel processing for scientific computing*. SIAM, Philadelphia
6. Grote MJ, Huckle T (1997) Parallel preconditioning with sparse approximate inverses. *SIAM J Sci Comput* 18(3): 838–853
7. Hysom D, Pothen A (2000) A scalable parallel algorithm for incomplete factor preconditioning. *SIAM J Sci Comput* 22(6): 2194–2215
8. Karypis G, Kumar V (1996) Parallel threshold-based ILU factorization. Technical report TR 96-061, Department of Computer Science, University of Minnesota, Minnesota
9. Karypis G, Kumar V (1997) ParMETIS: parallel graph partitioning and sparse matrix ordering library. Technical report TR 97-060, Department of Computer Science, University of Minnesota, Minnesota
10. Karypis G, Kumar V (1998) Parallel algorithms for multilevel graph partitioning and sparse matrix ordering. *J Parallel Distrib Comput* 48:71–95
11. Saad Y (2003) Iterative methods for sparse linear systems, 2nd edn. SIAM, Philadelphia

## Prefix

- Reduce and Scan
- Scan for Distributed Memory, Message-Passing Systems

## Prefix Reduction

- Reduce and Scan
- Scan for Distributed Memory, Message-Passing Systems

## Problem Architectures

- Computational sciences

## Process Algebras

ROCCO DE NICOLA  
Universita' di Firenze, Firenze, Italy

## Synonyms

Process calculi; Process description languages

## Definition

Process Algebras are mathematically rigorous languages with well-defined semantics that permit describing and verifying properties of concurrent communicating systems. They can be seen as models of processes, regarded as agents that act and interact continuously with other similar agents and with their common environment. The agents may be real-world objects (even people), or they may be artifacts, embodied perhaps in computer hardware or software systems. Many different approaches (operational, denotational, algebraic) are taken for describing the meaning of processes. However, the operational approach is the reference one. By relying on the so-called Structural Operational Semantics (SOS), labeled transition systems are built and composed by using the different operators of the many different process algebras. Behavioral equivalences are used to abstract from unwanted details and identify those systems that react similarly to external experiments.

## Introduction

The goal of software verification is to assure that developed programs fully satisfy all the expected requirements. Providing a formal semantics of programming languages is an essential step toward program verification. This activity has received much attention in the last 40 years. At the beginning the interest was mainly on sequential programs, then it turned also on concurrent program that can lead to subtle errors in very critical activities. Indeed, most computing systems today are concurrent and interactive.

Classically, the semantics of a sequential program has been defined as a function specifying the

induced input–output transformations. This setting becomes, however, much more complex when concurrent programs are considered because they exhibit nondeterministic behaviors. Nondeterminism arises from programs interaction and cannot be avoided. At least, not without sacrificing expressive power. Failures do matter, and choosing the wrong branch might result in an “undesirable situation.” Backtracking is usually not applicable because the control might be distributed. Controlling nondeterminism is very important. In sequential programming, it is just a matter of efficiency, in concurrent programming it is a matter of avoiding getting stuck in a wrong situation.

The approach based on process algebras has been very successful in providing formal semantics of concurrent systems and proving their properties. The success is witnessed by the Turing Award given to two of their pioneers and founding fathers: Tony Hoare and Robin Milner. Process algebras are mathematical models of processes, regarded as agents that act and interact continuously with other similar agents and with their common environment. Process algebras provide a number of constructors for system descriptions and are equipped with an operational semantics that describes systems evolution in terms of labeled transitions. Models and semantics are built by taking a compositional approach that permits describing the “meaning” of composite systems in terms of the meaning of their components.

Moreover, process algebras often come equipped with observational mechanisms that permit identifying (through behavioral equivalences) those systems that cannot be taken apart by external observations (*experiments* or *tests*). In some cases, process algebras have also algebraic characterizations in terms of equational axiom systems that exactly capture the relevant identifications induced by the behavioral operational semantics.

The basic component of a process algebra is its syntax as determined by the well-formed combination of operators and more elementary terms. The syntax of a process algebra is the set of rules that define the combinations of symbols that are considered to be correctly structured programs in that language. There are many approaches to providing a rigorous mathematical understanding of the semantics of syntactically correct process terms. The main ones are those also used for describing the semantics of sequential systems, namely, operational, denotational, and algebraic semantics.

An *operational semantics* models a program as a labeled transition system (LTS) that consists of a set of states, a set of transition labels and a transition relation. The states of the transition system are just process algebra terms, while the labels of the transitions between states represent the actions or the interactions that are possible from a given state and the state that is reached after the action is performed by means of visible and invisible actions. The operational semantics, as the name suggests, is relatively close to an abstract machine-based view of computation and might be considered as a mathematical formalization of some implementation strategy.

A *denotational semantics* maps a language to some abstract model such that the meaning/denotation (in the model) of any composite program is determinable directly from the meanings/denotations of its subcomponents. Usually, denotational semantics attempt to distance themselves from any specific implementation strategy, describing the language at a level intended to capture the “essential meaning” of a term.

An *algebraic semantics* is defined by a set of algebraic laws which implicitly capture the intended semantics of the constructs of the language under consideration. Instead of being derived theorems (as they would be in a denotational semantics or operational semantics), the laws are the basic axioms of an equational system, and process equivalence is defined in terms of what equalities can be proved using them. In some ways it is reasonable to regard an algebraic semantics as the most abstract kind of description of the semantics of a language.

There has been a huge amount of research work on process algebras carried out during the last 30 years that started with the introduction of CCS [31, 32], CSP [11], and ACP [6]. In spite of the many conceptual similarities, these process algebras have been developed starting from quite different viewpoints and have given rise to different approaches (for an overview see, e.g., [2]).

CCS takes the operational viewpoint as its cornerstone and abstracts from unwanted details introduced by the operational description by taking advantage of behavioral equivalences that allow one to identify those systems that are indistinguishable according to some observation criteria. The meaning of a CCS term is a labeled transition system factored by a notion of observational equivalence. CSP originated as the

theoretical version of a practical language for concurrency and is still based on an operational intuition which, however, is interpreted w.r.t. a more abstract theory of decorated traces that model how systems react to external stimuli. The meaning of a CSP term is the set of possible runs enriched with information about the interactions that could be refused at intermediate steps of each run. ACP started from a completely different viewpoint and provided a purely algebraic view of concurrent systems: processes are the solutions of systems of equations (axioms) over the signature of the considered algebra. Operational semantics and behavioral equivalences are seen as possible models over which the algebra can be defined and the axioms can be applied. The meaning of a term is given via a predefined set of equations and is the collection of terms that are provably equal to it.

At first, the different algebras have been developed independently. Slowly, however, their close relationships have been understood and appreciated, and now a general theory can be provided and the different formalisms (CCS, CSP, ACP, etc.) can be seen just as instances of the general approach. In this general approach, the main ingredients of a specific process algebra are:

1. A minimal set of carefully chosen operators capturing the relevant aspect of systems behavior and the way systems are composed in building process terms
2. A transition system associated with each term via structural *operational semantics* to describe the evolution of all processes that can be built from the operators
3. An equivalence notion that allow one to abstract from irrelevant details of systems descriptions

Verification of concurrent system within the process algebraic approach is carried out either by resorting to behavioral equivalences for proving conformance of processes to specifications or by checking that processes enjoy properties described by some temporal logic formulae [14, 28]. In the former case, two descriptions of a given system, one very detailed and close to the actual concurrent implementation, the other more abstract (describing the sequences or trees of relevant actions the system has to perform) are provided and tested for equivalence. In the latter case, concurrent systems are specified as process terms, while properties are specified

as temporal logic formulae, and model checking is used to determine whether the transition systems associated with terms enjoy the property specified by the formulae.

In the next section, many of the different operators used in process algebras will be described. By relying on the so-called structural operational semantic (SOS) approach [37], it will be shown how labeled transition systems can be built and composed by using the different operators. Afterward, many behavioral equivalences will be introduced together with a discussion on the induced identifications and distinctions. Next, the three most popular process algebras will be described; for each of them a different approach (operational, denotational, algebraic) will be used. It will, however, be argued that in all cases, the operational semantics plays a central rôle.

## Process Operators and Operational Semantics

To define a process calculus, one starts with a set of uninterpreted action names (that might represent communication channels, synchronization actions, etc.) and with a set of basic processes that together with the actions are the building blocks for forming newer processes from existing ones. The operators are used for describing sequential, nondeterministic, or parallel compositions of processes, for abstracting from internal details of process behaviors and, finally, for defining infinite behaviors starting from finite presentations. The operational semantics of the different operators is inductively specified through SOS rules: for each operator, there is a set of rules describing the behavior of a system in terms of the behaviors of its components. As a result, each process term is seen as a component that can interact with other components or with the external environment.

In the rest of this section, most of the operators that have been used in some of the best-known process algebras will be presented with the aim of showing the wealth of choices that one has when deciding how to describe a concurrent system or even when defining one's "personal" process algebra. A new calculus can, indeed, be obtained by a careful selection of the operators while taking into account their interrelationships with respect to the chosen abstract view of process and thus of the behavioral equivalence one has in mind.

A set of operators is the basis for building process terms. A labeled transition system (LTS) is associated to each term by relying on structural induction by providing specific rules in for each operator. Formally speaking, an LTS is a set of nodes (corresponding to process terms) and (for each action  $a$  in some set) a relation  $\xrightarrow{a}$  between nodes, corresponding to processes transitions. Often LTSSs have a distinguished node  $n_0$  from which computations start; when defining the semantics of a process term, the state corresponding to that term is considered as the initial state. To associate an LTS to a process term, inference systems are used, where the collection of transitions is specified by means of a set of syntax-driven inference rules.

*Inference systems:* An inference system is a set of inference rule of the form:

$$\frac{p_1, \dots, p_n}{q}$$

where  $p_1, \dots, p_n$  are the *premises* and  $q$  is the *conclusion*. Each rule is interpreted as an implication: if all premises are true, then also the conclusion is true. Sometimes, rules are decorated with predicates and/or negative premises that specify when the rule is actually applicable.

A rule with an empty set of premises is called *axiom* and written as:

$$\frac{}{q}$$

*Transition rules:* In the case of an operational semantics, the premises and the conclusions will be triples of the form  $(P, \alpha, Q)$ , often rendered as  $P \xrightarrow{\alpha} Q$ , and thus the rules for each operator  $op$  of the process algebras will be of the following form, where  $\{i_1, \dots, i_m\} \subseteq \{1, \dots, n\}$  and  $E'_i = E_i$  when  $i \notin \{i_1, \dots, i_m\}$ :

$$\frac{E_{i_1} \xrightarrow{\alpha_1} E'_{i_1} \dots E_{i_m} \xrightarrow{\alpha_m} E'_{i_m}}{op(E_1, \dots, E_n) \xrightarrow{\alpha} C[E'_1, \dots, E'_n]}$$

In the rule above, the target term  $C[ ]$  indicates the new context in which the new subterms will be operating after the reduction and  $\alpha$  represents the action performed by the composite system when some of the components perform actions  $\alpha_1, \dots, \alpha_m$ . Sometimes, these rules are enriched with side conditions that determine their applicability. By imposing syntactic constraints on the form of the allowed rules, *rule formats* are obtained

that can be used to establish results that hold for all process calculi whose transition rules respect the specific rule format.

A small number of SOS inference rules is sufficient to associate an LTS to each term of any process algebra. The set of rules is fixed once and for all. Given any process, the rules are used to derive its transitions. The transition relation of the LTS is the **least** one satisfying the inference rules. It is worth remarking that *structural induction* allows one to define the LTS of complex systems in terms of the behavior of their components.

*Basic actions:* An elementary action of a system represents the **atomic** (non-interruptible) abstract step of a computation that is performed by a system to move from one state to the next. Actions represent various activities of concurrent systems, like sending or receiving a message, updating a memory cell, and synchronizing with other processes. In process algebras two main types of atomic actions are considered, namely, *visible* or external actions and invisible or *internal* actions. In the sequel, visible actions will be denoted by Latin letters  $a, b, c, \dots$ , invisible actions will be denoted by the Greek letter  $\tau$ . Generic actions will be denoted by  $\mu$  or other, possibly indexed, Greek letters. In the following,  $A$  will be used to denote the set of visible actions while  $A_\tau$  will denote the collection of generic actions.

*Basic processes:* Process algebras generally also include a null process (variously denoted as *nil*,  $0$ , *stop*) which has no transition. It is inactive, and its sole purpose is to act as the inductive anchor on top of which more interesting processes can be generated. The semantics of this process is characterized by the fact that there is no rule to define its transition: it has no transition.

Other basic processes are also used: *stop* denotes a deadlocked state,  $\checkmark$  denotes, instead, successful termination.

$$\frac{}{\checkmark \xrightarrow{\checkmark} stop}$$

Sometimes, uninterpreted actions  $\mu$  are considered basic processes themselves:

$$\frac{}{\mu \xrightarrow{\mu} \checkmark}$$

*Sequential composition:* Operators for sequential composition are used to temporally order processes execution and interaction. There are two main operators

for this purpose. The first one is *action prefixing*,  $\mu\cdot$ , that denotes a process that executes action  $\mu$  and then behaves like the following process.

$$\overline{\mu \cdot E} \xrightarrow{\mu} E$$

The alternative form of sequential composition is obtained by explicitly requiring *process sequencing*,  $- ; -$ , that requires that the first operand process be fully executed before the second one.

$$\frac{E \xrightarrow{\mu} E'}{E; F \xrightarrow{\mu} E'; F} \quad (\mu \neq \checkmark) \quad \frac{E \xrightarrow{\checkmark} E' \quad F \xrightarrow{\mu} F'}{E; F \xrightarrow{\mu} F'}$$

*Nondeterministic composition:* The operators for nondeterministic choice are used to express alternatives among possible behaviors. This choice can be left to the environment (*external choice*) or performed by the process (*internal choice*) or can be mainly external, but leaving the possibility to the process to perform an internal move to prevent some of the choices by the environment (*mixed choice*).

The rules for mixed choice are the ones below. They offer both visible and invisible actions to the environment; however, only the former kind of actions can be actually controlled.

$$\frac{E \xrightarrow{\mu} E'}{E + F \xrightarrow{\mu} E'} \quad \frac{F \xrightarrow{\mu} F'}{E + F \xrightarrow{\mu} F'}$$

The rules for internal choice are very simple, they are just two axioms stating that a process  $E \oplus F$  can silently evolve into one of its subcomponents.

$$\overline{E \oplus F \xrightarrow{\tau} E} \quad \overline{E \oplus F \xrightarrow{\tau} F}$$

The rules for external choice are more articulate. This operator behaves exactly like the mixed choice in case one of the components executes a visible action; however, it does not discard any alternative upon execution of an invisible action.

$$\begin{array}{c} \frac{E \xrightarrow{\alpha} E'}{E \sqcap F \xrightarrow{\alpha} E'} \quad (\alpha \neq \tau) \quad \frac{F \xrightarrow{\alpha} F'}{E \sqcap F \xrightarrow{\alpha} F'} \quad (\alpha \neq \tau) \\ \hline \frac{E \xrightarrow{\tau} E'}{E \sqcap F \xrightarrow{\tau} E' \sqcap F} \quad \frac{F \xrightarrow{\tau} F'}{E \sqcap F \xrightarrow{\tau} E \sqcap F'} \end{array}$$

*Parallel composition:* Parallel composition of two processes, say  $E$  and  $F$ , is the key primitive distinguishing

process algebras from sequential models of computation. Parallel composition allows computation in  $E$  and  $F$  to proceed simultaneously and independently. But it also allows interaction, that is synchronization and flow of information between  $E$  and  $F$  on a shared channel. Channels may be synchronous or asynchronous. In the case of synchronous channels, the agent sending a message waits until another agent has received the message. Asynchronous channels do not force the sender to wait. Here, only synchronous channels will be considered.

The simplest operator for parallel composition is *interleaving*,  $- \parallel -$ , that aims at modeling the fact that two parallel processes can progress by alternating at any rate the execution of their actions.

$$\frac{E \xrightarrow{\mu} E'}{E \parallel F \xrightarrow{\mu} E' \parallel F} \quad \frac{F \xrightarrow{\mu} F'}{E \parallel F \xrightarrow{\mu} E \parallel F'}$$

Another parallel operator is *binary parallel composition*,  $- | -$ , that not only models the interleaved execution of the actions of two parallel processes, but also the possibility that the two partners synchronize whenever they are willing to perform complementary visible actions (below represented as  $a$  and  $\bar{a}$ ). In this case, the visible outcome is a  $\tau$ -action that cannot be seen by other processes that are acting in parallel with the two communication partners. This is the parallel composition used in CCS.

$$\begin{array}{c} \frac{E \xrightarrow{\mu} E'}{E | F \xrightarrow{\mu} E' | F} \quad \frac{F \xrightarrow{\mu} F'}{E | F \xrightarrow{\mu} E | F'} \\ \hline \frac{E \xrightarrow{\alpha} E' \quad F \xrightarrow{\bar{\alpha}} F'}{E | F \xrightarrow{\tau} E' | F'} \quad (\alpha \neq \tau) \end{array}$$

Instead of binary synchronization, some process algebras, like CSP, make use of operators that permit *multiparty synchronization*,  $- [[L]] -$ . Some actions, those in  $L$ , are deemed to be synchronization actions and can be performed by a process only if all its parallel components can execute those actions at the same time.

$$\begin{array}{c} \frac{E \xrightarrow{\mu} E'}{E [[L]] F \xrightarrow{\mu} E' [[L]] F} \quad (\mu \notin L) \\ \hline \frac{F \xrightarrow{\mu} F'}{E [[L]] F \xrightarrow{\mu} E [[L]] F'} \quad (\mu \notin L) \end{array}$$

$$\frac{E \xrightarrow{a} E' \quad F \xrightarrow{a} F'}{E \parallel [L] \parallel F \xrightarrow{a} E' \parallel [L] \parallel F'} \quad (a \in L)$$

It is worth noting that the result of a synchronization, in this case, yields a visible action, and that by setting the synchronization alphabet to  $\emptyset$  the multiparty synchronization operator  $| \emptyset |$  can be used to obtain pure interleaving,  $\parallel\parallel$ .

A more general composition is the *merge* operator,  $- \parallel -$  that is used in ACP. It permits executing two process terms in parallel (thus freely interleaving their actions), but also allows for communication between its process arguments according to a *communication function*  $\gamma : A \times A \rightarrow A$ , that, for each pair of atomic actions  $a$  and  $b$ , produces the outcome of their communication  $\gamma(a, b)$ , a partial function that states which actions can be synchronized and the outcome of such a synchronization.

$$\frac{\begin{array}{c} E \xrightarrow{\mu} E' \\ E \parallel F \xrightarrow{\mu} F' \parallel F \end{array}}{E \parallel F \xrightarrow{\mu} E' \parallel F'} \quad \frac{\begin{array}{c} F \xrightarrow{\mu} F' \\ E \parallel F \xrightarrow{\mu} E \parallel F' \end{array}}{E \parallel F \xrightarrow{\mu} E \parallel F'}$$

$$\frac{\begin{array}{c} E \xrightarrow{a} E' \quad F \xrightarrow{b} F' \\ E \parallel F \xrightarrow{\gamma(a,b)} E' \parallel F' \end{array}}{E \parallel F \xrightarrow{\gamma(a,b)} E' \parallel F'}$$

ACP has also another operator called *left merge*,  $- \parallel|_c -$ , that is similar to  $\parallel$  but requires that the first process to perform an (independent) action be the left operand.

$$\frac{E \xrightarrow{\mu} E'}{E \parallel|_c F \xrightarrow{\mu} E' \parallel F}$$

The ACP *communication merge*,  $- |_c -$ , requires instead that the first action be a synchronization action.

$$\frac{\begin{array}{c} E \xrightarrow{a} E' \quad F \xrightarrow{b} F' \\ E |_c F \xrightarrow{\gamma(a,b)} E' \parallel F' \end{array}}{E |_c F \xrightarrow{\gamma(a,b)} E' \parallel F'}$$

*Disruption:* An operator that is between parallel and nondeterministic composition is the so-called *disabling* operator,  $-[> -]$ , that permits interrupting the evolution of a process. Intuitively,  $E [> F$  behaves like  $E$ , but can be interrupted at any time by  $F$ , once  $E$  terminates  $F$  is discarded.

$$\frac{\begin{array}{c} E \xrightarrow{\mu} E' \\ E [> F \xrightarrow{\mu} E' [> F \end{array}}{E [> F \xrightarrow{\mu} E' [> F} \quad (\mu \neq \checkmark)$$

$$\frac{E \xrightarrow{\checkmark} E'}{E [> F \xrightarrow{\tau} E'}$$

$$\frac{F \xrightarrow{\mu} F'}{E [> F \xrightarrow{\mu} F']}$$

*Value passing:* The above parallel combinators can be generalized to model not only synchronization, but also exchange of values. As an example, below, the generalization of binary communication is presented.

There are complementary rules for sending and receiving values. The first axiom models a process willing to input a value and to base its future evolutions on it. The second axiom models a process that evaluates an expression (via the valuation function  $val(e)$ ) and outputs the result.

$$\frac{}{(v \text{ is a value})}{\overline{a(x).E \xrightarrow{a(v)} E\{v/x\}}} \quad \frac{}{\overline{a.e.E \xrightarrow{\bar{a} val(e)} E}}$$

The next rule, instead, models synchronization between processes. If two processes, one willing to output and the other willing to input, are running in parallel, a synchronization can take place and the perceived action will just be a  $\tau$ -action.

$$\frac{\begin{array}{c} E \xrightarrow{\bar{a} v} E' \quad F \xrightarrow{a(v)} F' \\ E | F \xrightarrow{\tau} E' | F' \end{array}}{E | F \xrightarrow{\tau} E' | F'} \quad \frac{\begin{array}{c} E \xrightarrow{a(v)} E' \quad F \xrightarrow{\bar{a} v} F' \\ E | F \xrightarrow{\tau} E' | F' \end{array}}{E | F \xrightarrow{\tau} E' | F'}$$

In case the exchanged values are channels, this approach can be used to provide also models for mobile systems.

*Abstraction:* Processes do not limit the number of connections that can be made at a given interaction point. But interaction points allow interference. For the synthesis of compact, minimal, and compositional systems, the ability to restrict interference is crucial.

The *hiding* operator,  $-/L$ , hides (i.e., transforms into  $\tau$ -actions) all actions in  $L$  to forbid synchronization on them. However, it allows the system to perform the transitions labeled by hidden actions.

$$\frac{\begin{array}{c} E \xrightarrow{\mu} E' \\ E/L \xrightarrow{\mu} E'/L \end{array}}{E \xrightarrow{\mu} E'/L} \quad (\mu \notin L) \quad \frac{E \xrightarrow{\mu} E'}{E/L \xrightarrow{\tau} E'/L} \quad (\mu \in L)$$

The *restriction* operator,  $- \backslash L$  is a unary operator that restricts the set of visible actions a process can perform. Thus, process  $E \backslash L$  can perform only actions not in  $L$ . Obviously, invisible actions cannot be restricted.

$$\frac{E \xrightarrow{\mu} E'}{E \backslash L \xrightarrow{\mu} E' \backslash L} \quad (\mu, \bar{\mu} \notin L)$$

The operator  $[f]$ , where  $f$  is a *relabelling* function from  $A$  to  $A$ , can be used to rename some of the actions

a process can perform to make it compatible with new environments.

$$\frac{E \xrightarrow{a} E'}{E[f] \xrightarrow{f(a)} E'[f]}$$

*Modeling infinite behaviors:* The operations presented so far describe only finite interaction and are consequently insufficient for providing full computational power, in the sense of being able to model all computable functions. In order to reach full power, one certainly needs operators for modeling non-terminating behavior. Many operators have been introduced that allow finite descriptions of infinite behavior. However, it is important to remark that most of them do not fit the formats used so far and cannot be defined by structural induction.

One of the most used is the construct  $\text{rec } x. \text{ -}$ , well-known from the sequential world. If  $E$  is a process that contains the variable  $x$ , then  $\text{rec } x. E$  represents the process that behaves like  $E$  once all occurrences of  $x$  in  $E$  are replaced by  $\text{rec } x. E$ . In the rule below, that models the operational behavior of a recursively defined process, the term  $E[f/x]$  denotes exactly the above mentioned substitutions.

$$\frac{E[\text{rec } x. E/x] \xrightarrow{\mu} E'}{\text{rec } x. E \xrightarrow{\mu} E'}$$

The notation  $\text{rec } x. E$  for recursion sometimes makes the process expressions more difficult to parse and less pleasant to read. A suitable alternative is to allow for the (recursive) definition of some fixed set of constants, that can then be used as some sort of procedure calls inside processes. Assuming the existence of an environment (a set of process definitions)

$$\Gamma = \{X_1 \triangleq E_1, X_2 \triangleq E_2, \dots, X_n \triangleq E_n\}$$

the operational semantics rule for a process variable becomes:

$$\frac{X \triangleq E \in \Gamma \quad E \xrightarrow{\mu} E'}{X \xrightarrow{\mu} E'}$$

Another operator used to describe infinite behaviors is the so-called *bang* operator,  $! -$ , or *replication*. Intuitively,  $!E$  represents an unlimited number of instances of  $E$  running in parallel. Thus, its semantics is rendered by the following inference rule:

$$\frac{E \parallel E \xrightarrow{\mu} E'}{!E \xrightarrow{\mu} E'}$$

## Three Process Algebras: CCS, CSP and ACP

A process algebra consists of a set of terms, an operational semantics associating LTS to terms, and an equivalence relation equating terms exhibiting “similar” behavior. The operators for most process algebras are those described above. The equivalences can be traces, testing, bisimulation equivalences, or variants thereof, possibly ignoring invisible actions.

Below, three of the most popular process algebras are presented. First the syntax, i.e., the selected operators, will be introduced, then their semantics will be provided by following the three different approaches outlined before: operational (for CCS), denotational (for CSP), and algebraic (for ACP). For CSP and ACP, the relationships between the proposed semantics and the operational one, to be used as a yardstick, will be mentioned. To denote the LTS associated to a generic CSP or ACP process  $p$  via the operational semantics, the notation  $\text{LTS}(p)$  will be used.

Reference will be made to specific behavioral equivalences over LTSs that consider as equivalent those systems that rely on different standing about which states of an LTS have to be considered equivalent. Three main criteria have been used to decide when two systems can be considered equivalent:

1. The two systems perform the same sequences of actions.
2. The two systems perform the same sequences of actions and after each sequence are ready to accept the same sets of actions.
3. The two systems perform the same sequences of actions and after each sequence exhibit, recursively, the same behavior.

These three different criteria lead to three groups of equivalences that are known as *traces* equivalences, *decorated-traces* equivalences (testing and failure equivalence), and *bisimulation-based* equivalences (strong bisimulation, weak bisimulation, branching bisimulation).

### CCS: Calculus of Communicating Systems

The Calculus of Communicating Systems (CCS) is a process algebra introduced by Robin Milner around 1980. Its actions model indivisible communications

between exactly two participants, and the set of operators includes primitives for describing parallel composition, choice between actions, and scope restriction. The basic semantics is operational and permits associating an LTS to each CCS term.

The set  $A$  of basic actions used in CCS consists of a set  $\Lambda$ , of labels and of a set  $\bar{\Lambda}$  of complementary labels.  $A_\tau$  denotes  $A \cup \{\tau\}$ . The syntax of CCS, used to generate all terms of the algebra, is the following:

$P ::= nil \mid x \mid \mu.P \mid P \setminus L \mid P[f] \mid P_1 + P_2 \mid P_1 | P_2 \mid rec\ x.\ P$   
where  $\mu \in A_\tau$ ;  $L \subseteq \Lambda$ ;  $f : A_\tau \rightarrow A_\tau$ ;  $f(\bar{\alpha}) = \overline{f(\alpha)}$  and  $f(\tau) = \tau$ . The above operators are taken from those presented in Section 2:

- The atomic process (*nil*)
- Action prefixing ( $\mu.P$ )
- Mixed choice (+)
- Binary parallel composition ( $|$ )
- Restriction ( $P \setminus L$ )
- Relabeling ( $P[f]$ )
- Recursive definitions ( $rec\ x.\ E$ )

The operational semantics of the above operators is exactly the same as the one of those operators with the same name described before, and it is thus not repeated here. CCS has been studied with bisimulation and testing semantics that are used to abstract from unnecessary details of the LTS associated to a term. Also denotational and axiomatic semantics for the calculus have been extensively studied. A denotational semantics in terms of so-called *acceptance trees* has been proved to be in full agreement with the operational semantics abstracted according to testing equivalences. Different algebraic semantics have been provided that are based on sound and complete axiomatizations of bisimilarity, testing equivalence, weak bisimilarity, and branching bisimilarity.

## CSP: A Theory of Communicating Sequential Processes

The first denotational semantics proposed for CSP associates to each term just the set of the sequences of actions the term could induce. However, while suitable to model the sequences of interactions a process could have with its environment, this semantics is unable to model situations that could lead to deadlock. A new approach, basically denotational but with a strong operational intuition, was proposed next. In this approach,

the semantics is given by associating a so-called refusal set to each process. A refusal set is a set of failure pairs  $\langle s, F \rangle$ , where  $s$  is a finite sequence of visible actions in which the process might have been engaged, and  $F$  is a set of action the process is able to reject on the next step. The semantics of the various operators is defined by describing the transformation they induce on the domain of refusal sets.

The meaning of processes is then obtained by postulating that two processes are equivalent if and only if they cannot be distinguished when their behaviors are observed and their reactions to a finite number of alternative possible synchronization is considered. Indeed, the association of processes to refusal sets is not one-to-one; the same refusal set can be associated to more than one process. A congruence is then obtained that equates processes with the same denotation.

The set of actions is a finite set of labels, denoted by  $\Lambda \cup \{\tau\}$ . There is no notion of complementary action. The syntax of CSP is reported below, and for the sake of simplicity, only finite terms (no recursion) are considered:

$E ::= STOP \mid skip \mid a \rightarrow E \mid E_1 \sqcap E_2 \mid E_1 \sqcap E_2 \mid [E_1][L] \mid E_2 \mid E/a$

- Two basic processes: successful termination (*skip*), null process (*STOP*)
- Action prefixing here denoted by  $a \rightarrow E$
- Internal choice ( $\oplus$ ) here denoted by  $\sqcap$  and external choice ( $\sqcap$ )
- Parallel composition with synchronization on a fixed alphabet ( $[E][L]$ ,  $L \subseteq \Lambda$ )
- Hiding ( $/a$ , an instance of the more general operator  $/L$  with  $L \subseteq \Lambda$ )

Parallel combinators representing pure interleaving and parallelism with synchronization on the full alphabet can be obtained by setting the synchronization alphabet to  $\emptyset$  or to  $\Lambda$ , respectively.

The denotational semantics of CSP compositionally associates a set of failure pairs to each CSP term generated by the above syntax. A function  $\mathcal{F}[\cdot]$  maps each CSP process (say  $P$ ) to set of pairs  $(s, F)$ , where  $s$  is one of the sequences of actions  $P$  may perform, and  $F$  represents the set of actions that  $P$  can refuse after performing  $s$ . As anticipated, there is a strong correspondence between the denotational semantics of CSP and the operational semantics that one could define by

relying on the one presented in the previous section for the specific operators.

- $\mathcal{F}[[P]] = \mathcal{F}[[Q]]$  if and only if  $\text{LTS}(P) \simeq_{test} \text{LTS}(Q)$ .

## ACP: An Algebra of Communicating Processes

The methodological concern of ACP was to present “first a system of axioms for communicating processes ... and next study its models” ([6], p. 112). The equations are just a means to realize the real desideratum of abstract algebra, which is to abstract from the nature of the objects under consideration. In the same way as the mathematical theory of rings is about arithmetic without relying on a mathematical definition of number, ACP deals with process theory without relying on a mathematical definition of process.

In ACP, a process algebra is any mathematical structure, consisting of a set of objects and a set of operators, like, e.g., sequential, nondeterministic, or parallel composition, that enjoys a specific set of properties as specified by given axioms.

The set of actions  $\Lambda_\tau$  consists of a finite set of labels  $\Lambda \cup \{\tau\}$ . There is no notion of complementary action. The syntax of ACP is reported below, and for the sake of simplicity, only finite terms (no recursion) are considered:

$$P ::= \sqrt{| \delta | a | P_1 + P_2 | P_1 \cdot P_2 | P_1 \| P_2 | P_1 \| P_2 | P_1 |_c P_2 | \partial_H(p)}$$

- Three basic processes: successful termination ( $\checkmark$ ), null process, here denoted by  $\delta$ , and atomic action ( $a$ )
- Mixed choice
- Sequential composition ( $;$ ), here denoted by  $\cdot$
- Hiding ( $\backslash H$  with  $H \subseteq \Lambda$ ), here denoted by  $\partial_H(-)$
- Three parallel combinators: merge ( $\parallel$ ), left merge ( $\|$ ) and communication merge ( $|_c$ )

The system of axioms of ACP is presented as a set of formal equations, and some of the operators, e.g., left merge ( $\|$ ) have been introduced exactly for providing finite equational presentations. Below, the axioms relative to the terms generated by the above syntax are presented. Within the axioms,  $x$  and  $y$  denote generic ACP processes.

- |                                                  |                                                |
|--------------------------------------------------|------------------------------------------------|
| (A1) $x + y = y + x$                             | (A2) $(x + y) + z = x + (y + z)$               |
| (A3) $x + x = x$                                 | (A4) $(x + y) \cdot z = x \cdot z + y \cdot z$ |
| (A5) $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ | (A6) $x + \delta = x$                          |
| (A7) $\delta \cdot x = \delta$                   |                                                |

The set of axioms considered above induces an *equality relation*, denoted by  $=$ . A *model* for an axiomatization is a pair  $(\mathcal{M}, \phi)$ , where  $\mathcal{M}$  is a set and  $\phi$  is a function (the unique isomorphism) that associates elements of  $\mathcal{M}$  to ACP terms. This leads to the following definitions:

1. A set of equations is *sound* for  $(\mathcal{M}, \phi)$  if  $s = t$  implies  $\phi(s) = \phi(t)$ .
2. A set of equations is *complete* for  $(\mathcal{M}, \phi)$  if  $\phi(s) = \phi(t)$  implies  $s = t$ .

Any model of the axioms seen above is an ACP process algebra. The simplest model for ACP has as elements the equivalence classes induced by  $=$ , i.e., all ACP terms obtained starting from atomic actions, sequentialization and nondeterministic composition and mapping each term  $t$  to its equivalence class  $[[t]]$  as determined by  $=$ . This model is correct and complete and is known as the *initial model* for the axiomatization.

Different, more complex, models can be obtained by first using the SOS rules to give the operational semantics of the operators, building an LTS in correspondence of each ACP term and then using bisimulation to identify some of them. This construction leads to establishing a strong correspondence between the axiomatic and the operational semantics of ACP. Indeed, if we consider the language with the null process, sequential composition and mixed choice we have:

- Equality  $=$  as induced by (A1)-(A7) is *sound* relative to bisimilarity  $\sim$ , i.e., if  $p = q$  then  $\text{LTS}(p) \sim \text{LTS}(q)$ ;
- Equality  $=$  as induced by (A1)-(A7) is *complete* relative to bisimilarity  $\sim$ , i.e., if  $\text{LTS}(p) \sim \text{LTS}(q)$  then  $p = q$ .

Similar results can be obtained when new axioms are added and weak bisimilarity or branching bisimilarity are used to factorize the LTSs.

## Future Directions

The theory of process algebra is by now well developed. The reader is referred to [7] to learn about its developments since its inception in the late 1970s to the early 2000. Currently, in parallel with the exploitation of the developed theories in classic areas such as protocol verification and in new ones such as biological systems, there is much work going on concerning:

- Extensions to model mobile, network aware systems
- Theories for assessing quantitative properties
- Techniques for controlling state explosion

In parallel with this, much attention is dedicated to the development of software tools to support specification and verification of very large systems and to the development of techniques that permit controlling the state explosion phenomenon that arise as soon as one considers the possible configurations resulting from the interleaved execution of (even a small number of) processes.

*Mobility and network awareness:* Much of the ongoing work is relative to the definition of theories and formalisms to naturally deal with richer classes of systems, like, e.g., mobile systems and network aware applications. The  $\pi$ -calculus [33] the successor of CCS, developed by Milner and coworkers with the aim of describing concurrent systems whose configuration may change during the computation has attracted much attention. It has laid the basis for research on process networks whose processes are mobile and the configuration of communication links is dynamic. It has also lead to the development of other calculi to support network aware programming: Ambient [13], Distributed  $\pi$  [25], Join [19], Spi [1], Klaim [9], etc. There is still no unifying theory and the name process calculi is preferred to process algebras because the algebraic theories are not yet well assessed. Richer theories than LTS (Bi-graph [34], Tiles [20], etc.) have been developed and are still under development to deal with the new dimensions considered with the new formalisms.

*Quantitative extensions:* Formalisms are being enriched to consider not only qualitative properties, like correctness, liveness, or safety, but also properties related to performance and quality of service. There has been much research to extend process algebra to deal with a quantitative notion of time and probabilities and

integrated theories have been considered. Actions are enriched with information about their duration, and formalisms extended in this way are used to compare systems relatively to their speed. For a comprehensive description of this approach, the reader is referred to [4]. Extensions have been considered also to deal with systems that in their behavior depend on continuously changing variables other than time (*hybrid systems*). In this case, systems descriptions involve differential algebraic equations, and connections with dynamic control theory are very important. Finally, again with the aim of capturing quantitative properties of systems and of combining functional verification with performance analysis, there have been extensions to enrich actions with rates representing the frequency of specific events and the new theories are being (successfully) used to reason about system performance and system quality.

*Tools:* To deal with non-toy examples and apply the theory of process algebras to the specification and verification of real systems, tool support is essential. In the development of tools, LTSs play a central role. Process algebra terms are used to obtain LTSs by exploiting operational semantics and these structures are then minimized, tested for equivalence, model checked against formulae of temporal logics, etc. One of the most known tools for process algebras is CADP (Construction and Analysis of Distributed Processes) [21]: together with minimizers and equivalence and model checkers, it offers many others functionalities ranging from step-by-step simulation to massively parallel model checking. CADP has been employed in an impressive number of industrial projects. CWB (Concurrency Workbench) [35] and CWB-NC (Concurrency Workbench New Century) [15] are other tools that are centered on CCS, bisimulation equivalence, and model checking. FDR (Failures/Divergence Refinement) [40] is a commercial tool for CSP that has played a major role in driving the evolution of CSP from a blackboard notation to a concrete language. It allows the checking of a wide range of correctness conditions, including deadlock and livelock freedom as well as general safety and liveness properties. TAPAs (Tool for the Analysis of Process Algebras) [12] is a recently developed software to support teaching of the theory of process algebras; it maintains a consistent double representation as term and as graph of each system.

Moreover, it offers tools for the verification of many behavioral equivalences, possibly with counterexamples, minimization, step-by-step execution, and model checking. TwoTowers is instead a versatile tool for the functional verification, security analysis, and performance evaluation of computer, communication, and software systems modeled with the stochastic process algebra EMPA [4].  $\mu$ CRL [23] is a toolset that offers an appropriate treatment of data and relies also on theorem proving. Moreover, it make use of interesting techniques for visualizing large LTSSs.

## Relationships to Other Models of Concurrency

In a private communication, in 2009, Robin Milner, one of the founding fathers of process algebras, wrote:

- ▶ The concept of process has become increasingly important in computer science in the last three decades and more. Yet we still don't agree on what a process is. We probably agree that it should be an equivalence class of interactive agents, perhaps concurrent, perhaps non-deterministic.

This quote summarizes the debate on possible models of concurrency that has taken place during the last thirty years and has been centered on three main issues:

- Interleaving vs true concurrency
- Linear-time vs branching-time
- Synchrony vs asynchrony

### Interleaving vs True Concurrency

The starting point of the theory of process algebras has been automata theory and regular expressions, and the work on the algebraic theory of regular expressions as terms representing finite state automata [16] has significantly influenced its developments. Given this starting point, the underlying models of all process algebras represent possible concurrent executions of different programs in terms of the nondeterministic interleaving of their sequential behaviors. The fact that a system is composed by independently computing agents is ignored and behaviors are modeled in terms of purely sequential patterns of actions. It has been demonstrated that many interesting and important properties of distributed systems may be expressed

and proved by relying on interleaving models. However, there are situations in which it is important to keep the information that a system is composed of the independently computing components. This possibility is offered by the so-called non-interleaving or true-concurrency models, with Petri nets [39] as the prime example. These models describe not only temporal ordering of actions, but also their causal dependences. Non-interleaving semantics of process algebras have also been provided, see, e.g., [36].

### Linear-time vs Branching-time

Another issue, again ignored in the initial formalization of regular expressions, is how the concept of nondeterminism in computations is captured. Two possible views regarding the nature of nondeterministic choice induce two types of models giving rise to the linear-time and branching-time dichotomy. A linear-time model expresses the full nondeterministic behavior of a system in terms of the set of possible runs; time is treated as if each moment there is a unique possible future. Major examples of structures used to model sets of runs are Hoare traces (captured also by traces equivalence) for interleaving models [26], and Mazurkiewicz traces [30] and Pratt's pomsets [38] for non-interleaving models. The branching-time model is the main one considered in process algebras and considers the set of runs structured as a computation tree. Each moment in time may split into various possible futures, and semantic models are computation trees. For non-interleaving models, event structures [44] are one of the best-known models taking into account both nondeterminism and true concurrency.

### Synchrony vs Asynchrony

There are two basic approaches to describing interaction between a sender and a receiver of a message (signal), namely, synchronous and asynchronous interaction. In the former case, before proceeding, the sender has to make sure that a receiver is ready. In the latter case, the sender leaves track of its action but proceeds without any further waiting. The receiver has to wait in both cases. Process algebras are mainly synchronous, but asynchronous variants have been recently proposed and are receiving increasing attention. However, many other successful asynchronous models have been developed. Among these, it is important to mention Esterel

[8], a full-fledged programming language that allows the simple expression of parallelism and preemption and is very well suited for control-dominated model designs; Actors [3], a formalism that does not necessarily records messages in buffers and puts no requirement on the ordering of message delivery; Linda [22], a model of coordination and communication among several parallel processes operating upon objects stored in and retrieved from shared, virtual, associative memory; and, to conclude, Klam [17], a distributed variant of Linda with a strong process algebraic flavor.

## Related Entries

- ▶ [Actors](#)
- ▶ [Behavioral Equivalences](#)
- ▶ [Bisimulation](#)
- ▶ [CSP \(Communicating Sequential Processes\)](#)
- ▶ [Pi-Calculus](#)

## Bibliographic Notes and Further Reading

A number of books describing the different process algebras can be consulted to obtain deeper knowledge of the topics sketched here. Unfortunately most of them are concentrating only on one of the formalisms rather than on illustrating the unifying theories.

**CCS:** The seminal book on CCS is [31], in which sets of operators equipped with an operational semantics and the notion of observational equivalence have been presented for the first time. The, by now, classical text book on CCS and bisimulation is [32]. A very nice, more recent, book on CCS and the associated Hennessy Milner Modal Logic is [29]; it also presents timed variants of process algebras and introduces models and tools for verifying properties also of this new class of systems.

**CSP:** The seminal book on CSP is [27], where all the basic theory of failure sets is presented together with many operators for processes composition and basic examples. In [41], the theory introduced in [27] is developed in full detail, and a discussion on the different possibilities to deal with anomalous infinite behaviors is considered together with a number of well-thought examples. Moreover the relationships between operational and denotational semantics are fully investigated.

Another excellent text book on CSP is [43] that also considers timed extensions of the calculus.

**ACP:** The first published book on ACP is [5], where the foundations of algebraic theories are presented and the correspondence between families of axioms, and strong and branching bisimulation are thoroughly studied. This is a book intended mainly for researchers and advanced students, a gentle introduction to ACP can be found in [18].

**Other approaches:** Apart from these books, dealing with the three process algebras presented in these notes, it is also worth mentioning a few more books. LOTOS, a process algebra that was developed and standardized within ISO for specifying and verifying communication protocols, is the central calculus of a recently published book [10] that discusses also the possibility of using different equivalences and finer semantics for the calculus. A very simple and elegant introduction to algebraic, denotational, and operational semantics of processes, which studies in detail the impact of the testing approach on a calculus obtained from a careful selection of operators from CCS and CSP, can be found in [24]. The text [42] is *the* book on the  $\pi$ -calculus. For studying this calculus, the reader is, however, encouraged to consider first reading [33].

## Bibliography

1. Abadi M, Gordon AD (1999) A calculus for cryptographic protocols: the spi calculus. *Inform Comput* 148(1):1–70
2. Aceto L, Gordon AD (eds) (2005) Proceedings of the workshop “Essays on algebraic process calculi” (APC 25), Bertinoro, Italy. Electronic notes in theoretical computer science vol 162. Elsevier, Amsterdam
3. Agha G (1986) Actors: a model of concurrent computing in distributed systems. MIT Press, Cambridge
4. Aldini A, Bernardo M, Corradini F (2010) A process algebraic approach to software architecture design. Springer, New York
5. Baeten JCM, Weijland WP (1990) Process algebra. Cambridge University Press, Cambridge
6. Bergstra JA, Klop JW (1984) Process algebra for synchronous communication. *Inform Control* 60(1–3):109–137
7. Bergstra JA, Ponse A, Smolka SA (eds) (2001) Handbook of process algebra. Elsevier, Amsterdam
8. Berry G, Gonthier G (1992) The esterel synchronous programming language: design, semantics, implementation. *Sci Comput Program* 19(2):87–152

9. Bettini L, Bono V, Nicola R, Ferrari G, Gorla D, Loreti M, Moggi E, Pugliese R, Tuosto E, Venneri B (2003) The klabel project: theory and practice. In: Global computing: programming environments, languages, security and analysis of systems, Lecture notes in computer science, vol 2874. Springer-Verlag, Heidelberg, pp 88–150
10. Bowman H, Gomez R (2006) Concurrency theory: calculi and automata for modelling untimed and timed concurrent systems. Springer, London
11. Brookes SD, Hoare CAR, Roscoe AW (1984) A theory of communicating sequential processes. *J ACM* 31(3):560–599
12. Calzolai F, De Nicola R, Loreti M, Tiezzi F (2008) Tapas: a tool for the analysis of process algebras. In: Transactions on Petri nets and other models of concurrency, vol 1, pp 54–70
13. Cardelli L, Gordon AD (2000) Mobile ambients. *Theor Comput Sci* 240(1):177–213
14. Clarke EM, Emerson EA (1982) Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Proceedings of logic of programs, Lecture notes in computer science, vol 131. Springer-Verlag, Heidelberg, pp 52–71
15. Cleaveland R, Sims S (1996) The ncsu concurrency workbench. In: CAV, Lecture notes in computer science, vol 1102. Springer-Verlag, Heidelberg, pp 394–397
16. Conway JH (1971) Regular algebra and finite machines. Chapman and Hall, London
17. De Nicola R, Ferrari GL, Pugliese R (1998) Klabel: a kernel language for agents interaction and mobility. *IEEE Trans Software Eng* 24(5):315–330
18. Fokkink W (2000) Introduction to process algebra. Springer-Verlag, Heidelberg
19. Fournet C, Gonthier G (2000) The join calculus: a language for distributed mobile programming. In: Barthe G, Dybjer P, Pinto L, and Saraiva J (eds) APPSEM, Lecture notes in computer science, vol 2395, Springer, Heidelberg, pp 268–332
20. Gadducci F, Montanari U (2000) The tile model. In: Plotkin G, Stirling C, Tofte M (eds) Proof, language and interaction: essays in honour of Robin Milner. MIT Press, Cambridge, pp 133–166
21. Garavel H, Lang F, Mateescu R (2002) An overview of CADP 2001. In: European Association for Software Science and Technology (EASST), vol 4. Newsletter, pp 13–24
22. Gelernter D, Carriero N (1992) Coordination languages and their significance. *Commun ACM* 35(2):96–107
23. Groote JF, Mathijssen AHJ, Reniers MA, Usenko YS, van Weerenburg MJ (2009) Analysis of distributed systems with mcrl2. In: Alexander M, Gardner W (eds) Process algebra for parallel and distributed processing. Chapman Hall, Boca Raton, FL, pp 99–128
24. Hennessy M (1988) Algebraic theory of processes. The MIT Press, Cambridge
25. Hennessy M (2007) A distributed pi-calculus. Cambridge University Press, Cambridge
26. Hoare CAR (1981) A calculus of total correctness for communicating processes. *Sci Comput Program* 1(1–2):49–72
27. Hoare CAR (1985) Communicating sequential processes. Prentice-Hall, Upper Saddle River
28. Kozen D (1983) Results on the propositional -calculus. *Theor Comput Sci* 27:333–354
29. Larsen KG, Aceto L, Ingolsdottir A, Srba J (2007) Reactive systems: modelling, specification and verification. Cambridge University Press, Cambridge
30. Mazurkiewicz A (1995) Introduction to trace theory. In: Rozenberg G, Diekert V (ed) The book of traces. World Scientific, Singapore, pp 3–67
31. Milner R (1980) A calculus of communicating systems. Lecture notes in computer science, vol 92. Springer-Verlag, Heidelberg
32. Milner R (1989) Communication and concurrency. Prentice-Hall, Upper Saddle River
33. Milner R (1999) Communicating and mobile systems: the pi-calculus. Cambridge University Press, Cambridge
34. Milner R (2009) The space and motion of communicating agents. Cambridge University Press, Cambridge
35. Moller F, Stevens P (1999) Edinburgh Concurrency Workbench user manual. Available from <http://homepages.inf.ed.ac.uk/perdita/cwb/>. Accessed January 2011
36. Olderog ER (1991) Nets, terms and formulas. Cambridge University Press, Cambridge
37. Plotkin GD (2004) A structural approach to operational semantics. *J Log Algebr Program* 60–61:17–139
38. Pratt V (1986) Modeling concurrency with partial orders. *Int J Parallel Process* 1:3371
39. Reisig W (1985) Petri nets: an introduction. Monographs in theoretical computer science. An EATCS Series, vol 4. Springer-Verlag, Berlin
40. Roscoe AW (1994) Model-checking csp. In: A classical mind: essays in honour of C.A.R. Hoare. Prentice-Hall, Upper Saddle River
41. Roscoe AW (1998) The theory and practice of concurrency. Prentice-Hall, Upper Saddle River
42. Sangiorgi D, Walker D (2001) The  $\pi$ -Calculus: a theory of mobile processes. Cambridge University Press, Cambridge
43. Schneider SA (1999) Concurrent and real time systems: the CSP approach. John Wiley, Chichester
44. Winskel G (1989) An introduction to event structures. In: de Bakker JW, de Roever WP, Rozenberg G (eds) Linear time, branching time and partial order in logics and models for concurrency – Rex workshop, Lecture notes in computer science, vol 354. Springer, Heidelberg, pp 364–397

## Process Calculi

► [Process Algebras](#)

## Process Description Languages

► [Process Algebras](#)

## Process Synchronization

- ▶ [Path Expressions](#)
- ▶ [Synchronization](#)

## Processes, Tasks, and Threads

Processes, Tasks, and Threads are programs or parts of programs under execution. A program does not define a process, task, or thread since the same code may underlay different processes, tasks, or threads. At any given time, the state of a process, task, or thread includes the location of the instruction being executed and the value stored in all memory locations accessible to the program. An important characteristic of processes, tasks, and threads is that they must execute their instructions at a speed greater than zero when they are not explicitly blocked by synchronization operations.

Although the notion of process, tasks, and threads differs across systems, typically a process may contain multiple threads, and threads share memory while processes communicate via messages.

## Processor Allocation

- ▶ [Job Scheduling](#)

## Processor Arrays

- ▶ [Systolic Arrays](#)

## Processors-in-Memory

Processors-in-memory (PIM) are devices that tightly integrate, for example, on a single chip, both processing logic and memory with the objective of reducing memory latency and increasing memory bandwidth at a low cost in terms of power, complexity, and space.

## Bibliography

1. Patterson D, Anderson T, Cardwell N, Fromm R, Keeton K, Kozyrakis C, Thomas R, Yellick K (1997) A case for intelligent RAM. *IEEE Micro* 17(2):34–44
2. Kogge PM, Brockman JB, Sterling T, Gao G Processing in memory: chips to petaflops. In: Workshop on mixing logic and DRAM: chips that compute and remember at ISCA'97 1997

## Profiling

- ▶ [Intel® Thread Profiler](#)
- ▶ [OpenMP Profiling with ompP](#)
- ▶ [Performance Analysis Tools](#)
- ▶ [Scalasca](#)
- ▶ [TAU](#)

## Profiling with OmpP, OpenMP

- ▶ [OpenMP Profiling with OmpP](#)

## Program Graphs

- ▶ [Data Flow Graphs](#)

## Programmable Interconnect Computer

- ▶ [Blue CHiP](#)

## Programming Languages

- ▶ [Array Languages](#)
- ▶ [C\\*](#)
- ▶ [Chapel \(Cray Inc. HPCS Language\)](#)
- ▶ [Cilk](#)
- ▶ [CoArray Fortran](#)
- ▶ [Concurrent ML](#)

- Connection Machine Fortran
- Connection Machine Lisp
- Deterministic Parallel Java
- Fortran 90 and Its Successors
- Fortress (Sun HPCS Language)
- Glasgow Parallel Haskell (GpH)
- HPF (High Performance Fortran)
- Linda
- \*Lisp
- Multilisp
- NESL
- OpenMP
- Functional Languages
- Logic Languages
- PGAS (Partitioned Global Address Space) Languages
- Sisal
- Stream Programming Languages
- Titanium
- UPC
- ZPL

## Programming Models

- Actors
- BSP (Bulk Synchronous Parallelism)
- Concurrent Collections Programming Model
- CSP (Communicating Sequential Processes)
- SPMD Computational Model

## Prolog

- Logic Languages

## Prolog Machines

Prolog machines [1, 2] include instructions and devices for the efficient implementation of Prolog programs. During the years of Japan's fifth generation project, several data flow machines were designed to implement KL1, a parallel variant of Prolog.

## Related Entries

- Data Flow Computer Architecture
- Logic Languages

## Bibliography

1. Warren DH (1982) A view of the fifth generation and its impact. *AI Mag* 3(4):34–39
2. Holmer BK, Sano B, Carlton M, Van Roy P, Despain AM (1994) Design and analysis of hardware for high performance prolog. *J Logic Program* 19(20):1–679

## Promises

- Futures

## Protein Docking

ROGER S. ARMEN<sup>1</sup>, ERIC R. MAY<sup>2</sup>, MICHELA TAUFER<sup>3</sup>

<sup>1</sup>Thomas Jefferson University, Philadelphia, PA, USA

<sup>2</sup>University of Michigan, Ann Arbor, MI, USA

<sup>3</sup>University of Delaware, Newark, DE, USA

## Definition

Predict the final atomic resolution structure of a protein–protein complex starting from the coordinates of the unbound conformation of each component protein.

## Discussion

### Introduction

Most successful approaches to protein–protein docking use multistage hierarchical methods typically consisting of an initial global rigid body search (or a simplified reduced search based on known information) to identify a number of candidate protein–protein complexes followed by a more sophisticated refinement procedure. After searching, filtering, and refining, the predicted complexes are rescored based on a sophisticated scoring function for selecting the lowest energy complex. The native protein–protein complex is at the global free energy minimum; therefore those models with the lowest (free) energy should be the most similar to the native complex.

### Approaches to Conformational Searching

#### Exhaustive Rigid Body Searching

The first step in docking two known protein structures is to generate configurations of protein–protein complexes. Traditionally, the generation of conformations

consists of large-scale simulations of independent jobs and thus can be performed in parallel on parallel or distributed systems such as clusters, volunteer computing systems, grid computing systems, or cloud computing platforms. Treating the proteins as rigid bodies is the most computationally efficient method and will be discussed in this section (incorporation of increasing levels of protein flexibility will be covered later). There are six degrees of freedom (three rotation and three translation) that must be explored for an exhaustive search of complex geometries.

The interactions that dominate the favorable binding energy of a protein–protein complex are interactions between the surface residues of the proteins. Therefore, one can naively assume that the most favorable complex geometries are those that bury the largest amount of surface area at the interface. This concept leads to the notion of surface complementarity (i.e., bulges on the surface of Protein A align with a valley on the surface of the Protein B), which is a phenomena observed in many experimentally determined protein complexes. To identify those complexes that maximize the interfacial surface area, a method of describing the individual protein surfaces must first be addressed. An effective method is to project each protein onto a 3D grid and identify which grid points lie at the protein surface and which in the protein interior, as was first described by Katchalski-Katzir et al. [1]. The grid can be represented by a function  $a$  when Protein A is placed in the grid, and function  $b$  when Protein B is placed in the grid, where

$$a_{l,m,n} = \begin{cases} 1 & \text{surface points} \\ << 0 & \text{interior points} \\ 0 & \text{exterior points} \end{cases}$$

and

$$b_{l,m,n} = \begin{cases} 1 & \text{surface points} \\ > 0 & \text{interior points} \\ 0 & \text{exterior points,} \end{cases}$$

where  $l$ ,  $m$ , and  $n$  are the grid indices. With these definitions, a correlation function can be calculated between the functions  $a$  and  $b$  as follows:

$$c_{\alpha,\beta,\gamma} = \sum_l \sum_m \sum_n a_{l,m,n} \cdot b_{l+\alpha,m+\beta,n+\gamma} \quad (1)$$

where  $\{\alpha, \beta, \gamma\}$  is a shift vector applied to Protein B. The value of  $c$  is zero when there is no contact between the proteins, is negative when there is overlap of protein interior regions, and is positive when the surface regions overlap. Complexes with strong positive values have the largest surface complementarity and are retained for further assessment.

The evaluation of  $c$  for a given configuration, requires  $O(N^3)$  computations (for a grid with dimensions  $N \times N \times N$ ); searching all translations of Protein B (all shift vectors) requires  $O(N^6)$  computations. A discrete Fourier transform (DFT) of Eq. 1 can be performed and the inverse transform allows  $c$  to be reexpressed as

$$c_{\alpha,\beta,\gamma} = \frac{1}{N^3} \sum_{o=1}^N \sum_{p=1}^N \sum_{q=1}^N \exp\left[\frac{2\pi i(o\alpha + p\beta + q\gamma)}{N}\right] \cdot C_{o,p,q} \quad (2)$$

where  $C$  is the DFT of  $c$ , defined by

$$C_{o,p,q} = A_{o,p,q}^* \cdot B_{o,p,q},$$

where  $B$  is the DFT of  $b$  and  $A^*$  is the complex conjugate of the DFT of  $a$ . Using a fast Fourier transform algorithm (FFT) reduces the  $O(N^6)$  computations to  $O(N^3 \cdot \ln[N^3])$ . Calculating  $c$  for all values of the shift vector on Protein B only accounts for translational degrees of freedom for a given relative rotation. To fully search the 6D space, Protein B must be incrementally rotated through the three Euler angles that define its relative rotation. Therefore the calculation in Eq. 2 is performed  $(360 \times 360 \times 180)/D^3$  times, where  $D$  is the angle increment.

Another method for finding surface complementarity uses spherical harmonics as the basis functions for describing the protein surfaces, instead of a Cartesian 3D grid, and was introduced by Ritchie and Kemp [2]. In this representation, the natural degrees of freedom are five Euler angles (two for Protein A, three for Protein B) and the center of mass separation of the two proteins. The shape complementarity can be computed via spherical polar Fourier correlations. This method exploits the property of spherical harmonics that they transform among themselves under rotation, which results in a more computationally efficient algorithm than the 3D Cartesian grid FFT method.

The correlation function above only accounts for the surface complementarity and does not account for the chemical nature of the interaction. However, one can use a more complex function that takes into account

electrostatics, van der Waals, and/or desolvation effects, which can be mapped onto the grid as well. The details of these different “scoring functions” are covered in the subsequent sections.

### Reduced Search Space Methods

Both the FFT and the spherical polar Fourier method are designed for an exhaustive search of the 6D space defined by the rigid body rotations and translations of one protein about the other fixed protein. In many instances, additional information about the system may be available which can narrow the search space. For example, when the binding site region of one protein is known, a targeted search can be performed about that site, significantly reducing the search space. A rigid body search is still performed in the initial phase, but the reduced space makes methods such as Monte Carlo and genetic algorithms (GA) viable options. In the Monte Carlo methods, an interaction potential is used to calculate an interaction energy; trial moves are accepted or rejected based on the Metropolis criterion. The programs RosettaDock [3] and ICM-DISCO [4] utilize Monte Carlo in their docking procedures. Neither RosettaDock nor ICM-DISCO use a GA, but in other approaches either a traditional GA or variations (e.g., Lamarckian GA) have been used to drive the search for new low-energy conformations in which the “chromosomes” of the GA consist of the relative positions, rotations, and orientations of one or both proteins. For example, GA can be used to move the surface of one protein relative to the other and locate the area of greatest surface complementarity between the two. Search methods using GA have been widely evaluated in several protein–ligand docking programs, e.g., AutoDock and DOCK.

The incorporation of NMR Nuclear Overhauser Effect (NOE) data, which indicates which residues are interacting, makes solving the docking problem much more tractable. Other more ambiguous data, such as chemical shift perturbations, residual dipolar couplings, and mutagenesis (i.e., alanine scanning), can also be incorporated. The program HADDOCK [5] uses these data sets in the form of Ambiguous Interaction Restraints (AIR)s. An AIR restraint is satisfied if an interface residue on one protein interacts with any interface residue on the other protein. The HADDOCK search method generates random orientations and does

a rigid body minimization, followed by a torsion angle-simulated annealing protocol that introduces both side chain and backbone flexibility. Cartesian space molecular dynamics are then conducted in explicit solvent. A newer approach toward reducing the search space is the prediction of protein active sites using bioinformatics techniques. While it is still difficult to reliably predict a protein “hot spot,” further development of these techniques provides a promising avenue of future study.

### Side-Chain Refinement

In some protein–protein complexation events, the interprotein contacts can induce large-scale conformation changes in the individual protein configurations. These complexes are the most difficult to predict through docking methods and require the proteins to be fully flexible, greatly increasing the configurational search space that must be sufficiently sampled, all of which comes at a great computational cost. In most complexes, just small local deviations from the individual protein crystal structures occur. The majority of these small configurational changes occur in the residue side chains, while the protein backbone remains (nearly) rigid. In multistage docking methods, after a favorable binding geometry is determined from rigid docking, the residue side chains can be (re)built and optimized. The side-chain configurations can be limited to a discrete set of states, derived from known protein structures. From this precalculated rotamer library of low-energy states, the global minimum of side-chain configurations is sought. A combinatorial approach can be used to sample all states, or more efficient algorithms can be used. Dead end elimination is one such algorithm, which removes possible rotamer states that are determined as not being present in the global energy minimum state. Full flexibility of the side chains can be introduced after rotamer optimization, via a molecular mechanics method, to sample deviations from the rotamer library configuration.

### Incorporation of Protein Flexibility in Protein–Protein Docking

Under physiological solution conditions, it is well known that proteins are dynamic rather than being entirely static and exhibit flexible motions that are thermally accessible within their thermodynamically favorable native-state topology. The range of accessible

motions scales from minor side-chain rearrangements and minor flexible loop backbone motions to more substantial backbone deviations (usually within 1.0–1.5 Å C<sub>a</sub>-RMSD) including larger-scale “shear motions” and “hinge motions.” The most significant and complicated rearrangements upon complex formation seem to involve partial unfolding and refolding of flexible segments of the protein (usually termini). Although many protein–protein complexes exhibit minor deviations in the unbound to bound conformation of the component proteins, it appears that a modest number of protein–protein complexes undergo significant rearrangement upon complex formation. Therefore, incorporation of protein flexibility for each protein in a given complex is an important aspect required for accurate predictions of these challenging complexes.

Incorporating protein flexibility into the docking search is a significant computational challenge; many different approaches and algorithms are currently being explored and evaluated. Most successful approaches incorporate some level of backbone flexibility at the level of the global search. This is because even modest backbone rearrangements have been shown to have a significant outcome on the identification of the native protein–protein interface. Therefore, most approaches first generate an ensemble of discrete flexible conformations, use this ensemble for the global search phase, and identify the most likely solutions before moving forward to more detailed model refinements and final scoring of complexes. One way of reducing the search space is to consider only one of the two binding partners to be flexible and only a small ensemble of the more flexible binding partner docking to a representative structure of the other binding partner.

There are numerous approaches to generate an ensemble of structures for ensemble docking that incorporate backbone deviations. The ensemble can be taken from an experimental NMR ensemble or from multiple diverse conformations from X-ray crystallography. In many cases, where there is experimental structural information, there remains insufficient structural diversity in the known structures. If some conformational diversity does exist in experimental structures, it is then possible to use sophisticated tools (e.g., DynDom, HingeFind, FlexProt) to compare the known structure for the identification of flexible loops, as well as shear and hinge motions. It is also possible to use methods

like targeted molecular dynamics to generate a continuous trajectory of conformations that connect the known experimental conformations.

In the more generic case, where only one unbound conformational state is known, it is necessary to generate an ensemble of discrete flexible conformations using computational methods that aim to accurately represent the assessable states of the flexible protein. To this end, it is possible to predict flexible segments using molecular framework approach algorithms (FIRST) and hinge-detection algorithms. Following this approach, conformers can then be generated by sampling a limited set of flexible degrees of freedom to reduce the search space. Alternatively, diverse conformations can be generated directly from physical approaches that more fully sample the available degrees of freedom to the entire protein. These approaches include Molecular Dynamics (MD), MD methods that employ advanced conformational sampling techniques, Essential Dynamics (Principle Component Analysis), and variations of Normal Mode Analysis (NMA).

Standard MD techniques are based on describing the all-atom structure of the system with a detailed force field and modeling dynamics by numerical integration of Newton's equations of motion. MD-simulated annealing conformational searches have been shown to be quite effective in protein–ligand docking (CDOCKER). However, using MD in this way for protein–protein docking is not feasible due to the huge differences in the search-space (geometry of a protein–ligand binding site vs. protein–protein interface) and the number of flexible degrees of freedom. Standard MD methods can explore local minor conformational changes of a protein including side-chain rearrangements and minor backbone relaxations on the picoseconds to nanosecond timescale. However, large-scale changes in protein conformational space occur on longer experimental timescales (microseconds and milliseconds to seconds) and are separated by large energy barriers. Other advanced sampling techniques (simulated annealing, biased methods, torsion angle dynamics, replica exchange) can be coupled with MD to allow for more rapid crossing of energy barriers and sampling alternative low-energy stable conformations. The conformational space sampled by these MD methods can be clustered in Cartesian space to identify low-energy representative conformations.

Alternatively, any given set of generated conformations (such as the conformational space explored by MD) can also be analyzed using Essential Dynamics that is also known as Principle Component Analysis (PCA). In PCA, a square covariance matrix is constructed from the conformational space describing the deviation of each atom coordinate from the average position. When this matrix is diagonalized, the largest eigenvalues represent the “principle components” of the proteins flexibility, where the direction of motion is described by the eigenvector and the amplitude of motion by the eigenvalue. Linear combinations of a subset of the most important eigenvectors can be used to generate an ensemble of “eigenstructures” for docking (CONCORD and Dynamite).

Normal Mode Analysis (NMA) is a method for analytically describing the thermally available deviations from a given equilibrium reference structure within the harmonic approximation. For a given potential energy function describing the system around a minimum energy conformation, a Hessian matrix can be constructed from the mass-weighted second derivatives of the potential energy. When this matrix is diagonalized, the eigenvectors of the matrix are the “normal modes” of the proteins flexibility, where the direction of motion is described by the eigenvector and the amplitude of motion by the eigenvalue. Linear combinations of a subset of the most important eigenvectors can be used to generate “eigenstructures” for docking. Several studies with both normal mode analysis and elastic network normal mode analysis have demonstrated that a few of the low-frequency modes are able to successfully describe experimentally observed large-scale motions of proteins. The set of low-frequency normal modes can also be used for predicting hinge motions (HingeProt) and for estimating the conformational energy penalty that should be associated with a given conformational change. Tama and Sanejouand have pointed out one potential issue which is that in some cases, the normal modes calculated from open conformations of proteins correlate better with observed conformational changes than those calculated from closed and compact conformations [6]. To avoid this problem, May and Zacharias calculate normal modes from the starting structure, generate new initial conformations along deformations of normal mode eigenvectors, and then recalculate the normal modes assuming the new structure is in

equilibrium in an iterative fashion [7]. Another potential issue is that conformational changes involving loop movements are associated with high-frequency rather than low-frequency modes, so new approaches aim to explore and identify conformationally relevant “normal modes” following the observations of Abagyan and coworkers [8].

## Scoring Functions for Protein–Protein Docking

As most protein–protein docking approaches use multistage hierarchical methods, scoring and ranking of putative conformations also typically occur at different steps, and have different requirements for accuracy and speed. The first step of scoring is usually simultaneous with a global or reduced rigid body search where a specified number of favorable complexes are selected for additional refinement. Following this refinement, the most accurate scoring functions are applied to select the most favorable complexes for the final prediction. If there is insufficient accuracy in the first step of discrimination, then native-like complexes are never refined and are missed in the final step of more accurate scoring of refined complexes. On the other hand, if the first step of scoring is too computationally expensive, it limits the amount of conformational space that can be searched in the initial phases.

In the global rigid body search phase (6D search of translational/rotational space), experience has shown that allowing some steric overlap between the surfaces of the two proteins is necessary for success; thus some minor or significant rearrangement of the binding partners may be required. This is handled differently in geometric complementarity-based searching strategies like geometric hashing depending on how the surfaces match or geometric complementarity is calculated. Although not all of the most common search strategies employ surface-matching/geometric complementarity as the primary scoring metric during the sampling phase, many successful protocols (BIGGER, ClusPro, 3D-Dock, DOT, Molfit, PatchDock, SKE-DOCK, SmoothDock, and ZDOCK) use various surface-matching/geometric complementarity metrics in combination with other metrics such as electrostatics and desolvation to determine which complexes should move into the refinement phase. For molecular mechanics-based force fields, allowing some steric

overlap is usually accomplished by using a “soft” or smoothed force field with reduced vdW repulsion. In small-molecule protein–ligand docking, it is extremely common to use a soft vdW potential early in the conformational search, and then in the refinement procedure introduce a more “hard” or standard Lennard–Jones potential. This practice has also been incorporated in some variations of protein–protein docking (ROTAFIT).

Although it is true that exposed hydrophobic patches of residues on a protein surface often indicate a protein–protein interaction surface, several studies that have analyzed experimental protein–protein interfaces show that there is no one single determining feature for prediction of the correct native-like interface. Although some propensities for certain residues can be detected at a sequence level, there is no single interface property (i.e., total buried surface area, number of hydrogen bonds, hydrophobicity, electrostatic complementarity, predicted desolvation energy, residue side-chain rotamers) that is a strong predictor of the native interface over a variety of protein complexes. For this reason, most successful scoring functions designed for correct identification of native protein–protein interfaces are increasingly complex and sophisticated, and involve a combination of these types of interface properties (e.g., shape complementarity, hydrophobic interactions and electrostatics). In general, there are three generic categories of scoring functions: (1) molecular mechanics, (2) knowledge-based, and (3) empirical scoring functions. Research groups have demonstrated success and improvements in native-like geometry discrimination using various flavors and combinations of all three of these types of scoring functions.

Most molecular mechanics force fields are quite similar in their overall generic form. The total potential energy of the system is the pair-wise sum of van der Waal interactions modeled from the Lennard–Jones potential, electrostatic interactions, and the sum of all the internal degrees of freedom (bonds, angles, and torsions) that describe the molecular geometry of the bonded atoms:

$$V = \sum_{\text{bonds}} k_b(b - b_0) + \sum_{\text{angles}} k_\theta(\theta - \theta_0) + \sum_{\text{dihedrals}} k_\phi[1 + \cos(n\phi - \delta)]$$

$$+ \sum_{\text{nonbonded}} \varepsilon_{\text{vdW}} \left[ \left( \frac{R_{\min_{ij}}}{r_{ij}} \right)^{12} - \left( \frac{R_{\min_{ij}}}{r_{ij}} \right)^6 \right] + \frac{q_i q_j}{\varepsilon_{\text{elec}} r_{ij}} \quad (3)$$

The most common molecular mechanics force field in the protein–protein docking field is CHARMM. Several forms of implicit solvation potential energy can also be included in the CHARMM potential, which are based on continuum electrostatics theories including Poisson–Boltzmann, generalized Born, and more simplified representations that are analytical approximations of these. Including these terms can allow for a more rigorous calculation of the desolvation energy upon formation of a protein–protein interface. The electrostatics of a protein–protein interface is a complicated balance between favorable and unfavorable electrostatic complementarity (e.g., H-bonds) and (un)favorable desolvation energy. Vajda and coworkers have determined that at the interface it is more important to put a larger weight on the vdW terms [9].

The field of knowledge-based scoring functions has its origin in the protein-folding field (for the correct identification of native-like folds), and is based on forming statistical potentials from inverse Boltzmann weighting of what is statistically observed in native-like structures. One of the most basic forms of a knowledge-based potential for protein–protein docking is a potential based on the statistical preference of certain residue–residue pairs across the interface. Additional variations of this include a wide range of sophisticated atom–atom contact potentials and also distance-dependent atom pair potentials. One widely used knowledge-based potential of this type is the “atomic contact potential” (ACP) [10] which is an atom–atom extension of the widely used Miyazawa and Jernigan potential [11]. Delisi and coworkers develop the ACP by considering 18 protein atom types and a contact radii of 6.0 Å where the total contact energy of the interface  $E_c$  is defined as

$$E_c = \sum_{i=1}^{18} \sum_{j=1}^{18} E_{ij} n_{ij} \quad (4)$$

where  $E_{ij}$  is the absolute contact energy between atoms  $i$  and  $j$  shown in Eq. 5:

$$E_{ij} = e_{ij} + E_{i0} + E_{0j} - E_{00}. \quad (5)$$

The effective contact energies  $e_{ij}$  can be written in terms of the observed contact numbers in a database where  $\bar{n}_{ij}$  is the number of  $i-j$  contacts in the most probable distribution in an unbiased sample:

$$e_{ij} = -\ln \left( \frac{\bar{n}_{ij}\bar{n}_{00}}{\bar{n}_{i0}\bar{n}_{0j}} \right). \quad (6)$$

It has also been possible to express variations of such atomic contact potentials and variations of distance-dependent pair potentials into functional forms that allow their incorporation into FFT methods such as ZDOCK and PIPER. Incorporation of such potentials has been shown to improve the discrimination of native-like geometries over decoys.

Empirical scoring functions attempt to approximate the relative statistical weights from various parameters to optimize performance from regression-based training and validation studies. One of the best examples of these in protein–protein docking is the complicated scoring function employed in RosettaDock. As outlined in Gray et al. [3], the same functional form has different optimized weights during three phases of the algorithm, i.e., packing, minimization, and final discrimination. In the potential for the discrimination phase, there are 11 terms that all have optimized weights (repulsive vdW, attractive vdW, surface area solvation, Gaussian solvent-exclusion, rotamer probability, hydrogen bonding, residue pair probability, electrostatic short-range repulsion, electrostatic short-range attraction, electrostatic long-range repulsion, and electrostatic long-range attraction) so the final version of the functional form is

$$\begin{aligned} \text{score} = & W_{\text{vdw-atr}} S_{\text{vdw-atr}} + W_{\text{vdw-rep}} S_{\text{vdw-rep}} + W_{\text{sol}} S_{\text{sol}} \\ & + W_{\text{sasa}} S_{\text{sasa}} + W_{\text{hb}} S_{\text{hb}} + W_{\text{rotamer}} S_{\text{rotamer}} \\ & + W_{\text{pair}} S_{\text{pair}} + W_{\text{elec sr-rep}} S_{\text{elec sr-rep}} \\ & + W_{\text{elec sr-atr}} S_{\text{elec sr-atr}} + W_{\text{elec lr-rep}} S_{\text{elec lr-rep}} \\ & + W_{\text{elec lr-atr}} S_{\text{elec lr-atr}} \end{aligned} \quad (7)$$

In the RosettaDock algorithm, this empirical functional form is accurate enough to successfully repack the side chains on the protein–protein interface during the refinement and discrimination phases. Many other empirical scoring functions are employed at the final discrimination step in other docking approaches. Another example is FastContact that rapidly estimates the vdW, electrostatic, and desolvation components of

the free energy using an empirical contact potential for the desolvation contribution.

## Critical Assessments of Protein–Protein Docking Methods

The protein–protein docking field has significantly benefited from the Critical Assessment of Predicted Interactions (CAPRI) experiment. CAPRI is a community-wide blind prediction experiment, where experimentalists provide newly solved protein–protein complexes and withhold the coordinates. CAPRI participants are given the 3D coordinates of each unbound subunit and predictions are due 6–8 weeks later. Using available experimental data in the literature (e.g., mutations that may indicate the location of the binding site) is also allowed, so some targets offer assessments of protein–protein docking where some information is known which can limit the search space, while for most of the targets no information is available. Each participating group is allowed to submit 10 top-ranked models of the protein–protein complex. The accuracies of the predictions are categorized as: (1) high, (2) medium, (3) acceptable, and (4) incorrect according to several evaluation metrics that are summarized in Table 1. The three primary metrics are: (1)  $C_\alpha$  RMSD of the ligand (smaller protein) from the native complex structure after the superposition of the receptor (larger protein), (2)  $C_\alpha$  RMSD of the backbone of the interface residues, and (3) the percentage of native contacts on the interaction interface.

The initial rounds 1–5 of CAPRI occurred from 2001 to 2004 and the results have been published in the literature [12, 13]. The next rounds 6–12 occurred from 2005 to 2007 and these results have also been published [14]. The most recent rounds 13–19 were recently evaluated in a CAPRI meeting held in Barcelona, Spain, in December 2009. Over the 19 rounds of CAPRI so far, 42 targets

**Protein Docking.** Table 1 CAPRI evaluation metrics for predicted protein–protein complexes

| Rank        | Ligand ( $C_\alpha$ RMSD) | Interface ( $C_\alpha$ RMSD) | Interface $f_{\text{Nat. Cont.}}$ |
|-------------|---------------------------|------------------------------|-----------------------------------|
| High***     | $\leq 1.0$                | $\leq 1.0$                   | $\geq 0.5$                        |
| Medium**    | 1.0–5.0                   | 1.0–2.0                      | $\geq 0.3$                        |
| Acceptable* | 5.0–10.0                  | 2.0–4.0                      | $\geq 0.1$                        |
| Incorrect   |                           |                              | $< 0.1$                           |

have been released and predicted by various groups, although some of the targets were canceled during the evaluation period. Over rounds 1–2, the best performing methods were: ICM-DISCO, SmoothDock, Molfit, 3D-Dock, and DOT [12]. Over rounds 3–5 the best overall performance was from ICM-DISCO, PatchDock, ZDOCK, FT-Dock, RosettaDock, and SmoothDock [13]. Over rounds 6–12, the best overall performance was from ZDOCK, HADDOCK, Molfit, 3D-Dock, ClusProt, RosettaDock, and ICM-DISCO [14]. From the most recent rounds 13–19, which were evaluated at the Barcelona meeting, it was shown that some of the Web servers are also now performing as well as some of the other participating groups. The best performing Web servers were ClusPro and PatchDock over rounds 6–12. The performance of the best performing Web servers that participated in rounds 13–19 is shown in Table 2, along with some of the other most popular Web servers. The improved performance of some of the Web servers is very important to the experimental community, as many groups use these servers to predict likely binding modes of protein complexes of interest. These top-ranked binding modes are then often used by experimental groups to design experiments aimed at validating the complex interface using experimental methods such as site-directed mutagenesis, enzymatic proteolysis, mass spectrometry, crosslinking, and FRET.

**Protein Docking. Table 2** Protein–protein docking Web servers and performance in CAPRI rounds 13–19 (CAPRI meeting Barcelona Dec 2009)

|                           | Web Servers (CAPRI rounds 13–19)                                                                                                                                                       | Total | High | Medium |
|---------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|------|--------|
| 1                         | CLUSPRO (Vajda group, Boston University, Boston MA) <a href="http://cluspro.bu.edu">http://cluspro.bu.edu</a>                                                                          | 5     | 1*** | 3**    |
| 2                         | HADDOCK (Bonvin group, Utrecht University, the Netherlands) <a href="http://haddock.chem.uu.nl">http://haddock.chem.uu.nl</a>                                                          | 4     | 1*** | 1**    |
| 3                         | GRAMM-X (Vakser group, U. of Kansas, Lawrence, KS) <a href="http://vakser.bioinformatics.ku.edu/resources/gramm/grammx">http://vakser.bioinformatics.ku.edu/resources/gramm/grammx</a> | 2     | 2*** | -      |
| 4                         | SKE-DOCK (Takeda-Shitaka group, Kitasato U., Japan) <a href="http://www.pharm.kitasato-u.ac.jp/bmd/files/SKE_DOCK.html">http://www.pharm.kitasato-u.ac.jp/bmd/files/SKE_DOCK.html</a>  | 2     | 1*** | -      |
| 5                         | PatchDock, FiberDock, FireDock (Wofson group, Tel Aviv U.) <a href="http://bioinfo3d.cs.tau.ac.il/PatchDock/">http://bioinfo3d.cs.tau.ac.il/PatchDock/</a>                             | 1     | 1*** | -      |
| 6                         | TOPDOWN (Nakamura group, Inst. for Prot. Res. Osaka Japan) <a href="http://pdbjs6.pdbj.org/TopDown/">http://pdbjs6.pdbj.org/TopDown/</a>                                               | 1     | 1**  | -      |
| <b>Other Web Servers:</b> |                                                                                                                                                                                        |       |      |        |
|                           | ZDOCK (Z. Weng group U. of Massachusetts, Worcester, MA) <a href="http://zdock.bu.edu/">http://zdock.bu.edu/</a>                                                                       |       |      |        |
|                           | RosettaDock (J. Gray group John Hopkins U., Baltimore, MD) <a href="http://rosettadock.graylab.jhu.edu/">http://rosettadock.graylab.jhu.edu/</a>                                       |       |      |        |
|                           | HEX (Ritchie group, Nancy, France) <a href="http://www.loria.fr/~ritchied/hex/">http://www.loria.fr/~ritchied/hex/</a>                                                                 |       |      |        |
|                           | FiberDock (Wofson group, Tel Aviv U.) <a href="http://bioinfo3d.cs.tau.ac.il/FiberDock">http://bioinfo3d.cs.tau.ac.il/FiberDock</a>                                                    |       |      |        |
|                           | 3D-Garden (Sternberg group, Imperial College, London, UK) <a href="http://www.sbg.bio.ic.ac.uk/~3dgarden/">http://www.sbg.bio.ic.ac.uk/~3dgarden/</a>                                  |       |      |        |

The most recent CAPRI rounds have shown that some of the more easy targets that did not involve significant conformational changes could be predicted by many groups and Web servers using a variety of methods. This suggests that there is an emerging consensus among the various docking methods for adequate predictions for the easiest of the test cases. The most challenging targets were clearly those where backbone flexibility was very important, as well as targets where one of the protein complex components needed to be predicted by homology modeling. Another clear lesson is that some limited knowledge from biochemical information, when used to limit the search to a certain space area, dramatically improves predictions for several of the methods. The greatest challenge in the field is clearly the issue of incorporating backbone flexibility into the predictions. One reason why this is challenging is that incorporation of backbone flexibility can further increase the number of false positives by nonnative complexes that score better than the native interface.

## Applications of Protein–Protein Docking and Large-Scale Predictions

Due to the importance of protein–protein interactions in understanding the connections and regulations between biochemical pathways, many high-throughput

proteomics efforts have also been aimed at creating large-scale comprehensive catalogues (an *Interactome*) of experimentally verified protein–protein interactions. Significant progress has been made in this area in the model organism yeast, and similar efforts are underway in many other areas, including important pathogenic organisms and human cancer cell types. Experimental studies of these protein–protein interaction networks in cells have shown that some proteins may interact with up to 100 other proteins. Despite these ongoing experimental efforts, it is clear that structural genomics efforts alone (which systematically determine the experimental 3D structures of high interest proteins) will not be able to determine the structures of all of the detected and important protein–protein interactions and reliable computational prediction of these complexes will be more and more important in the future. To successfully perform large-scale protein–protein docking predictions, it will be necessary to employ homology-based modeling methods. The necessity to rely on homology models as the initial input to protein–protein docking, increases the need for improved accuracy both for the initial homology modeling steps and for the reliable incorporation of flexibility in the protein–protein docking steps.

The European 3D-Repertoire project (<http://www.3drepertoire.org>) aims to experimentally determine the 3D structures of some 100 high-interest protein–protein complexes from yeast that are amenable to X-ray crystallography, NMR spectroscopy, and cryo-electron microscopy methods. Still a large number of the total number of yeast protein–protein interactions (10,000 high confidence complexes from a total of 6,000 genes) will be predicted by computational methods. It is also true that for some fraction of the protein–protein targets, the entire complex itself including the interface may be modeled reliably based on homology to other known complexes that are sufficiently similar in sequence. However, for the vast majority of the targets, this approach is not possible and it becomes necessary to perform protein–protein docking of homology models for each individual component of the complex (given that sufficient templates are available for each component of the complex). The first publication describing this effort came out in 2009, describing the prediction of 3,000 protein–protein complexes starting from 217 experimental structures and 1,023

homology models [15]. The authors used both ZDOCK 3.0 along with the pyDOCK scoring scheme and provided results for both a widely used protein–protein docking benchmark dataset [15]. Despite the fact that the accuracy of the benchmark is only in the range of 11–42% (depending on if the top 1, 3, or 10 complexes are considered), these predictions of 3,000 complexes from the yeast *Interactome* are still a very exciting preliminary step toward more large-scale prediction efforts like this in the future. These preliminary efforts highlight the ongoing need for improved accuracy in protein–protein docking methods.

## Bibliographic Notes and Further Reading

More on genetic algorithms and protein–protein docking: (1) Morris GM, Goodsell DS, Halliday RS, Huey R, Hart WE, Belew RK, Olson AJ (1998) Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function. *J Comput Chem* 19(14):1639–1662. (2) Taylor JS, Burnett RM (2000) DARWIN: A program for docking flexible molecules. *Proteins* 41:173–191. (3) Gardiner EJ, Willett P, Artymiuk PJ (2001) Protein docking using a genetic algorithm. *Proteins* 44:44–56.

Introduction in molecular dynamics: (1) Leach A (2001) Molecular modelling: principles and applications, 2nd edn. Prentice Hall, Englewood Cliffs. (2) Rapaport DC (2004) The art of dynamics simulation, 2nd edn. Cambridge University Press, Cambridge.

Key recent reviews on protein docking: (1) Zacharias M (2010) Accounting for conformational changes during protein–protein docking. *Curr Opin Struct Biol* 20:180–186. (2) Vajda S, Kozakov D (2009) Convergence and combination of methods in protein–protein docking. *Curr Opin Struct Biol* 19:164–170. (3) Andrusier N, Mashinach E, Nussinov R, Wolfson HJ (2008) Principles of flexible protein–protein docking. *Proteins* 73:271–289.

## Bibliography

1. Katchalski-Katzir E, Shariv I, Eisenstein M, Friesem AA, Aflalo C, Vakser IA (1992) Molecular surface recognition: determination of geometric fit between proteins and their ligands by correlation techniques. *Proc Natl Acad Sci* 89:2195–2199
2. Ritchie DW, Kemp GJL (2000) Protein docking using spherical polar Fourier correlations. *Proteins* 39:178–194

3. Gray JJ, Moughon S, Wang C, Schueler-Furman O, Kuhlman B, Rohl CA, Baker D (2003) Protein-protein docking with simultaneous optimization of rigid-body displacement and side-chain conformations. *J Mol Biol* 331:281–299
4. Fernandez-Recio J, Totrov M, Abagyan R (2003) ICM-DISCO Docking by global energy optimization with fully flexible side-chains. *Proteins* 52:113–117
5. Dominguez C, Boelens R, Bonvin MJ (2003) HADDOCK: a protein-protein docking approach based on biochemical or biophysical information. *J Am Chem Soc* 125:1731–1737
6. Tama F, Sanejouand YH (2001) Conformational change of proteins arising from normal mode calculations. *Protein Eng* 14:1–6
7. May A, Zacharias M (2008) Energy minimization in low-frequency normal modes to efficiently allow for global flexibility during systematic protein-protein docking. *Proteins* 70:794–809
8. Cavasotto CN, Kovacs JA, Abagyan RA (2005) Representing receptor flexibility in ligand docking through relevant normal modes. *J Am Chem Soc* 127:9632–9640
9. Camacho CJ, Vajda S (2001) Protein docking along smooth association pathways. *Proc Natl Acad Sci* 98:10636–10641
10. Zhang C, Vasmatzis G, Cornette JL, DeLisi C (1999) Free energy landscapes of encounter complexes in protein-protein association. *Biophys J* 76(3):1166–1178
11. Miyazawa S, Jernigan RL (1985) Estimation of effective inter-residue contact energies from protein crystal structures: quasi-chemical approximation. *Macromolecules* 18:534–552
12. Mendez R, Leplae R, De Maria L, Wodak SJ (2003) Assessment of blind predictions of protein–protein interactions: current status of docking methods. *Proteins* 52:51–67
13. Mendez R, Leplae R, Lensink MF, Wodak SJ (2005) Assessment of CAPRI predictions in rounds 3–5 shows progress in docking procedures. *Proteins* 60:150–169
14. Lensink MF, Wodak SJ, Mendez R (2007) Docking and scoring protein complexes: CAPRI 3rd edition. *Proteins* 69:704–718
15. Mosca R, Pons C, Fernandez-Recio J, Aloy P (2009) Pushing structural information into the yeast interactome by high-throughput protein docking experiments. *PLOS Comp Biol* 5(8):1–13

## Pthreads (POSIX Threads)

► [POSIX Threads \(Pthreads\)](#)

## PVM (Parallel Virtual Machine)

AL GEIST  
Oak Ridge National Laboratory, Oak Ridge, TN, USA

### Synonyms

[Message passing](#)

## Definition

Parallel Virtual Machine (PVM) is a software package that permits a heterogeneous collection of Unix and/or Windows computers hooked together by a network to be used as a single large parallel computer. Thus large computational problems can be solved more cost-effectively by using the aggregate power and memory of many computers. The software is very robust and portable. The source, which is available free from the PVM website, has been compiled on everything from laptops to CRAYs. Because PVM supports fault tolerance and dynamic adaptability, hundreds of sites around the world continue using PVM to solve important scientific, industrial, and medical problems. PVM has also been popular as an educational tool to teach parallel programming because of its simple intuitive user interface.

## Discussion

### Introduction

PVM (Parallel Virtual Machine) is often lumped together with the Message Passing Interface (MPI) standard, because PVM was the precursor to MPI and the PVM developers, most notably, Jack Dongarra started and lead the initial MPI forum that defined the MPI 1.0 standard. But message passing is only a small part of the PVM package. PVM is an integrated set of software tools and libraries that emulates a general-purpose, fault-tolerant, heterogeneous parallel computing framework on interconnected computers of varied architectures and operating systems. As a demonstration at the 1992 Supercomputing conference, PVM was used to hook together all the supercomputers on the exhibit floor into one giant parallel virtual machine. PVM is a byproduct of the heterogeneous-distributed computing research project at Oak Ridge National Laboratory, Emory University, and the University of Tennessee. The programming model upon which PVM is based differs substantially from MPI. The PVM programming model has:

- *Dynamic, user configured, host pool.* The set of computers that make up the virtual machine may themselves be parallel or serial computers and PVM can add or delete machines while programs are running. For example, to add more computing power, or delete hosts that have failed.

- *PVM tasks are dynamic and asynchronous.* The tasks that make up a parallel program do not have to start all together. New tasks can be spawned off in the middle of a computation, tasks can be killed and programs started outside the parallel virtual machine can join PVM and become another task in the dynamic set of tasks that make up the parallel program.
- *Dynamic groups.* Users can define any number of subgroups of PVM tasks that can then perform collective operations such as broadcast data to all members of the group or set a barrier. The members of the groups can change dynamically during a computation (an important feature for fault tolerance). PVM provides calls to manage dynamic groups.
- *Translucent access to hardware.* Application programs may view the hardware environment as an attributeless collection of hosts or may choose to exploit the capabilities of specific machines in the host pool by positioning certain computational tasks on the most appropriate computers. PVM provides calls to convey host attributes to a running application.
- *Transparent heterogeneity support.* A single PVM job can run across a collection of Linux, Windows, and Unix computers, that themselves can be multicore, serial or parallel computers. PVM transparently takes care when transferring data between computers that have different data representations, for example, big Endian and little Endian and different word lengths.
- *Explicit message-passing.* Data is transferred between PVM tasks using a small number of point-to-point and collective message-passing calls. When a sender or receiver is detected to be dead, PVM automatically deletes it from the virtual machine and notifies the application.

The PVM system is composed of two parts. The first part is a daemon, called *pvm<sub>d</sub>*, that resides on all computers making up the virtual machine. PVM is designed so that anyone with a valid login can install this daemon on a machine. When a user wishes to run a PVM application, he first creates a parallel virtual machine by starting PVM with a list of hosts upon which he has already installed *pvm<sub>d</sub>*. Multiple users can configure overlapping virtual machines and each user can execute

several PVM applications simultaneously on the same parallel virtual machine. The second part of the system is a library of PVM interface routines that is linked with the users PVM application. The library contains routines for resource management (modifying the pool of hosts), process control (spawning and killing tasks), managing dynamic task groups, passing messages, and fault tolerance. The basic PVM system supports C, C++, and Fortran languages, but the PVM user community has extended PVM so it can be used with Java, Python, Perl, Lisp, S-lang, R, tk/tcl, and IDL languages.

## Resource Management

The underlying set of hosts that make up a particular PVM instantiation is a dynamic resource. Typically, a host list is just fed into the PVM startup routine to create a custom parallel virtual machine tailored to the problem being solved, but PVM provides much more resource management flexibility. PVM provides *pvm\_addhosts* and *pvm\_delhosts* routines that allow any PVM task to add or delete a set of hosts in the parallel virtual machine at any time. These routines are sometimes used to set up a virtual machine, but more often they are used to increase the flexibility and fault tolerance of a large scientific simulation. These routines allow a simulation to increase the available computing power (adding hosts) if it determines that the problem is getting harder to solve, then to shrink back in size when the problem does not need all the extra resources. Another use of these routines is to increase the fault tolerance of a simulation by having it detect the failure of a host and replacing the failed host on-the-fly with *pvm\_addhosts*.

PVM provides routines to return information about the present set of hosts in the parallel virtual machine. The information includes the host names, their architecture types (from which the individual data representations can be determined), and their relative CPU speed (which can be useful to an intelligent load balancing program). A PVM task can also find out what host it is running on and use this architectural awareness to run algorithms tuned for that architecture.

The PVM resource management system also includes a way to set/get various options that determines how the underlying PVM system works. For example, if the user prefers speed to flexibility, the communication can be set so that it is as fast as MPI. Similarly, if the user

knows that the task groups are going to be static then the underlying system can be told and it will utilize local cache information to speed up all the group routines. If tracing or debugging are desired there are options in PVM to set which tasks to trace, and where to send the trace output. PVM has over a dozen options that can be set and the list is extensible to add additional system hints or options in the future.

## Process Control

PVM provides routines to dynamically spawn and kill tasks and allows tasks to join a running PVM system and to exit a PVM system. The `pvm_spawn` routine is the central function in the PVM process control. The `pvm_spawn` routine starts up one or more copies of an executable file. The executable is usually a PVM program, but it does not have to be. Spawning can launch any command or executable program that the user has permission to execute on the hosts. The `spawn` routine can pass an argument list to the launched executable(s) and can specify where the tasks should be started. If unspecified, then PVM will spread the tasks evenly across the entire virtual machine. `Pvm_spawn` can be called multiple times to start many different types of tasks, which is needed to support functional parallelism. It can be called anytime during program execution to add additional computation tasks, or to replace a failed task, or to add a different type of task in order to create a program that adapts on-the-fly to the problem being solved. Besides being able to place tasks on specific hosts, PVM has a plug-in interface that allows users to plug-in their own task placement/load balancing routines into the parallel virtual machine. The flexibility of the `pvm_spawn` routine allows PVM to support all types of programming methodologies, not just SPMD.

To complement the ability to create new tasks dynamically, PVM provides a routine that allows any task to kill any other task, and a routine to allow a task to exit the PVM system, i.e., to cleanly kill itself. The key to these routines is to be sure that all pending parallel virtual machine operations that involve the particular tasks are taken care of before they are killed.

The PVM process control system has a routine that returns information about the tasks running on PVM at any given moment. The routine, called `pvm_tasks`, has three options: it allows a user's PVM program to

determine information about all tasks in the parallel virtual machine, information about all tasks running on a specific host (useful if there is a fault or a load balance issue), or information about a particular task.

## Message Passing

The PVM communication model provides point-to-point, and collective message-passing routines. It also provides the concept of persistent messages and message handlers. The number of communication routines is quite small compared to the MPI standard, which has over 300 functions. For point-to-point communication PVM only provides asynchronous blocking send, asynchronous blocking receive, and nonblocking receive functions. There is also a nonblocking probe routine to check if a particular message has arrived. For collective communication PVM provides multicast to a set of tasks, broadcast to a user-defined group of tasks, reduce operation across a group, i.e., global max, min, sum, etc., and barrier across a group.

There is a four-step process in sending a PVM message. First the message is “packed.” Each message can contain an arbitrary structure of data types and arrays. For example, a message may have an integer defining how long a vector is, a vector of integers defining the indices of a sparse array, and a vector of double-precision floating point numbers. Each of these data types could be packed into a single message. In the second step the message is sent to its destination. In the third step the message is received by one or more PVM tasks. In the fourth step the message is unpacked. PVM supplies a pack and unpack routine for every data type supplied by computer manufacturers.

Persistent messages were added to the PVM interface in version 3.4. Persistent messages are packed as usual with an arbitrarily complex or simple data structure, but instead of being sent to a destination, they are stored and retrieved by “name.” The sender of the message can specify if other tasks are allowed to update the message, or delete it. The sender also specifies if the message should persist beyond the sender exiting PVM or if it should be cleaned up on exit. Persistent messages have many powerful uses in the dynamic, fault-tolerant environment of PVM’s programming model. It allows a message to be sent to a task that does not exist yet. An example of this use is rendezvous – the first task leaves a persistent message defining where it

can be found and how to attach, another example of this use case is in-memory checkpointing. The feature also allows a message to be stored persistently and retrieved by a set of tasks that will not be defined until sometime in the future. An example of this is a fault in the system affects a set of tasks and they need the stored information to recover. Persistent messages provide a distributed information database for dynamic programs inside PVM. For example, a monitoring or computational steering application can attach to a PVM program and leave information in a named space without having to know anything about the tasks in the PVM program, which will retrieve and act on the information. The concept is similar to tuple space used by the Linda system [1]. Persistent messages are implemented in PVM 3.4 using only four additional routines: `pvm_putinfo()`, `pvm_recvinfo()`, `pvm_delinfo()` to delete a persistent message, and lastly, `pvm_getbboxinfo()` that allows a user to search the persistent message namespace with a regular expression.

User-defined message handlers were also added to PVM in version 3.4. There are no restrictions on the handler function that can be set up. The handler is triggered whenever a message arrives matching the specified context, message tag, and source. This feature has many uses. For examples, it allows users to define new control features inside a parallel virtual machine and it provides the ability to implement active messages to increase communication performance.

## **Dynamic Task Groups**

The dynamic process group functions are built on top of the core PVM routines. There was some debate about how groups should be handled in PVM. The issues include efficiency, fault tolerance, and robustness. There are tradeoffs between static versus dynamic groups and tradeoffs if use of the function calls is restricted to only tasks in a group. In keeping with the PVM philosophy, the group functions are designed to be very general and transparent to the user, at some cost in efficiency. Any PVM task can join or leave any group at any time without having to inform any other task in the affected groups. Tasks can broadcast messages to groups of which they are not members. Tasks can be members of multiple groups. In general, any PVM task can call any of the group functions at any time with only two exceptions: `pvm_lvgroup()` and `pvm_barrier()` which

by their nature require the calling task to be a member of the specified group. To boost performance, PVM provides a user function to freeze a group, which allows the underlying PVM system to treat that group as static from that point forward and exploiting all the efficiency gains afforded by having a static group structure.

Dynamic group functions in PVM include joining a group, leaving a group, getting information about a group such as size and members, broadcasting to all members of a group, doing a reduce across all members of a group with predefined or arbitrary functions, and lastly setting a barrier across a group where tasks in the group wait until all tasks in the group have called the barrier routine.

## **Fault Tolerance**

The parallel virtual machine constantly monitors itself even when no PVM applications are running. If a fault is detected the set of `pvm` coordinate with each other and automatically reconfigure the parallel virtual machine to remove the fault and keep running. The PVM interface also has a notify function that allows a running PVM task to request to be notified of changes in the system including where and what type of fault was detected. Typical notifications are for task exit, host deletion, and a host being added. Note that the latter notification is useful during a recovery phase when replacement resources are being brought into the virtual machine. A fault-tolerant PVM application has many choices about how to utilize the notification function in PVM. The application may have all the tasks notified if a fault occurs (SPMD style), it could have one specially written task dedicated to monitoring and recovery of the application if a fault occurs anywhere (MIMD). It could have a set of tasks that get spawned dynamically with a recovery expertise specific to the notification. It could have a set of monitoring and recovery tasks that back each other up. The flexibility of PVM and its dynamic programming model make it very attractive to groups who need fault-tolerant applications. One real life example – PVM was used to create a monitoring system across a hospital's operating rooms with information from individual patient monitors being fed to a central observation/control room, the system needed to be tolerant as patients were constantly being plugged and unplugged from various resources as they went from prep to operating room to recovery room, etc.

The PVM notification feature is not limited to fault recovery. It can also be used by load-balancing programs that increase and decrease the host pool to meet the changing load of the application over time. It can be used by logging and accounting tasks to keep track of system usage by PVM users. As with many of the PVM functions they are designed to provide the maximum use and flexibility to the user.

## Related Entries

- ▶ [Broadcast](#)
- ▶ [Clusters](#)
- ▶ [Collective Communication](#)
- ▶ [Distributed-Memory Multiprocessor](#)
- ▶ [Fault Tolerance](#)
- ▶ [MPI \(Message Passing Interface\)](#)
- ▶ [Parallel Computing](#)

## Bibliographic Notes and Further Reading

The PVM project began in the summer of 1989 at Oak Ridge National Laboratory when Vaidy Sunderam, a visiting faculty from Emory University, and Al Geist designed and built the prototype PVM 1.0 to explore the concept of heterogeneous parallel computing. This prototype was only used internally at the lab and was not released. Jack Dongarra joined the project in 1990 and saw the value of PVM for the larger community. A robust and hardened version of the prototype was written at the University of Tennessee and publicly released

as PVM 2.0 in March 1991. During the next 2 years PVM rapidly grew in popularity. After user feedback and a number of evolutionary changes (PVM 2.1–2.4), a complete rewrite was undertaken, and PVM 3.0 was released in February 1993. PVM 3.0 continued to evolve and incorporate new features and capabilities to meet the needs of the exponentially growing number of users and applications. In 1999, 10 years after its inception, the PVM feature set was frozen at PVM 3.4. Since then PVM has continued to be supported and new versions released to port to new architectures and operating systems, for example, in 2009, 20 years after PVM started, PVM 3.4.6 was released and distributed freely as has every other version of PVM. For further reading the PVM website (<http://www.csm.ornl.gov>) contains tutorials, tools, and information about the PVM development team. For the past 15 years there has been an annual European conference dedicated to advances to PVM and MPI. The topics and papers from all these conferences can be found at <http://pvmmpl08.ucd.ie/previous>.

## Bibliography

1. Bjornson R, Carriero N, Gelernter D, Leichter J (January 1988) Linda, the portable parallel. Research Report Yale/DCS/RR-520
2. Geist A, Beguelin A, Dongarra J, Jiang W, Manchek R, Sunderam V (1994) PVM: parallel virtual machine. MIT Press, Cambridge, Massachusetts
3. Snir M, Otto S, Huss-Lederman S, Walker D, Dongarra J (1996) MPI: the complete reference. MIT Press, Cambridge, Massachusetts

# Q

## **QCD apeNEXT Machines**

OLIVIER PÈNE  
University de Paris-Sud-XI, Orsay Cedex, France

### **Definition**

apeNEXT is a massively, fully custom, parallel computer conceived and designed by an Italian–German–French collaboration and delivered since 2006. It is the latest computer of the APE series. It is a 3D array of computing nodes with periodic boundary conditions. It is dedicated to the calculations of lattice quantum chromodynamics, which is the “ab initio” computational method for the theory of the strong interactions in subnuclear matter.

### **Discussion**

#### **Introduction**

All the matter surrounding us is made of atoms, which are made up of electrons surrounding an atomic nucleus. The latter nuclei are made up of protons and neutrons, which themselves are made up of quarks. The force responsible for binding quarks into protons, neutrons, and many other such particles, generically named “hadrons,” is the “strong interaction” of subnuclear matter. The same strong force binds the protons and neutrons together to form atomic nuclei. This force is so strong that quarks are “confined” into hadrons, which means that one cannot find in nature free quarks, not bound into hadrons. Only at incredibly high temperature the quarks become deconfined.

The theory of this force was discovered in the 1970s and this discovery was awarded by the Nobel Prize of physics in 2004. It is called “quantum chromodynamics (QCD).” This theory demonstrates that the strong force is mediated by new fundamental particles named “gluons.” However, computing within his theory is a

formidable task. The most rigorous method uses a four dimensional lattice which provides a discretised description of space-time. This technique is called lattice QCD (LQCD). It allows to compute ab initio the masses of particles such as the proton, the neutron, the pion, etc., and their properties. This method is the best since it relies uniquely on the theory itself. However it is highly demanding in terms of needed computing power.

This is why the LQCD community has developed since the mid-1980s a series of dedicated computers, in order to be able to perform these computations at an affordable price. This has been done in the USA, in Japan, in Europe. There have been several generations of these computers. The last generation is the QCDOC designed by a US–Britain collaboration with IBM. This is the ancestor of the BlueGene series of IBM. In continental Europe, the APE series started in Italy in 1984. Nicola Cabibbo [1] and Giorgio Parisi were among the initiators of APE. The first APE was operative in 1986. The next generation, the apecento, started in 1993. The APEmille, the third generation, became operative in 2001. The APE series has been supported in Italy by the Italian institute for nuclear and particle physics (INFN). The third generation of APE, APEmille, was done in collaboration with a German group in DESY-Zeuthen. The fourth APE was named apeNEXT with the additional participation of a French group (Université Paris-sud and IRISA/Rennes). The apeNEXT collaboration has published a series of paper or conference contributions [2–7]. More details can be found on the documentation Web sites [8, 9].

### **The apeNEXT Hardware**

#### **The Computing Node**

Since the operations performed in LQCD are mainly multiplicatons of complex matrices, the computing units are harware devised to work on complex numbers.

For every clock cycle, the arithmetic unit performs the APE normal operation:

$$a * b + c \quad (1)$$

where  $a, b, c$  are complex numbers. Expressed in terms of floating point operations it corresponds to eight flops per clock cycle. The peak computing power is thus eight times the clock frequency per processor. This allows for a low clock frequency (about 140 MHz) for a peak performance of 1.2 Gflop per processor, and consequently a low power consumption and a reasonable heat dissipation: Air cooling is enough. It uses intensively “Very Large Scale Integration.” A word of 128 bits is processed every clock cycle, corresponding either to a double precision complex number or to a very long instruction word. There are no cache, but there role is provided by a large register file (256 words of 128 bits).

Each processor is fully independent with its private memory bank of 256–1024 Mbytes based on standard DDR-SDRAM. The memory stores both data and program. The memory controller receives the input from an address generation unit that allows to perform integer operations independently of the arithmetic unit. Error correction is implemented. The memory access has a latency of 15 clock cycles. Since most often the calculation proceeds via large matrices, i.e., in large arrays, the call for numbers in an array is pipe-lined in such a way that, after the arrival of the first complex number of the array, which costs the latency, the following ones come at the speed of one per cycle. A data buffer close to the arithmetic unit allows for a prefetch of the data, thus minimizing the effect of the latency.

Since both data and instructions are imported from memory there is a risk of conflict. This is minimized by a compression of the microcode containing the instructions, and by an instruction buffer allowing prefetch and storing the critical kernels for repeated operations. Instruction decompression is performed on the fly by the hardware.

### The Torus Network

The network is a custom one, designed for apeNEXT. The computing nodes are organized according to a 3D torus. Every node is directly connected to six neighbors, up and down in three directions. The last one along one direction is connected to the first one, thus producing a

circle. This is fully adapted to LQCD. Indeed the LQCD calculations are performed on a discrete lattice which is a 4D torus. Every lattice site communicates with its eight neighbors and the last one along one line communicates with the first one. Thus it is natural to decompose the LQCD lattice into sublattices located on every computing node. For example, let us consider a LQCD lattice of size  $32^3 * 64$  (where 64 is in the time direction). An apeNEXT rack has 512 processors organized in a  $8^3 * 3D$  torus. It is thus possible to decompose the LQCD lattice into sublattices of size  $4^3 * 64$ . Other combinations are of course possible. Thus every lattice site is located on one computing node, and its neighbors are either on the same computing node or on neighbouring ones.

The hardware allows an access to the memory banks located on the considered computing nodes and to those of the neighboring computing nodes. Each link is bidirectional and moves one byte per clock cycle and the start-up latency is about 20 cycles, i.e., about 150 ns, which is very short. To minimize the effect of the latency via a prefetch of the data from neighboring nodes, six dedicated buffers are used for the data from the six neighboring nodes. The maximum throughput of the network is reached for rather small bursts of 12 words (200 bytes) of data. This is convenient for the LQCD algorithms.

This torus organization allows for a high scalability when the number of processors increases. The parallel implementation is Single Program Multiple Data (SPMD) which means that all the computing nodes perform the same operations simultaneously, including their communication with neighbors. There is no data transfer bottleneck during the computation except for a limited number of cases (e.g., when a global sum has to be performed).

### The Global Features

Every board contains 16 computing nodes. A crate contains 16 boards assembled thanks to a backplane which provides all the communication links. A crate thus contains 256 nodes. A rack contains two crates. It has a peak performance of 0.6 Tflop, a footprint of 1 m<sup>2</sup> only, a power consumption less than 10 kW which allows for air cooling. The apeNEXT system is accessed from a master front-end PC which controls slave PCs (typically

one for two boards). The PCs communicate via custom designed host-interface PCI boards. It uses a simple I2C network for operating system requests, to know the state of the machine, handle errors and exceptions. Input/output operations are handled via a highspeed link, called “7th link,” and proceed via the PCs. Several racks can also be connected together for a large scale calculation.

## The apeNEXT Software

### The TAO Language

Details on the software can be found on the Web site [9]. The series of APE computers use a special high-level language named “TAO.” It is a rather simple language, somehow in the spirit of Fortran. It contains some improvements. For example one can define a mathematical object which is a  $SU(3)$  matrix (i.e., a  $3 \times 3$  unitary complex matrix of determinant 1). Once defined all the operations on these matrices, one can overload the “multiply” symbol:

$$C = A * B \quad (2)$$

where  $C, B, A$  are  $SU(3)$  matrices, means that  $C$  is the matrix product of  $A$  and  $B$ . This is extremely useful, knowing that the  $SU(3)$  matrices are basic objects in LQCD.

TAO language contains a few additional commands related to the SPMD parallelism. For example, an “if” is different whether it applies to the full system (in which case it induces a branching) or to one node (in which case it will go through both branches with an appropriate mask). The transfer of data between nodes is simply performed by address numbers which are understood as being on the remote node.

Finally TAO is associated with a preprocessing language, “ZZ.” It allows to define new data types, operations between these types and to write loops which are unrolled at compile time.

### The Compiler

The compiler also has been written by the apeNEXT collaboration. It contains several successive modules. The Tao code is first translated into the user level Assembly language named SASM. The latter is then converted into the microcode level Assembly, MASM. An optimizer

named SOFAN [10] then eliminates the useless instructions or combines several instructions. It turns the original MASM code into an improved MASM code. The “shaker,” then, converts the MASM into the microcode and reorganizes the program to eliminate, as much as possible, useless cycles, e.g., while waiting for data from memory.

The compiler decides how to use the register files in order to minimize the latency of data retrieval from memory or from other nodes. The programmer can also take some decisions concerning the use of the registers, e.g., demanding some often reused data to stay in registers.

More generally it is noticeable that, the architecture being rather simple and transparent, the programmer can know very exactly what happens during a calculation including every step of data transfer. Having thus a complete and clear understanding of how the computing time is used, one can efficiently improve the code’s efficiency. It also helps when debugging.

Of course, the software also contains a custom operating system which is adapted to the architecture.

## The Theoretical Physics Research Thanks to apeNEXT

The apeNEXT is the latest computer of the APE series, after APE, APEcento, and APEmille. The apeNEXT racks have been installed in 2006 in three places: In an international computational laboratory in the university of Rome “La Sapienza” (13 racks belonging to the INFN and used by the majority of Italian LQCD groups, plus 2 racks supported by the French CNRS and National Research Agency (ANR)). Six racks are in the Bielefeld university in Germany and four racks in DESY-Zeuthen (close to Berlin).

During these 3 years, the apeNEXT computers have been intensively used for research in theoretical physics. The major difficulty in LQCD calculations is the presence of very light quarks in nature. The computing time increases very fast when the quarks become lighter. apeNEXT and other computers of that generation allowed the LQCD community to really perform calculation with quarks significantly lighter than what was possible before, although, it was not enough to reach a quark mass as light as in nature. This is left to the next generation of computers.

The LQCD group in Bielefeld has used its apeNEXT's to study the properties of QCD at finite temperature, a situation which has existed after the big bang and is still observable in accelerators in which very heavy ions collide at very large energy. They have among other quantities, computed at which temperature a phase transition takes place which deconfines the quarks and gluons, i.e., liberates them from being bound into hadrons.

The apeNEXT computers in Zeuthen and Rome have been mainly devoted to the zero temperature case. They have studied the properties of the proton, neutron, and the pion which constitute the atomic nuclei and represent more than 90% of the visible mass of our universe. They have also devoted many studies to states containing the heavy quarks named "charm quarks" and "beauty quarks." These quarks are not frequent in our everyday surrounding, they have to be produced by accelerators of particles. They have been extensively studied experimentally in dedicated accelerators because they allow an accurate test of the standard model of particle physics, which is our present understanding of the fundamental laws of nature. They also open a window toward an understanding of the laws of nature at still smaller scales than the ones we already understand. These experiments in combination with theoretical studies, mainly based on LQCD, have confirmed the "Cabibbo–Kobayashi–Maskawa (CKM)" model which describes the mixing between different quark species and a mechanism for CP violation (asymmetry between particles and their antiparticles). Thanks to this vast experimental and theoretical work, the CKM model has been awarded by the Nobel Prize in 2008.

It is impossible of course to quote all the publications stemming from calculations performed thanks to the apeNEXT computer. They have provided a great help to LQCD in continental Europe. In Italy, they are still today the major tool to perform the heavy LQCD calculations. In France, it was so until fall 2008 when a BlueGene was installed in the CNRS computing center, but they are still under very active use.

## Conclusion

The apeNEXT, the latest computer of the APE series, was, during the last years, the only fully custom high-power computer designed and produced in Europe. All APE computers were created without the help of any

major computer company. The Italian SME "Eurotech" company has built the prototypes of apeNEXT and the market products. The apeNEXT team was essentially an academic collaboration of physicists and computer scientists from Italy, Germany, and France. The funding came from different scientific institutions in these three countries.

This created a rich know-how which was recently reused in the QPACE project building a LQCD dedicated supercomputer using IBM-CELL [11, 12] processors and an apeNEXT inspired network. This know-how is also used in the recent Aurora project [13, 14] of a supercomputer using Intel/Nehalem processors. It is finally used in the petaQCD project of the "Agence Nationale de la Recherche" about software and hardware problems of the petaflop era for LQCD [15]. Beyond the know-how, the community which was created around the apeNEXT project is still alive and may be fruitful in the future.

Finally, the apeNEXT computers played an essential role in allowing the European LQCD community to enter really in the era of lattice calculations with light dynamical quarks.

## Related Entries

- ▶ [IBM Blue Gene Supercomputer](#)
- ▶ [QCDSP and QCDOC Computers](#)

## Bibliographic Notes and Further Reading

A popular and deep presentation of QCD can be found in Frank Wilczek's Nobel lecture [16] or paper in Physics Today named "QCD made simple" [17].

The history of the APE series has been recently summarized by Nicola Cabibbo [1].

The apeNEXT collaboration has published a series of paper or conference contributions [2–7]. More details can be found on the documentation Web sites [8, 9] and [18, 19].

## Bibliography

1. [http://www.ba.infn.it/~apenext/PRESENTATIONS\\_FILES/Cabibbo.pdf](http://www.ba.infn.it/~apenext/PRESENTATIONS_FILES/Cabibbo.pdf)
2. Bodin F et al (2002) Nucl Phys Proc Suppl 106:173, arXiv:hep-lat/0110197
3. Alfieri R et al. APE Collaboration, arXiv:hep-lat/0102011

4. Ammendola R et al (2003) Nucl Phys Proc Suppl 119:1038, arXiv:hep-lat/0211031
5. Bodin F et al (2003) In: The proceedings of 2003 conference for computing in high-energy and nuclear physics (CHEP 03), La Jolla, 24–28 Mar 2003, pp THIT005, arXiv:hep-lat/0306018
6. Bodin F et al (2003) ApeNEXT collaboration. In: The proceedings of 23rd international conference on physics in collision (PIC 2003), Zeuthen, 26–28 Jun 2003, pp FRAPI5, arXiv:hep-lat/0309007
7. Bodin F et al (2005) ApeNEXT collaboration. Nucl Phys Proc Suppl 140:176
8. <http://www.fe.infn.it/fisicacomputazionale/ricerca/ape/NXTPROJECT/doc/root.html>
9. <http://apegate.roma1.infn.it/APE/index.php?go=/documentation.tpl>
10. <http://hpc.desy.de/ape/software/sofan/>
11. Baier H et al (2009) PoS LAT 2009:001. <http://arxiv.org/abs/0911.2174>
12. Baier H et al. arXiv:0810.1559, hep-lat
13. Aurora Science Collaboration (2010) PoS LATTICE 2010:225
14. Abramo A et al. Tailoring petascale computing. In: Proceedings of ISC09, June 200923–26 (Hamburg)
15. <https://www.petaqcd.org/>
16. <http://nobelprize.org/nobelprizes/physics/lauriates/2004/wilczek-lecture.html>
17. [http://www.frankwilczek.com/Wilczek\\_Easy\\_Pieces/](http://www.frankwilczek.com/Wilczek_Easy_Pieces/)
18. Belletti F et al (2006) ApeNEXT collaboration. Nucl Instrum Meth A 559:90
19. Belletti F et al (2006) ApeNEXT collaboration. Comput Sci Eng 8:8

## QCD (Quantum Chromodynamics) Computations

KARL JANSEN  
NIC, DESY Zeuthen, Zeuthen, Germany

### Definition

Quantum Chromodynamics (QCD) is the prime and generally accepted theory of the strong interactions between quarks and gluons. In QCD, there appear intrinsically nonperturbative effects such as the confinement of quarks, chiral symmetry breaking, and topology. These effects can be analyzed by formulating the theory on a four-dimensional space-time lattice and solving it by large-scale numerical simulations. An overview of the present simulation landscape, a physical example and a description of simulation and parallelization aspects of QCD on the lattice is given.

## Discussion

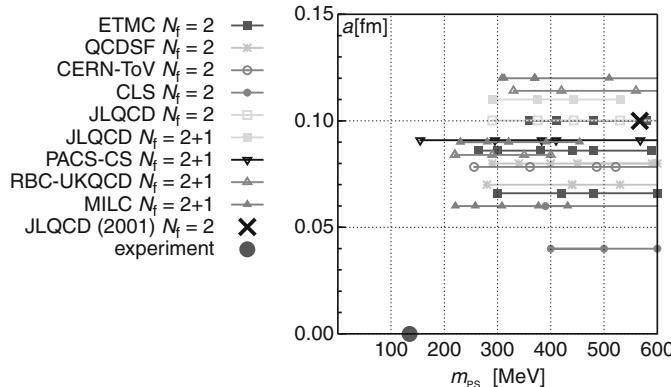
### Introduction

When experiments at large accelerators such as HERA at DESY and LEP and LHC at CERN are performed, in the detectors one observes hadrons such as pions, protons, or neutrons. On the other hand, from the particular signature of these particles as seen in the detectors, it is known that the hadrons are not fundamental particles but that they must have an inner structure.

It is strongly believed nowadays that the constituent particles of all hadrons are the quarks with the gluons being the interaction particles that “glue” the quarks together to perform the bound hadron states, i.e., the particles that are seen in the experiments. The force that binds the quarks and gluons together – the strong interaction – is theoretically described by quantum chromodynamics (QCD). The postulation of QCD is that at very short distances much below 1 fm, the quarks behave as almost free particles that interact only very weakly, a phenomenon called asymptotic freedom. On the other hand, at large distances, at the order of 1 fm, the quarks interact extremely strongly, in fact so strong that they will never be seen as free particles but rather form the observed hadron-bound spectrum. The latter phenomenon is called confinement.

Since the interaction between quarks becomes so strong at large distances, analytical methods such as perturbation theory fail to analyze QCD. A method to nevertheless tackle the problem is to formulate QCD on a four-dimensional, euclidean space-time lattice [1] with a nonzero lattice spacing  $a$ . This setup first of all allows for a rigorous definition of QCD and to address theoretical questions in a conceptually clean and fundamental way. On the other hand, the lattice approach enables theorists to perform numerical simulations [2] and therefore to investigate also problems of nonperturbative nature.

In the past, lattice physicists had to work with a number of limitations when performing numerical simulations because these simulations are extremely computer time expensive, needing petaflop computing and even beyond, a regime of computing power which is just reached today. Therefore, for a long time the quarks were treated as infinitely heavy, since in this situation the computational costs are orders of magnitude smaller. However, this is a crude approximation given



**QCD (Quantum Chromodynamics) Computations.** Fig. 1 The values of the lattice spacing  $a$  and pseudo scalar masses  $m_{PS}$  as employed in typical QCD simulations by various collaborations as (incompletely) listed in the legend. The *blue dot* indicates the physical point where in the continuum limit the pseudo scalar meson assumes its experimentally measured value of  $m_{PS} = 140$  MeV. The *black cross* represents a state of the art simulation by the JLQCD collaboration in 2001

that the up and down quarks have masses of only  $O(\text{MeV})$ . In a next step, only the lightest quark doublets, the up and down quarks, were taken into consideration, although their mass values as used in the simulation have been unphysically large, about two to three times their physical values.

Nowadays, besides the up and down quarks, also the strange quark is included in the simulations. In addition, these simulations are performed in almost physical conditions, having the quark masses close to their physical values, large lattices with about 3 fm linear extent and small values of the lattice spacing such that a continuum limit can be performed. The situation [3] of the change of the simulation landscape is illustrated in Fig. 1. In the figure, the blue dot indicates the physical point, i.e., a zero value of the lattice spacing and a pion mass reaching its physical value of 140 MeV. The black cross represents a state-of-the-art simulation in the year 2001. As can be seen in the graph, most of the simulations currently go well beyond what could be reached in 2001, clearly demonstrating the progress in performing realistic simulations.

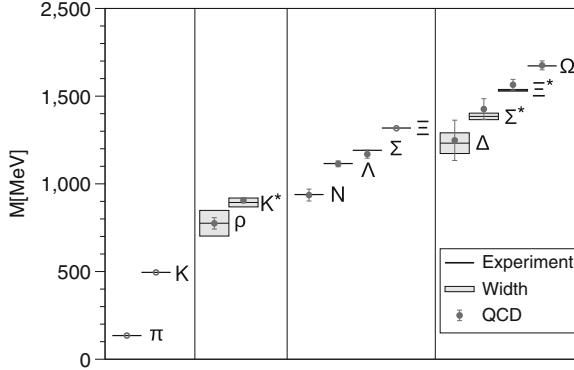
This quite significant change in the situation is due to three main developments: (a) algorithmic breakthroughs [4–7]; (b) machine development; the computing power of the present BlueGene P systems of about 1 Petaflops is even outperforming Moore’s law (c) conceptual developments, such as the use of improved actions which reduce lattice artefacts and the development of nonperturbative renormalization.

As one example of a physics result, the computation of the light hadron spectrum from first principles using lattice simulations [8] is shown in Fig. 2. Using only two input masses, the ones for the pion and the Kaon, all other masses are a result from these ab initio calculations. Note that the lattice data reach in general a few percent accuracy while the experimentally obtained values are partly at the sub-percent level. In addition, some of the states are resonances which would need a special treatment on the lattice. Nevertheless, the good agreement between the experimentally observed and the numerically computed spectrum is thus a nice confirmation that QCD is indeed the correct theoretical description of the strong interaction, although further tests using more physical observables are necessary to eventually consolidate and verify this result.

Results of lattice computations, the progress in the field, and a discussion of many technical aspects are documented in the proceedings of the lattice symposia taking place worldwide on a yearly basis.

## The Principle of Lattice Gauge Theory Calculations

The main target of lattice calculations is to compute expectation values  $\langle \mathcal{O} \rangle$  of physical observables  $\mathcal{O}$ , the hadron masses discussed above being one example. The general problem of such calculations is to solve a specific, high-dimensional integral ( $O(10^{\text{several thousand}})$ ).



**QCD (Quantum Chromodynamics) Computations. Fig. 2**

The light hadron spectrum as obtained in ref. [8]

To illustrate the problem, let us consider a one-dimensional problem in only one variable  $x$ . The eventual aim is the computation of the “expectation value”  $\langle f(x) \rangle$  of a function  $f(x)$ . In analogy to QCD, this amounts to calculate the integral  $\int dx f(x) e^{-S(x)}$  where  $S(x)$  is the so-called action which defines the physical system under consideration. If a very simple form of such an action is taken, namely,  $S(x) = \frac{1}{2}x^2$  then the integral can be solved by generating Gaussian random numbers of unit variance as integration points  $x_i$ . This is the principle of “importance sampling.” With this choice of integration variables  $x_i$ ,  $\langle f(x) \rangle$  is simply given by the sum over all  $x_i$ ,  $\langle f(x) \rangle \approx \frac{1}{N} \sum_i f(x_i)$  where  $N$  is the total number of integration points.

In lattice QCD the same principle of importance sampling is used to compute physical observables but the action used is much more complicated. The form of the action is given as  $S = \sum_{(x,y)} \bar{\Psi}(x) M(x,y) \Psi(y)$ . Here  $(x,y)$  denotes four-dimensional, integer lattice points and the quark fields  $\Psi$  (anti-quark fields  $\bar{\Psi}$ ) are defined at these discrete lattice points. The matrix  $M(x,y)$  describes the interaction between the quark fields. It has to be remarked already at this point that  $M$  is a very large matrix of the order of a hundred million times a hundred million for most advanced simulations nowadays.

However, a big advantage is that the general structure of the matrix  $M$  is such that only quark fields at nearest neighbour points are coupled. The general form of the interaction is

$$\bar{\Psi}^{(i,\alpha)}(x) U^{(i,j)}(x, \mu) \Gamma^{\alpha, \beta} \Psi^{(j,\beta)}(x + a\hat{\mu}). \quad (1)$$

Here, field  $\Psi^{(i,\alpha)}(x)$  is equipped with internal indices  $i = 1, 2, 3$  representing colour and  $\alpha = 1, 2, 3, 4$  representing the Dirac structure. The so-called gauge field  $U^{(i,j)}(x, \mu)$  is a SU(3)-matrix describing the gluon degrees of freedom. It goes beyond the scope of this article to discuss in detail the physical motivation and interpretation of this coupling structure of neighbouring fields and for our purposes it is more important to just look at the mathematical structure of this coupling. Let us remark also that the gauge fields  $U$  interact with themselves which can be expressed through another action  $S_g(U)$  the particular form of which is not important here.

The expectation value of a physical observable corresponds now to solving the integral

$$\langle \mathcal{O} \rangle = \int \mathcal{D}\Psi \mathcal{D}\bar{\Psi} \mathcal{D}U \mathcal{O} e^{-S(U, \bar{\Psi}, \Psi)} \quad (2)$$

where the integration is to be performed over all quark and gluon fields and where the action is given by

$$S(U, \bar{\Psi}, \Psi) = S_g(U) + \sum_{(x,y)} \bar{\Psi}(x) M(x,y) \Psi(y). \quad (3)$$

The action in Eq. 3 contains two free parameters, the quark mass and the gauge coupling. These parameters have to be tuned in order to achieve a particular physical situation.

Before proceeding, another technical difficulty has to be discussed. The quark fields are not represented by normal, commuting numbers but by so-called Grassmann variables which *anti-commute*. As a consequence, the quark fields cannot be dealt with on a computer directly since computers deal with normal numbers only. However, the Grassmann fields can be integrated over leading to an expression (assuming that the matrix  $M$  is such that all the integrations discussed here can be performed, which is, of course, the case for real lattice QCD)

$$\langle \mathcal{O} \rangle = \int \mathcal{D}U \mathcal{O} \det[M(U)] e^{-S_g(U)}. \quad (4)$$

However, given the large size of the matrix  $M$ , computing its determinant is hopeless. The solution of this difficulty is to integrate the determinant in again using ordinary, bosonic fields  $\Phi$  leading to our final expression for a physical observable,

$$\langle \mathcal{O} \rangle = \int \mathcal{D}\Phi^\dagger \mathcal{D}\Phi \mathcal{D}U \mathcal{O} e^{-S(U, \Phi^\dagger, \Phi)} \quad (5)$$

with

$$S(U, \Phi^\dagger, \Phi) = S_g(U) + \Phi^\dagger M^{-1} \Phi. \quad (6)$$

This shows that for the evaluation of the action  $S(U, \Phi^\dagger, \Phi)$  the vector  $X = M^{-1}\Phi$  is needed. The basic numerical problem in lattice QCD is thus to solve a set of linear equations

$$MX = \Phi \quad (7)$$

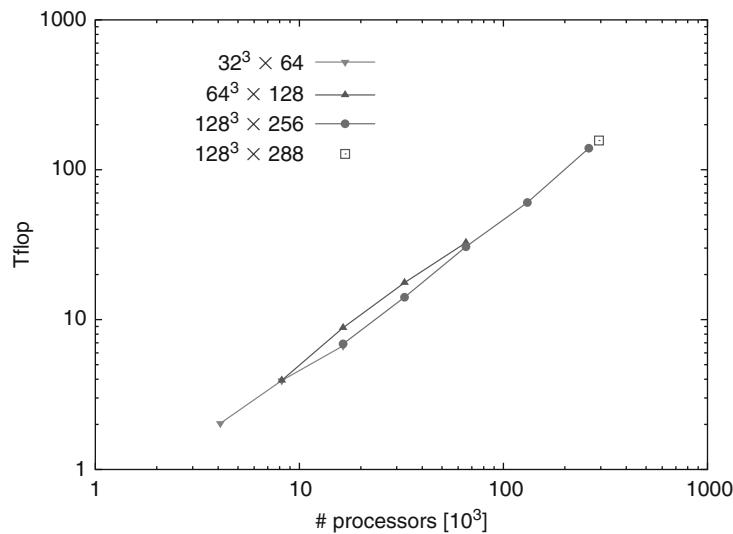
with the fermion matrix  $M$  having the particular interaction form given in Eq. 1 with the mentioned size of hundred million times hundred million. Since  $M$  connects only nearest neighbor points on the lattice, it is a sparse matrix with only the diagonal and a few  $O(10)$  sub-diagonals occupied and the techniques to solve such sparse systems of linear equations [9] can be applied. Moreover, what is actually (hard-) coded is only the application of the fermion matrix  $M$  on a vector and hence  $M$  need not be stored in memory. Nevertheless, the matrix  $M$  is large and a solver, such as, e.g., conjugate gradient (CG) [9], needs several hundred to thousands of iterations to converge to the solution with the desired accuracy of about  $10^{-14}$  (relative precision). Moreover, the computation of the integral of Eq. 5 over the gauge fields  $U$  needs several thousands of these gauge fields and hence Eq. 7 needs to be solved hundred thousands of times to obtain just one physical result at just one value of the action parameters, i.e., the quark mass and

the gauge coupling. In order to have a complete result, simulations at many of these action parameters have to be performed.

## Parallelization

The description of the computational elements given above leads to a very high total computational cost. For example, if a moderately sized lattice of  $24^3 \cdot 48$  is taken, on one rack of a BG/P system, corresponding to 4096 processors, one would get about 200 gauge field configurations per day. A simulation leading to a sufficiently precise estimate of  $\langle \mathcal{O} \rangle$  requires about 5,000 of such configurations and would thus need about 1 month. Clearly, without any parallelization, it would be totally unrealistic to solve this problem.

Fortunately, the structure of the fermion interaction with only nearest neighbor couplings renders the problem of parallelization rather easy. There are essentially two approaches. The first is to use MPI. Here, the total lattice is distributed over the number of desired processors (domain decomposition). The total lattice is then divided into local sub-lattices of size  $l_t \cdot l_x \cdot l_y \cdot l_z$  on a  $N_t \cdot N_x \cdot N_y \cdot N_z$  processor grid. The parallelization strategy is then to equip the fields on the sub-lattice with “borders” which will take the neighboring lattice points needed for the sub-lattice on a particular processor. Before any computation is performed, each processor



**QCD (Quantum Chromodynamics) Computations. Fig. 3** Strong scaling of the lattice fermion matrix on the Jugene BG/P system at the Jülich supercomputer center for three different global lattice volumes

sends the necessary neighbor points to corresponding adjacent processors where these points are then stored in the borders. This concept works well for machines with a very fast communication network with, however, a rather large latency.

Another concept is a direct memory access to neighboring processors. Here, a single datum is sent to a neighboring processor whenever it is needed. In an application code this looks like  $\Phi(x + a\hat{\mu} + mn)$  where “mn” stands for a so-called magic number which addresses the memory location of the field  $\Phi(x + a\hat{\mu})$  but on a neighbouring processor. Realizations of this concept have been made on the dedicated lattice QCD machines APE [10] and QCDOC [11].

Other, new developments are machines with multi-core processors [12] and GPUs. Here, new parallelization concepts have to be developed and the problem of working efficiently on many GPUs in general is still under investigation.

[Figure 3](#) represents an example of the (strong) scaling behaviour of a certain implementation of a lattice QCD code [13]. The total performance in Teraflops is shown for the application of the fermion matrix on three different volumes – a  $32^3 \cdot 64$ , a  $64^3 \cdot 128$ , and a  $128^3 \cdot 256$ . The performance is shown as a function of the number of processors in a double log plot. A clear linear scaling is observed for all volumes used. The single point in [Fig. 3](#) represents the performance measurement for a  $128^3 \cdot 288$  volume which was chosen to use the full 72-rack installation of the BG/P system at the supercomputer center in Jülich. The graph shows that lattice QCD is very well suited for massively parallel machines and shows an almost perfect strong scaling behaviour. Note that lattice QCD applications typically reach 20–30% of the peak performance of the supercomputers used.

The total simulation cost depends strongly on the volume, the pion mass, and the lattice spacing used in the simulations. For a discussion on the scaling of the algorithm employed in lattice QCD applications, I refer to refs. [3, 14]. For quantities, such as some of the hadron masses and decay constants, it can be expected that machines in the multi-Petaflops regime are sufficient to match the experimental accuracy. For other more complicated observables for example connected to scattering states or unstable particles, clearly exaflops computing capabilities are required.

## Bibliography

1. Wilson KG (1974) Confinement of quarks. *Phys Rev D*10:2445–2459
2. Creutz M (1980) Monte Carlo study of quantized SU(2) Gauge theory. *Phys Rev D*21:2308–2315
3. Jansen K (2008) POSCI LATTICE2008, 010, Lattice QCD: a critical status report. [[arXiv:0810.5634 \[hep-lat\]](#)].
4. Lüscher M (2005) Schwarz-preconditioned HMC algorithm for two-flavour lattice QCD. *Comput Phys Commun* 165:199
5. Urbach C, Jansen K, Shindler A, Wenger U (2006) HMC algorithm with multiple time scale integration and mass preconditioning. *Comput Phys Commun* 174:87–98
6. Clark MA, Kennedy AD (2007) Accelerating dynamical fermion computations using the rational hybrid Monte Carlo (RHMC) algorithm with multiple pseudofermion fields. *Phys Rev Lett* 98:051601
7. Lüscher M (2007) Deflation acceleration of lattice QCD simulations. *JHEP* 12:011
8. Dürr S et al (2008) Ab initio determination of light hadron masses. *Science* 322:1224–1227
9. Saad Y (2003) Iterative methods for sparse linear systems, 2nd edn. SIAM, Philadelphia, PA
10. Belletti F et al (2006) Computing for LQCD: apeNEXT. *Comput Sci Eng* 8:18–29
11. Boyle PA et al (2005) QCDOC: project status and first results. *J Phys Conf Ser* 16:129–139
12. Baier H et al QPACE – a QCD parallel computer based on Cell processors, [[arXiv:0911.2174](#)]
13. Jansen K, Urbach C (2009) tmLQCD: a program suite to simulate Wilson Twisted mass Lattice QCD. *Comput Phys Commun* 180:2717–2738
14. Jung C (2009) POSCI LAT2009, 002, Status of dynamical ensemble generation. [[arXiv:1001.0941 \[hep-lat\]](#)].

## QCD Machines

NORMAN H. CHRIST

Columbia University, New York, NY, USA

## Definition

QCD machines are parallel computers whose design and construction are specially tailored to efficiently carry out lattice QCD calculations. Quantum chromodynamics (QCD) is that part of the standard model of particle physics which describes the interactions of quarks and gluons. Lattice QCD is a discrete formulation of this theory which permits its study by numerical methods.

## Discussion

### Introduction

The neutrons and protons which compose the atomic nucleus are themselves made of quarks and gluons which are accurately described by the fundamental theory of quantum chromodynamics (QCD). QCD is a natural generalization of Maxwell's theory of electromagnetism. To a large degree, the theory of electromagnetism can be understood using classical physics with quantum phenomena appearing as important refinements of the classical theory, revealing quantum granularity when only a few quanta are present. This theory is well described by Maxwell's four partial differential equations with quantum corrections that can be computed using quantum perturbation theory.

In contrast, the phenomena of QCD have no relevant classical limit and must be understood by solving the full quantum mechanical problem without recourse to perturbation theory. This challenging theory is best formulated as a Feynman sum over histories or a Wiener integral, which in present applications requires the evaluation of an integral over 100 million variables – a problem that demands the power of highly parallel computing. This large number of variables is only an approximation to the infinite number that would be needed if an integration variable were assigned to each point in the space-time continuum. By approximating the space-time continuum by a four-dimensional grid or lattice of points, the number of integration variables is made finite. Lattice QCD exploits such a grid and evaluates the resulting integral by Monte Carlo methods.

While many important problems require the enormous computing power that potentially results from a large degree of parallelism, lattice QCD has particular features which make it a natural first target for parallelization and support the construction of parallel computers specialized for its solution. In order of importance, these features are:

*Homogeneity.* As a fundamental description of the physics of quarks and gluons in space-time, QCD offers enormous natural parallelism. Since each lattice site must be treated symmetrically, the parallel computation that must be performed on the variables at one site must be the same for every other. This high degree of

homogeneity makes the code simple and leads to perfect load balancing if each processor in a parallel system is assigned the same number of lattice sites.

*Locality.* An important consequence of Einstein's theory of special relativity is that all interactions are local. Only nearby variables in space-time are coupled, and long-range effects come about as changes propagate from one site to the next, finally crossing many sites. Thus, communication between the homogenous processes described in the previous paragraph need only be provided between neighboring processes. This vastly simplifies the needed communication in a parallel computer and supports the simplest of mesh communication fabrics.

*Simplicity.* The basic algorithms needed to stochastically evaluate the QCD path integral are intrinsically simple. The code needed for the early study of only the gluon variables requires a simple heat bath or Metropolis update and takes only a few hundred lines. If the quarks are included in the calculation, then the Dirac matrix must be inverted. This sparse matrix acts on variables defined at each lattice site and connects only neighboring sites. Its inverse can be efficiently calculated using the conjugate gradient algorithm, again with a few hundred lines of code. In this situation, rewriting QCD code to target special floating point units is a manageable task.

*Stability.* The most effective algorithms for lattice QCD calculations have evolved very slowly. While algorithmic improvements have provided spectacular gains in performance, often outstripping the gains coming from computer technology, these improvements have involved little change to the fundamental structure of the computational method. There is reasonable expectation that a computer architecture that is presently efficient for QCD will also be efficient in a few years.

Over the past 30 years, the computing resources available for lattice QCD have increased by more than six orders of magnitude, from one million floating point additions or multiplications per second (1 Mflops) to more than one trillion (1 Tflops). Combined with substantial advances in algorithms, this has allowed the size of a typical space-time volume to increase from perhaps  $8^3 \times 16$  to  $32^3 \times 64$  or more. With this substantial increase, QCD becomes a “multi-scale” problem in which substantial advantage follows from using increasingly sophisticated methods to treat differently

those parts of the calculation which are physical with a scale of 10–20 lattice spacings and unphysical, cut-off effects associated with distances of 1–4 lattice spacings.

These new methods are not simple and are evolving quickly. Thus, the simplicity and stability of QCD code is less evident now than it was 15 or 20 years ago.

These four features of lattice QCD have made this application an attractive target for the construction of specialized parallel computers. The first of these were the Cosmic Cube at Caltech, the Columbia 16-node machine, and the APE machine in Rome in the 1984–1986 period. The next generation of machine, constructed in the 1987–1993 period, were the 64- and 256-node Columbia machines, the ACPMAPS machine at Fermilab, the GF11 at IBM, and the APE-100 in Rome. Finally, the most recent group of QCD machines, completed in the 1993–2010 time frame, are QCDSF, APEmille, QCDOC, apeNEXT, and QPACE computers. QCDSF, QCDOC, and apeNEXT are treated in separate entries in this volume; the others are described below. These special purpose machines have both made important contributions to the study of QCD and served as examples and inspiration for the development of commercial parallel computers.

## First Examples

The story of QCD machines begins in 1980, a time when three important opportunities converged: (1) the physics potential offered by large-scale lattice QCD calculations, (2) the possibility to exploit large-scale parallelism, and (3) the availability of complex functionality in highly integrated, easily interconnected and inexpensive VLSI (very large scale integration) computer chips. The first machine to exploit these three possibilities was the Cosmic Cube of Fox and Seitz at Caltech [1]. This machine was constructed both to perform lattice QCD calculations and to provide a platform on which parallel implementations of many computational science problems could be mounted. Therefore, a very general hypercube communication network was constructed. The first machine had 64 nodes. The network was constructed to treat these nodes as the 64 corners of a six-dimensional cube with each node connected to its six neighboring nodes in the cube. The individual nodes were built from Intel 8086 and 8087 chips. The 8087 contained floating point hardware with a peak speed of 50 Kflops. The performance for QCD on the 64-node

cosmic cube was  $\sim 2$  Mflops [2], and interesting physics results for the heavy quark potential were published in 1984 [3].

A similar effort to construct special purpose computers for lattice QCD was started by Christ and Terrano in the Physics Department of Columbia University at the same time. Here the focus was on providing high-performance floating point capability while constructing the simplest communications network required to support this floating point capability. Thus, the first Columbia QCD machine was a single floating point accelerator which was attached to a DEC PDP11 computer and used to perform the six triples of  $3 \times 3$  matrix multiplication and accumulation that are required for the Monte Carlo sampling of the Feynman path integral for the theory of gluons alone [4]. This machine used a TRW 16-bit integer multiplier, cascaded four-bit adders, and increased the performance of the PDP11 code by a factor of 20 for a performance twice that of a VAX 11/780.

Following this successful demonstration of the ease with which standard components could be combined to provide fast floating point arithmetic, this same team constructed the Columbia 16-node machine [5, 6]. This machine contained 16 nodes, each on a single-large printed circuit board with wire-wrap connections, arranged as a  $4 \times 4$  mesh. The right and left edges of this mesh were joined as were the top and bottom edges changing the two-dimensional topology from a square to a torus. Each node was composed of Intel 80286 and 80287 chips, providing easily programmable control, address generation, and floating point capability. However, the real power of the node was provided by a vector processor composed of a pair of TRW chips, a floating point adder, and an integer multiplier enhanced by extra circuitry to perform floating point operations. This unit performed long sequences of pipelined, floating point additions and multiplications with the specific operations and operand addresses determined by the 56 bits in a sequence of microcode words stored in a  $4K \times 56$ -bit wide memory. Running at 8 MHz, this unit had a peak speed of 16 Mflops, and the entire machine a peak speed of 256 Mflops.

By modern standards, the nearest-neighbor communication provided in this machine were somewhat primitive. Each node was connected to its four nearest neighbors by 16 bidirectional data wires. If we locate

each node with  $x$  and  $y$  coordinates running between 0 and 3, then these data paths expand the memory of the node with coordinates  $(x, y)$  to include the memories of the two nodes with coordinates  $((x + 1)\%4, y)$  and  $(x, (y + 1)\%4)$ , where  $\%$  indicates the usual mod operation. No address lines were included in these network connections, and off-node communication was carried out in a synchronous SIMD (single instruction multiple data) mode in which all nodes functioned in lock-step and the needed data in the off-node memories was addressed by the local processors (either micro- or vector processor) on those nodes.

The programming model for this machine followed naturally from the homogenous character of both lattice QCD and the computer's architecture. The code was written from the perspective of a single node with an off-node access treated no differently from one which was local. Such single-node code was loaded and run on each node. Those portions of the code that were strictly local could run in MIMD (multiple instruction multiple data) mode without synchronization. Metropolis accept-reject decisions could be easily made in this mode. However, during off-node communications, all code had to be executing the same branch and the Intel 80286 processors had to be synchronous on a cycle-by-cycle level.

Creating code for the 80286/80287 processors was straightforward with good compiler support. This single-node execution model and memory-mapped nearest neighbor communication is relatively easy to understand and follow. Programming the vector unit was more difficult even with a rudimentary assembler. Perhaps the greatest software challenge was created by the heterogenous character of the computer node. Much effort was required to reach an optimal balance between slow but easily created and modified code running on the microprocessors and that executing on the vector processor. Performance improved as bottle necks in the 80286/80287 portions of the code were identified and replaced by vector processor routines.

This machine executed QCD code at a speed 20% faster than its Cray 1 contemporary, sustaining  $\sim 50$  Mflops or 20% efficiency for critical codes. It began physics production in March of 1985 and was used primarily to study the QCD phase transition for a theory of only gluons. Calculations on this machine were able to

reach far enough into the weak coupling regime to verify the dependence of the transition temperature on the coupling strength expected in perturbation theory [7].

The third of these early QCD machines was the APE computer designed and built by an Italian group lead by Parisi and centered at Rome [8]. The first physics results [8] were obtained on a machine of four relatively powerful nodes connected in a one-dimensional ring. This machine later grew into the full APE computer with a larger ring of 16 of these nodes. In contrast to the two machines described above, the APE architecture was entirely designed by the APE collaboration and used no commercial microprocessor. The machine was driven by compiler-generated microcode, which controlled the floating point units (eight Weitek chips on each node), and a sequencer, which determines the next microcode instruction that will be executed. The ring architecture was highly programmable. If nodes are numbered 0 through 15 and the separate memory units, also labeled 0–15, then in a given cycle, processor number  $k$  could be joined to access memory number  $k + l\%16$  for all  $0 \leq k < 16$  with a fixed value of  $l$  determined by program control.

The machine operated in SIMD mode with all nodes controlled by common microcode. Because of the homogeneous character of the machine, a high-level compiler was created to support a Fortran-like “APE language.” The output of the compiler could directly control the computer and could also be written out as symbolic tables which could inform further hand optimization of the input code. A four-node APE computer had a peak speed of 256 Mflops and boasted a 70% efficiency. Two 4-node machines and a later 16-node machine enabled the extensive and influential physics program of the APE collaboration during the late 1980s.

## Second Generation QCD Machines

This next period of QCD machines spans the 1987–1992 time frame and includes significant evolutions in the Caltech and Columbia projects. The full 16-node, 1 Gflops peak ( $10^9$  floating point operations per second) APE machine was brought into operation in 1988 [9]. In addition, new QCD machines were built in Japan, at Fermilab, and by IBM.

The Cosmic Cube described above lead to a sequence of larger hypercube machines at Caltech.

The 32-node Mark IIIfp computer added a Weitek chip set to each node and achieved a 150 Mflops performance on QCD code. While the principal purpose of this machine was to enable parallel computation for a broad range of scientific applications, an interesting state-of-the-art study of the static quark potential was performed on this machine [10]. This series of Caltech machines was instrumental in stimulating Intel to begin producing parallel computers targeting the scientific market.

The Columbia 16-node machine was followed by a 64-node machine which began operation in August of 1987 with a peak speed of 1 Gflops and sustained performance of  $\sim$ 200 Mflops. This machine used the same Intel microprocessor, mesh architecture, and host computer communication as did its 16-node predecessor. The only change was an increased memory size (from 1 Mbyte to 2 Mbytes/node) and a change to 32-bit IEEE Weitek floating point chips. Thus, only the microcode needed to be refreshed. This was followed in 1989 by a third, 256-node machine. Again the same microprocessor was used, the network bandwidth was doubled, and a large enhancement to the floating point unit was realized by using a pair of more advanced Weitek chips each with two external Weitek registers files all running at 16 instead of 8 MHz. This gave an increase in performance of 16 $\times$  over the 64-node machine for a peak performance of 16 Gflops and a sustained QCD performance of 6.5 Gflops for SU(3) conjugate gradient inversion. It should be recognized that this performance was greater or equal to the fastest commercially available machines. For example, a sustained speed of 6.23 Gflops was achieved a year later on a 64K node CMII machine [11].

The new project begun at Fermilab, called ACPMAPS (Advanced Computer Program Multidimensional Processor System) [12], targeted a machine of 256 nodes based on a unified processor architecture constructed from a Weitek chip set. This provided a peak speed of 20 Mflops per node and 5 Gflops for the full machine. The Fermilab approach was crafted to give both cost-effective performance for QCD codes as well as a user-friendly programming environment that would be well suited to the development of new algorithms. The communications network was a hierarchy of 16  $\times$  16 crossbar switches, providing a greater degree

of interconnectivity than the one-dimensional and two-dimensional networks of the APE and Columbia machines. Each of the 32 crates contained eight processor boards interconnected with one crossbar switch. The 32 crates were themselves interconnected as the nodes in a  $2^5$  hypercube. The first physics results from ACPMAPS were reported at the 1991 Lattice Field Theory Symposium in Japan [13, 14]. This machine was upgraded to one of 50 Gflops peak speed in 1992 with a substantial enhancement to the node architecture (the Weitek chip set was replaced by a pair of Intel i860 processors) but with no change in the network [15]. The flexibility of this machine was supported by an innovative software environment which included C-level programming for the individual nodes and a system called Canopy [12] which provided a grid-aware framework that allowed parallel programs to be created with the details of grid communication managed by the software.

During this same time frame, the QCDAIX computer was designed [16] and put into operation [17] at Tsukuba in Japan. This machine contained 480 nodes arranged in a two-dimensional mesh with the edges connected to form a torus. The individual nodes were constructed from a Motorola 68020 processor and a gate-array controlled LSI Logic L64133 floating point chip. The 480 node machine had a peak speed of 14 Gflops and sustained speed of 2.6 Gflops, with 4–5 Gflops for the inversion of the Wilson Dirac operator. First physics results were presented in 1991 [18].

The last of the machines built during this period is the IBM GF11 of Beetham, Deneau, and Weinergarten [19]. This had begun in 1983 and first physics results were presented in 1991 [20]. Like the APE machines, the GF11 was microcode controlled by a standard computer, in this case a RT/PC, and the resulting sequence of microcoded instructions was broadcast to the 566 processors in the system as well as to inputs that controlled a large 576  $\times$  576 switch, which could realize an arbitrary permutation between its 576 data inputs and outputs. During actual program execution, the microcode instructions could choose between 1,024 preloaded switch configurations. The extra 10 ports on the switch were connected to 10 disks, so this switch also provided a very flexible connection between the 566 processors and these 10 disks. Each processor contained a hierarchy of memory (2 MB DRAM, 16K SRAM, and

256 word register file) and a Weitek floating point adder and multiplier, giving each node a peak speed of 20 Mflops. The typical sustained performance reported on a 440-node version of the machine was 6.9 Gflops [21].

While the 8-year time-to-completion suggests the difficult struggle needed to complete the GF11, once online, this machine was very successful. It produced the first calculation, in which a systematic effort was made to remove the errors associated with finite lattice spacing, large quark mass, and finite volume [22]. Later influential work studied the masses and widths of the lightest glueball states (particles formed from gluons rather than quarks) [23].

## Toward One Teraflops

The projects described above succeeded in bringing a few Gigaflops of sustained computational speed to bear on the problems of lattice QCD. During the period 1992–1998, these speeds were dramatically increased by projects in the USA, Italy, and Japan. The APE collaboration developed the APE-100 which enlarged the one-dimensional architecture of the original APE machine to a three-dimensional mesh and incorporated an integrated arithmetic chip which contained the entire original APE node. The first 128-node, 6 Gflops APE100 machines were operational in 1992 and the 2048-node, 100 Gflops version by 1995. This was constructed commercially, and commercial machines, with the name “Quadrics,” were purchased by other groups in Europe to support their own lattice QCD calculations.

This was followed by a still faster APE machine: APEmille. Begun in 1994 [24], this machine replaced the single central controller of the earlier APE machine with an array of controllers with one per eight nodes, all mounted on a single PC board along with memory and communications. APEmille benefited from substantial board-level integration and a 2× faster processor chip. Separate installations of this machine in Germany, France, Italy, and Wales were producing interesting results by 2000.

The group at Columbia began designing a new QCD machine in 1993 [25]. This machine, called QCDS (QCD with Digital Signal Processors), followed a different architecture than the earlier Columbia machines, being built from credit card-size nodes interconnected in a four-dimension network. Machines with 8K and 12K nodes, sustaining 120 and 180 Gflops, were built

at Columbia and the RIKEN BNL Research Center (RBRC). This machine and the follow-on QCDOC computer are described in a related entry.

Finally, in this period, the Tsukuba group, working with the Hitachi company, constructed CP-PACS (Computational Physics Parallel Array Computer System) [26]. Begun in 1992, this machine was complete in 1996 and sustained 200 Gflops [27]. It was constructed as a three-dimension array of Hewlett Packard RISC processors. These were a version of a commercial HP chip modified to include four times the usual 32 floating point registers that were accessed through a sliding window architecture. The network was more powerful than a simple nearest neighbor mesh. While arranged on a three-dimensional Cartesian grid, each node was joined to three crossbar switches which allowed direct communication with any node whose  $(x, y, z)$  coordinates differed in only one variable. This was a fully asynchronous machine with communication performed by message passing. The machine contained 2,048, 300 Mflops nodes for a peak speed of 614 Gflops. A commercial version of this machine was then sold by Hitachi.

## Beyond One Teraflops

Three projects have succeeded in creating QCD machines capable of sustained performance in excess of 1 Tflops. Two of these are described in related entries. The first is the QCDOC machines designed and constructed by a collaboration of Columbia University, the RBRC, and UKQCD. Begun in 1999 and completed in 2005 three QCDOC machines, each with a sustained performance of 4 Tflops were installed at the RBRC, Edinburgh, and the Brookhaven National Laboratory. The second, apeNEXT, was begun in 2000 and led to large machines in Bielefeld, DESY/Zeuten, and Rome with peak speeds of 3.6, 2.4, and 8 Tflops, respectively.

The third machine in this multi-teraflops class is the QPACE machine. This was constructed by a collaboration centered at the University of Regensburg with significant involvement from IBM. Four racks were installed in 2009 at the University of Wuppertal and four more at the Jülich Supercomputing Centre. The aggregate peak speed of the eight racks is 200 Tflops [28] with a sustained performance of 20–30% of peak. The machine is based on an IBM/Sony CELL processor similar to that used in the Playstation 3 but upgraded to perform double-precision arithmetic.

The nodes are interconnected in a three-dimensional mesh network which is connected to each node by a field programmable gate array (FPGA). This machine had a remarkably short design and construction time which was achieved in part by avoiding the design of a special purpose chip (as was done in the QCDOC and apeNEXT projects) and using instead an FPGA in which initial errors could be easily corrected. Physics calculations are expected to begin on this hardware in January 2010.

## Conclusion

The theory of quantum chromodynamics is a computationally simple but highly demanding application area in which the strategy for parallelization is obvious. It is therefore a natural target for parallel computing and, as described above, a natural target for application-specific, highly parallel machines. During the period 1983 through the late nineties QCD machines lead or rivaled the most powerful parallel machines available commercially. For example, the 1 Tflops aggregate peak speed of the QCDSF machines put into operation in 1998 was comparable 3.1 Tflops of the Intel ASCII Red installed at Sandia in 1999.

The later QCD machines were built at costs ranging between \$1 and \$15 million, as might be expected for large research instruments. During the present 2000–2010 period, the 10–20 times larger sums spent on commercial supercomputers and their increasingly effective architectures (inspired in some cases by QCD machines) have lead commercial machines to outstrip by an order of magnitude or more the power available from special purpose machines. Thus, at present (2010), with the possible exception of the QPACE machine, the most challenging QCD problems are run on commercial computers.

Because of the necessary investment of effort and money, machines built for QCD targeted important research goals requiring the largest possible computer capability to achieve. However, beginning in the late 1990s, smaller QCD problems began to be tackled with what were known as Beowulf clusters, collections of 10–100 standard workstations connected by a commercial network [29]. While these machines, often configured in a fashion optimized for lattice QCD, are not able to efficiently manage the small lattice volumes per computer node that are required to mount the most

demanding lattice QCD problems on a very large number of processors, they have become increasingly capable, are highly cost effective, and can be easily upgraded every year. These cluster machines are now enjoying a further increase in performance by incorporating the latest graphics processing units (GPU), which provide enormous performance boosts when a very large local volume can be used on each node. In the USA, the great majority of lattice QCD research is now performed on workstation clusters and the IBM BlueGene computers.

To an even greater extent than in 1980, the deeper study of the physics of quarks, gluons, and new strongly interacting particles likely to be discovered at the Large Hadron Collider at CERN requires vast computational resources not available from current computers. In fact, many important problems require computing performance on the exaflops ( $10^{18}$  flops) scale. Given the experienced and talented research groups in Germany, Great Britain, Italy, Japan, and the USA, we may hope that new technologies will be recognized that can provide this needed computational power and that can be exploited by new generations of specially built QCD machines.

## Related Entries

- [QCD apeNeXT Machines](#)
- [QCD \(Quantum Chromodynamics\) Computations](#)
- [QCDSF and QCDOC Computers](#)

## Bibliographic Notes and Further Reading

The plans, progress, and first results from the QCD machines described here were usually reported in the annual International Symposia on Lattice Field Theory which provide many of the bibliographic citations here. An early volume describing both QCD and other special purpose machines of B. Alder [1] may be of interest. Further detailed information about the first QCD machines can be found in the workshop proceedings of Li, Qiu, and Ren [30], and more recent machines in those of Iwasaki and Ukawa [31]. The theoretical foundations of these lattice methods are described in many monographs and textbooks, for example, those of Creutz [32], Montvay and Muenster [33], Smit [34], Rothe [35], and Degrand and Detar [36].

## Bibliography

1. Alder BJ (ed) (1988) Special purpose computers. Academic, Boston
2. Otto SW (1984) Lattice gauge theories on a hypercube computer. In: Proceedings, gauge theory on a lattice, Argonne, pp 12–20
3. Otto SW, Stack JD (1984) The SU(3) Heavy quark potential with high statistics. *Phys Rev Lett* 52:2328
4. Christ NH, Terrano AE (1984) Hardware matrix multiplier / accumulator for lattice gauge theory calculations. *Nucl Instr Meth A* 222:534–539
5. Christ NH, Terrano AE (1984) A very fast parallel processor, *IEEE T Comput C*-33:344–350
6. Christ NH, Terrano AE (1986) A micro-based supercomputer. *Byte* 11:145–160
7. Christ NH, Terrano AE (1986) The deconfining phase transition in lattice QCD. *Phys Rev Lett* 56:111
8. APE Collaboration, Albanese M et al (1987) The APE computer: an array processor optimized for lattice gauge theory simulations. *Comput Phys Commun* 45:345–353
9. APE Collaboration, Remiddi E (1988) From APE to APE100. In: Proceedings, lattice 88, Batavia, pp 562–565
10. Ding HQ, Baillie CF, Fox GC (1990) Calculation of the heavy quark potential at large separation on a hypercube parallel computer. *Phys Rev D* 41:2912
11. Liu WC. Fast QCD conjugate gradient solver on the connection machine. In: Proceedings, lattice 90, Tallahassee, pp 149–152 (see High Energy Physics Index 29 (1991) No. II041)
12. Fischler M et al (1989) The Fermilab lattice supercomputer project. *Nucl Phys Proc Suppl* 9:571–576
13. Mackenzie PB (1992) Charmonium with improved Wilson fermions. 1. A determination of the strong coupling constant. *Nucl Phys Proc Suppl* 26:369–371
14. El-Khadra AX (1992) Charmonium with improved Wilson fermions. 2. The spectrum. *Nucl Phys Proc Suppl* 26:372–374
15. Fischler M, Gao M, Hockney G, Isely M, Uchima M (1993) Reducing communication inefficiencies for a flexible programming paradigm. *Nucl Phys Proc Suppl* 30:301–304
16. PAX Collaboration, Iwasaki Y, Hoshino T, Shirakawa T, Oyanagi Y, Kawai T (1988) QCD-PAX: a parallel computer for lattice QCD simulation. *Comput Phys Commun* 49:449–455
17. QCDPAX Collaboration, Iwasaki Y et al (1991) QCDPAX: present status and first physical results. *Nucl Phys Proc Suppl* 20:141–144
18. QCDPAX Collaboration, Kanaya K et al (1991) Pure QCD at finite temperature: results from QCDPAX. *Nucl Phys Proc Suppl* 20:300–304
19. Beetem J, Denneau M, Weingarten D (1985) The gfl1 supercomputer. In: The 12th annual international symposium on computer architecture, Boston, pp 108–115
20. Butler F, Chen H, Sexton J, Vaccarino A, Weingarten D (1992) Volume dependence of the valence Wilson fermion mass spectrum. *Nucl Phys Proc Suppl* 26:287–289
21. Weingarten DH (1992) Parallel QCD machines. *Nucl Phys Proc Suppl* 26:126–136
22. Butler F, Chen H, Sexton J, Vaccarino A, Weingarten D (1994) Hadron masses from the valence approximation to lattice QCD. *Nucl Phys B* 430:179–228, hep-lat/9405003
23. Sexton J, Vaccarino A, Weingarten D (1995) Numerical evidence for the observation of a scalar glueball. *Phys Rev Lett* 75: 4563–4566, hep-lat/9510022
24. Bartoloni A et al (1995) The new wave of the APE project: APEmille. *Nucl Phys Proc Suppl* 42:17–20
25. Arsenin I et al (1994) A 0.5-teraflops machine optimized for lattice QCD. *Nucl Phys Proc Suppl* 34:820–822
26. Oyanagi Y (1993) New parallel computer project in Japan dedicated to computational physics. *Nucl Phys Proc Suppl* 30:299–300
27. CP-PACS Collaboration, Iwasaki Y (1998) The CP-PACS project. *Nucl Phys Proc Suppl* 60A:246–254, hep-lat/9709055
28. Baier H et al. QPACE – a QCD parallel computer based on Cell processors. arXiv:0911.2174
29. Gottlieb SA (2001) Comparing clusters and supercomputers for lattice QCD. *Nucl Phys Proc Suppl* 94:833–840, hep-lat/0011071
30. Li X-Y, Qiu Z-M, Ren H-C (eds) (1987) Lattice gauge theory using parallel processors. In: Proceedings, CCAST (world laboratory) symposium/workshop, Peking, 21 May–2 June, 1987. Gordon & Breach, New York, 644 p (proceedings of the CCAST symposium/workshop, I)
31. Iwasaki Y, Ukawa A (eds) Lattice QCD on parallel computers. In: Proceedings, international workshop, Tsukuba, 10–15 Mar 1997. Prepared for international workshop on lattice QCD on parallel computers, Tsukuba, 10–15 Mar 1997
32. Creutz M (1983) Quarks, gluons and lattices. Cambridge University Press Cambridge, 169 p (Cambridge monographs on mathematical physics)
33. Montvay I, Munster G (1994) Quantum fields on a lattice. Cambridge University Press Cambridge, 491 p (Cambridge monographs on mathematical physics)
34. Smit J (2002) Introduction to quantum fields on a lattice: a robust mate. *Camb Lect Notes Phys* 15:1–271
35. Rothe HJ (2005) Lattice gauge theories: an introduction. *World Sci Lect Notes Phys* 74:1–605
36. DeGrand T, Detar CE (2006) Lattice methods for quantum chromodynamics. World Scientific, New Jersey, 345 p

## QCDSP and QCDOC Computers

NORMAN H. CHRIST

Columbia University, New York, NY, USA

### Definition

QCDSP (quantum chromodynamics on digital signal processors) and QCDOC (quantum chromodynamics on a chip) are two computer architectures optimized for large-scale calculations in lattice QCD, a first-principles, numerical formulation of the theory which

describes the interactions of quarks and gluons. Both are mesh machines with a communication network arranged as a four- and six-dimensional torus, respectively. Completed in 1998 the largest QCDSP machine had a peak speed of 0.6 Tflops while three QCDOC machines were completed in 2004/2005, each with a peak speed of 10 Tflops. Both architectures delivered a sustained performance for QCD of about 30% of peak.

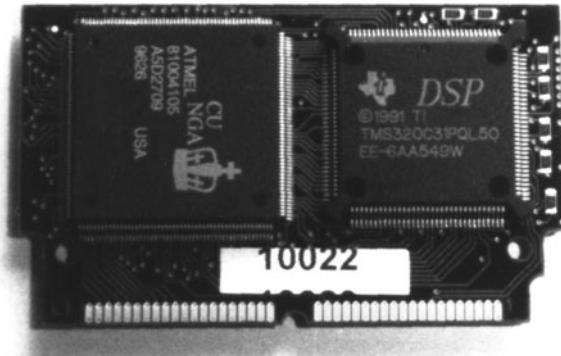
## Discussion

### Overview

The QCDSP and QCDOC machines represent a second generation of massively parallel computers optimized for QCD. Earlier QCD machines constructed at Columbia University had exploited single-board computers to assemble large machines with 64 or 256 nodes, where each node was a large printed circuit board containing a microprocessor and much support electronics including memory, internode communications and substantial added floating point hardware. By the early 1990s, all of these functions could be contained in a very few chips. In particular, if one of those chips was a custom-designed ASIC (applications specific integrated circuit) then even the “glue” or interface logic special to the particular design could be integrated into this small number of chips.

This large increase in integration capability made it possible to shrink one of the large node boards, for example, a  $45 \times 33$  cm board making up one node in the Columbia 256-node machine, to the small, credit-card size,  $6.7 \times 4.5$  cm, QCDSP node shown in Fig. 1. While the size of the node was drastically reduced, the floating point performance remained nearly the same. The 50 Mflops peak speed of this new DSP chip compared well with the 64 Mflops peak speed of the two 16 MHz Weitek chips used on the earlier node. Thus, a large increase in computational power could be achieved if this minaturization was exploited to achieve a substantial increase in the number of nodes. In fact, compact packaging allowed 12,288 QCDSP nodes to be accommodated in a  $5\text{ m}^2$  footprint, very similar to the area occupied by the 256 nodes of the earlier machine. The result was a nearly  $40\times$  increase in performance.

Such a large increase in the number of nodes required a change in the way the nodes were interconnected. If the two-dimensional torus interconnection



**QCDSP and QCDOC Computers.** Fig. 1 A single node of the QCDSP computer. The chip on the right is the Texas Instruments TMS320C31 DSP. That on the left is the ASIC which provides the interface connecting the DSP and memory and manages internode communication over the four-dimensional network interconnecting the nodes. Five memory chips (which are not visible) are mounted on the opposite side of this  $6.7 \times 4.5$  cm DIMM card

were simply expanded from the  $16 \times 16$  mesh of the earlier 256 node machine the result would be  $64 \times 64$  mesh for even a 4K node partition of a 12K node machine. Since the difficulty of the typical lattice QCD calculation increases at least as fast as the seventh power of the linear size of the system under study, the  $40\times$  increase in computer power would only allow a  $40^{1/7} = 1.7$  times increase in problem size, far smaller than the increase in mesh size from 16 to 64. Of course, the linear size of a mesh of 4,096 processors can be easily reduced by increasing the dimension of the mesh. For QCDSP a four-dimensional mesh was used because this nicely matches the four-dimensional space-time mesh of sites used in lattice QCD. With such an interconnect and the onboard wiring adopted for QCDSP, 4,096 nodes can be configured as a  $16 \times 16 \times 4 \times 4$  mesh allowing the full processing power of such a ten times more powerful 4K node partition to be applied to the same size problem as could be mounted on the earlier 256-node machine. The QCDOC network was further expanded to a six-dimensional mesh in order to allow a more flexible mapping of four- or five-dimensional problems onto a fixed partition geometry as discussed below.

## Networks

Typically such massively parallel machines require a variety of networks. In addition to the mesh network which provides high-bandwidth data communication, a combining and broadcast network is needed to perform global sums or to identify the largest or smallest case of a variable appearing across all nodes and then to distribute the result to all nodes. A third network, often distributing interrupt signals, is needed to implement barriers for synchronization among the nodes. This or a similar network is needed to announce errors and bring all nodes to a halt as quickly as possible so that a malfunctioning node can be easily identified. Finally a control network is needed to boot the machine and perform and report the results of power-on diagnostics.

The network most important for high computational performance is that transmitting the data between nodes as the calculation progresses. As discussed above for the QCDS and QCDOC computers, this network is a four- or six-dimensional toroidal mesh. A four-dimensional lattice QCD problem can be easily mapped onto the four-dimensional network of QCDS. Each processor node in an  $N_x \times N_y \times N_z \times N_t$  network can be assigned an  $n_x \times n_y \times n_z \times n_t$  local volume so the resulting space-time volume contained in the entire machine is an  $N_x \cdot n_x \cdot N_y \cdot n_y \times N_z \cdot n_z \times N_t \cdot n_t$  volume assembled by juxtaposing these smaller local volumes in the obvious way. The standard interactions between neighboring sites in this  $N_x \cdot n_x \times N_y \cdot n_y \times N_z \cdot n_z \times N_t \cdot n_t$  mesh of physical space-time points can then be accomplished either within a processing node or by communication between adjacent nodes in the  $N_x \times N_y \times N_z \times N_t$  network of nodes.

While this four-dimension mesh worked very well, the rigid coupling between the problem size and the machine size required frequent reconfiguration of the QCDS partitions as the size of problem changed. This reconfiguration was accomplished by a physical reconnection of the communications cables, a significant impediment that often delayed making needed changes in problem size.

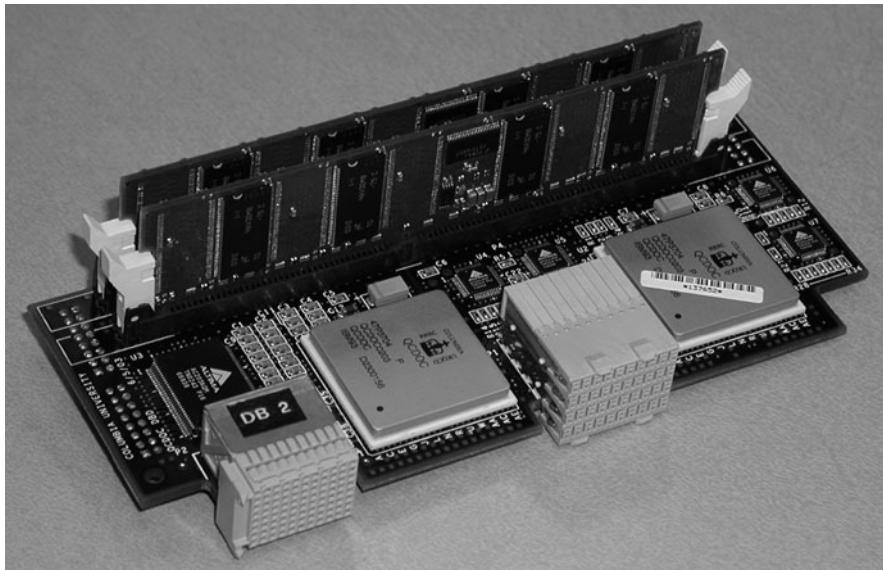
In the QCDOC computer, this coupling between problem size and partition geometry was weakened by increasing the number of dimensions of the network from four to six. For both QCDS and QCDOC each motherboard held 64 nodes. For QCDS this was wired as a  $4 \times 4 \times 2 \times 2$  mesh with the first “4” connected back to

itself to form a one-dimensional torus within the motherboard. The other six faces of this four-dimensional cube were wired to six external connectors. Multi-conductor ribbon cables then joined the corresponding faces between logically adjacent motherboards connecting the other three, off-board dimensions of the four-dimensional torus.

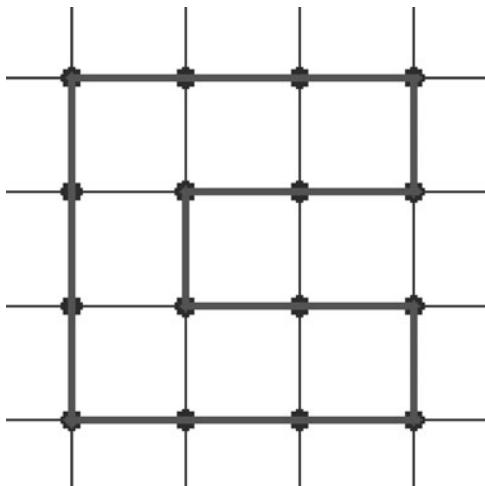
For QCDOC the 64 nodes are interconnected on a motherboard as a  $2 \times 2 \times 2 \times 2 \times 2 \times 2$  mesh. For this machine a daughter card, shown in Fig. 2 contains two nodes to accommodate the long dual inline memory modules (DIMM) provided for each node. The two nodes on a daughter card are connected in a two-dimensional torus and the other network connections made to the motherboard. The motherboard is wired to join these 32 daughter cards into a  $2^5$  cube. Two additional dimensions of this  $2^5$  cube are connected as two-node torii and connections to the remaining six faces of the  $2^6$  cube of nodes are carried to 6 connectors leaving the motherboard. Thus, as in QCDS, external cables join the motherboards to assemble a three-dimensional torus of motherboards.

Since it is easy to map a lower dimensional torus within one of higher dimension, the six-dimensional network of QCDOC can be used to allow a single six-dimensional QCDOC partition to support a greater variety of four-dimensional mesh geometries than can be accommodated in a single four-dimensional QCDS partition. This is illustrated in Fig. 3 which shows the example of a two-dimensional network in which a subset of the links are used to realize a one-dimensional mesh. Used in this way, the three-dimensional  $2 \times 2 \times 2$  QCDOC network that is wired on the motherboard can be combined with the more distributed  $2N_x \times 2N_y \times 2N_z$  network connecting  $N_x N_y N_z$  motherboards in a variety of ways. For example, these extra eight processors can be wired into an added one-dimensional torus in the time dimension of size 8 giving a  $2N_x \times 2N_y \times 2N_z \times 8$  torus. Alternatively only two of these dimension could be used to form a one-dimensional torus of size 4 and the third dimension could be combined with the  $x$ -direction to double its length giving a  $4N_x \times 2N_y \times 2N_z \times 4$  torus.

Neither QCDS and QCDOC has a separate/combining broadcast network. Instead hardware features are added to the mesh communications network which identify particular paths though the mesh network that can be used to combine or broadcast



**QCDSP and QCDOC Computers. Fig. 2** A single daughter board of the QCDOC computer. This card holds two QCDOC nodes (the two silver colored chips), an Ethernet five-port hub (the rectangular Alitma chip visible on the left), four Ethernet “PHY” chips (identifiable as the four small square Alitma chips) and two DIMM sticks which varied in size between 128 and 512 MB



**QCDSP and QCDOC Computers. Fig. 3** An example of mapping a one-dimensional 16-site torus into a two-dimensional  $4 \times 4$  torus

data. For example, additional hardware was provided that allows data passed from one or more neighboring nodes into a given node to be added or compared

to local data and then the sum or maximum or minimum to be passed on with minimal latency to a specially designated neighbor. The QCDSP design had a separate interrupt network which allows the implementation of barriers and the transmission of interrupts. For QCDOC this network is realized in two ways. First a simple network of three interrupt lines was connected throughout the machine which allows a signal sent by one processor to be seen by all others which is used during boot-up and to signal fatal errors. A larger and more flexible system of interrupts is provided by the distribution of 8-bit interrupt packets over the six-dimensional mesh network.

The networks used for booting, diagnostics, and file transfer for QCDSP and QCDOC were quite different. For QCDSP a tree whose individual links used the commercial SCSI bus conventions connected the host workstation, typically a SUN, to each of the motherboards. For QCDOC this host communication and control function is provided by standard Ethernet connections. In fact each node has two Ethernet ports. The first is connected directly to the JTAG control of the processor allowing JTAG commands embedded in Ethernet packets to load boot code and boot the QCDOC Power PC 440 processor. The second port is configured

as a standard Ethernet device, supporting conventional remote procedure call (RPC) communication with the host and a standard NFS client which runs on each node and provides access to networked disk servers. This Ethernet hardware is likely the greatest source of difficulty with the QCDOC design with frequent hang-ups requiring elaborate software work-arounds.

### Mechanical Design

As shown in Fig. 1, the individual QCDSP nodes were mounted on small printed circuit boards with the form factor of a standard DIMM card. This design provided very reliable. Sixty three of these daughter cards were socketed on a QCDSP motherboard. The 64th node was soldered directly to the mother board and supplied with added connections giving it more memory, access to two SCSII interface chips and an EEPROM. This modular design allowed easy repair of faulty daughter cards and was continued for QCDOC. In this case two nodes are mounted on a daughter card as shown in Fig. 2. A QCDOC motherboard is shown in Fig. 4.

For both machines eight motherboards are mounted in a backplane that makes up a crate as shown for QCDOC in Fig. 5. Two crates are stacked to form a rack. The racks are water cooled with a heat exchanger installed below each crate followed by a tray of muf-fin fans pushing the air upward. The return path for

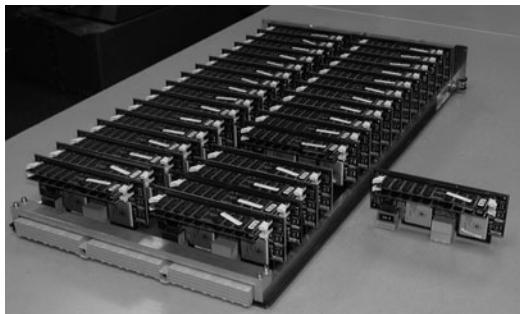
the air is provided inside the cabinet holding the two crates. The racks can then be lined up in a row with cables running from one rack to the next. If desired, these racks can also be stacked two high with a final “top hat” added which contains the Ethernet switches. Figure 6 shows the largest QCDOC installation at the Brookhaven National Laboratory.

Both machines are remarkably low power with a single QCDSP node consuming about 3 W and a QCDOC node slightly more. Thus, QCDSP achieved about 5 Gflops/KW while QCDOC achieves 60 Gflops/KW. Both machines were also highly economical at the time they were finished. The price per sustained performance for QCDSP was \$13/Mflops while for QCDOC this number falls to a little more than \$1/Mflops.

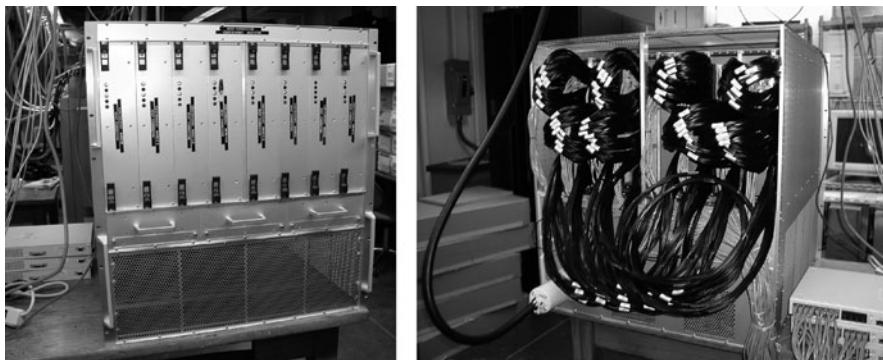
### Software

The software for these two machines is composed of application code which performs the lattice QCD calculation and the operating system (OS) which boots the machine, responds to error conditions, loads and passes arguments to the application code, and provides network and file system support. On both machines the custom OS is divided between the nodes of the parallel machine and the host computer. At boot-up the OS on the host is active, loading the operating system on each node, carrying out extensive diagnostic tests of the parallel machine and loading the user code. Control is then passed to the OS on each node which starts the application code and provides the services requested by that code. The OS running on each node is designed to provide a “C” and “C++” environment for the application code with much of the standard UNIX functionality supported. To the extent that resources beyond those available on the node are required, for example data from a remote disk file or the ability to print to the user’s screen, the node component of the OS requests these services from the OS running on the host computer. Now the host component of the OS acts as the slave to the OS running on each node.

The simple and lean structure of the node OS is critical to the high performance of these machines. No multitasking or asynchronous behavior interferes with the application code on any of the nodes. The node OS ceases to execute when the application code starts running.



**QCDSP and QCDOC Computers. Fig. 4** A QCDOC motherboard holding 31 daughter cards with the 32nd daughter card removed. The three large edge connectors are wired directly through the backplane to three groups of eight cable connectors. Each cable is a bundle of 20 twisted, differential pairs, 16 of which provides uni-directional communications for the QCDOC mesh network



**QCDSP and QCDOC Computers. Fig. 5** Front and rear views of a QCDOC crate. The bundles of black cables carry the mesh communications while the gray cables leaving the lower right provide the 64 Ethernet connections to the eight motherboards



**QCDSP and QCDOC Computers. Fig. 6** The QCDOC installation at the Brookhaven National Laboratory (BNL). The RIKEN-funded RBRC QCDOC makes up the 12 racks on the right while the DOE-funded USQCD QCDOC computer appears on the left. Each machine contains 12,288 nodes, has a peak speed of 10 Tflops and a sustained speed in excess of three Tflops for lattice QCD

Q

Both machines were typically used in a mode where the same code is executing on each node. Since there was no requirement for precise synchronization, the programs on different nodes can execute different branches. However, if one node runs ahead of the rest and begins communications with its slower neighbors, that faster node will wait until needed data is sent by its neighbors or the data it has sent to its neighbors has been removed from the neighbor's input buffer and this action acknowledged so that further data can be sent. This self-synchronization provided by the communications hardware is automatic and requires no action from the user program.

The QCDSP DSP chip came with "C" and "C++" compilers although the "C++" compiler quickly became

out-of-date. Given the lack of a data cache, standard "C" code gave poor performance (typically a few percent of peak) so that frequently used routines had to ultimately be written in assembler. The PowerPC core in QCDOC is much more efficient with its 32 KB instruction and data caches. With reasonable precautions "C" code might achieve 10 or even 20% of peak. The most critical code was best written in assembler, a process greatly aided by the Boyle assembler generation library (BAGEL) which was used to generate much of the assembler used in the QCD code.

Such single node code can be run directly on QCDSP or QCDOC machines with thousands of nodes if this many-node environment is kept in mind when that code is written. For example, if data is sent to a

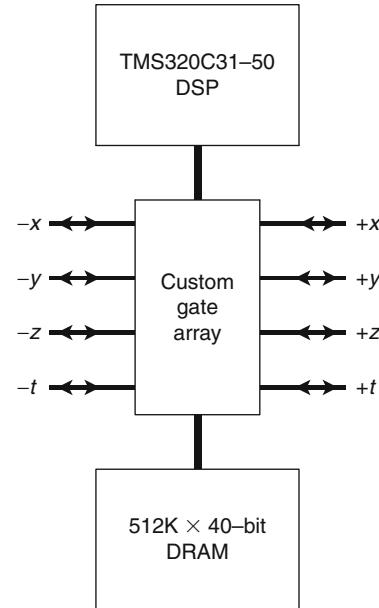
neighboring node in the  $+\mu$  direction, then the corresponding action of the neighbor in the  $-\mu$  direction must be anticipated and a matching  $-\mu$  direction receive instruction invoked as well. If a sum across all lattice sites is to be performed, this must be done by a call to a global sum routine which returns the sum performed across all nodes in the machine. The sends and receives of data in a particular direction are initiated by “C” function calls which begin the transmissions and test to see if they are complete.

On QCDOC in addition to these special function calls, internode communications can also be accomplished by using the quantum message passing (QMP) library. This software was produced with SciDAC support by the USQCD collaboration. QMP provides communication routines which overlap in functionality with the standard message passing interface (MPI). Code which incorporates QMP calls can be run on many platforms including workstation clusters and other massively parallel supercomputers allowing portable code to be run on QCDOC. This QMP support was highly successful and resulted in most of the code used by groups in the USA studying lattice QCD being run on the QCDOC machines at BNL.

## QCDS Architecture

A mesh architecture permits essentially all of the computing hardware of the parallel computer (with the exception of the network wires and perhaps some transceivers) to be built into the computing node. This was realized in the QCDS machines. Essentially all of the computer’s functions were carried out by the three basic components on the daughter card shown in Fig. 1. Thus, the remaining parts of the computer, the motherboard and backplane, were to a large degree passive, providing a clock signal and power to the daughter cards and routing the communication wires.

This concentration of functionality on the QCDS daughter board was achieved with the custom-designed ASIC (referred to here as the “node gate array” or NGA) that was the third component, beyond the DSP and memory, which was mounted on the daughter board. Figure 7 shows the organization of the daughter card. The NGA provided the interface between the DSP and



**QCDS and QCDOC Computers. Fig. 7** A block diagram showing a QCDS node. The central block is the node gate array (NGA) which provided the interface between the upper DSP chip and the lower DRAM chips. It also provided serial communications to each of the eight nearest neighbors connected to this node by the four-dimensional QCDS mesh network

the dynamic random access memory (DRAM), supplying the periodic refresh signal and error checking and correction (ECC) required by this dense, economical memory technology.

The NGA also contained a 32-location circular buffer which prefetched and stored DRAM data according to commands from the DSP. Thus, when the DSP began to multiply an 18-element link matrix by a 12 component fermion spinor, the circular buffer could be instructed by the DSP to load the next 32-bit data word from memory followed by the additional 17 sequential locations corresponding to the link matrix. These data were held in the circular buffer, available for later, low-latency DSP access.

The NGA also sent and received data to and from the eight neighbors of the node. It managed the low-level protocol needed to serialize and transmit the data to be sent, or receive and deserialize incoming data. The NGA would also send an acknowledgment packet after 32 bits

of data had been received and the receive buffer had been cleared so that a further 32-bits could be received. Received data with bad parity was acknowledged with an error packet and would be retransmitted.

The data being sent or received was fetched or stored directly from/to memory by an independent, direct memory access (DMA) controller, one for each of the eight directions. The DSP needed only load the starting address for such a transfer and a start command in order to begin the transfer. Additional preloaded control bits determined how much data was to be transferred and if the data should be fetched from or stored to memory as a series of blocks of fixed length separated by a fixed stride. Thus, the DSP could quickly start such a data transfer and then resume calculation using data not involved in the transfer. All eight directions could be active simultaneously at 50 MHz for a total off-mode transfer rate of 50 MB/s.

## QCDOC Architecture

The QCDOC architecture is a natural evolution of that described above for QCDSP, exploiting the much higher degree of integration that was available in 1999 when the QCDOC design was begun. Here the PowerPC processor is an IBM ASIC library component and was combined with the serial communications unit (SCU) into the single chip that makes up nearly all of a QCDOC node. The entire 2 MB memory of the QCDSP chip was doubled and included in the QCDOC chip as embedded DRAM (EDRAM), again exploiting IBM technology. In addition two Ethernet controllers, a synchronous DRAM controller, and an 0.8 Gflops double precision attached floating point unit were incorporated in this QCDOC chip as shown in Fig. 8.

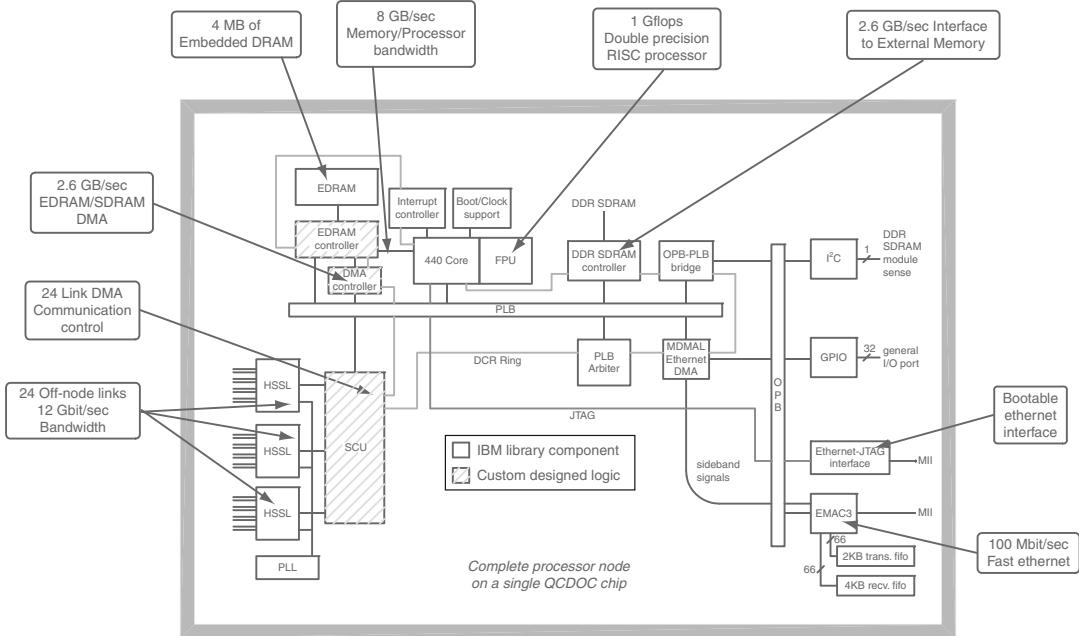
These improvements give the QCDOC machine more than ten times the capability of QCDSP. The processor has increased from 50 to 400 MHz and the peak performance from 50 to 800 Mflops. The memory bandwidth has grown from 25 MB/s to 2.1 GB/s while the total off-node bandwidth increased from 0.025 to 0.6 GB/s for send and for receive. Finally the limited DSP with its non-IEEE compliant single precision has been replaced by the IEEE, 64-bit precision of the IBM FPU, making QCDOC a highly capable and convenient-to-program computer.

## Conclusion

Both the QCDSP and QCDOC machines have been used for a wide range of influential physics calculations. The large QCDSP machines at Columbia and the RBRC began operation in 1998 and continued until January 31, 2006. A smaller 2-crate machine was also installed at the Supercomputer Computations Research Institute of Florida State University. The first 12,288-node QCDOC machine began operation in December of 2004 at the University of Edinburgh which was followed in a few months by the RBRC and DOE 12,288-node machines at BNL. The two large machines at BNL are still operating at full capacity at the time of this writing (May 2011).

The QCDSP machines were used to develop and exploit the chiral, domain wall fermion (DWF) formalism [1, 2]. This approach to lattice calculations with spin-1/2 particles preserves the independent symmetry of the left- and right-handed quarks in the limit of vanishing quark mass by adding a fifth dimension to the usual four-dimensional space-time lattice. The substantial increase in capability provided by QCDSP allowed the study of these new five-dimensional lattices with lattice extent varying between 16 and 128 in this fifth dimension. Among the more notable results achieved with these machines were a thorough quenched study of low-energy QCD using DWF [3], the application of this new method to the weak decays of the K meson including results for the complex, CP violating amplitudes  $A_0$  and  $A_2$  [4], the successful application of RI/MOM techniques to non-perturbatively renormalize the seven four-Fermi operators which appear in this calculation [4, 5], and the development of methods to include the effects of DWF quark loops in lattice QCD calculations at both zero [6] and nonzero [7] temperatures.

This inclusion of quark loops, a step necessary for consistency, was at the limit of the capabilities of QCDSP. However, this preliminary work set the stage for much larger scale calculations enabled by the QCDOC machines. Thus, the RIKEN, BNL, Columbia (RBC) and UKQCD collaborations, which had worked together on the design and construction of QCDOC began a joint program of large-scale DWF calculations using the completed QCDOC computers to study a wide range of questions important in particle and nuclear physics on  $16^3 \times 32$  and  $24^3 \times 64$  grids. Among



**QCDSP and QCDOC Computers. Fig. 8** A block diagram of the ASIC which makes up each QCDOC node. The cross-hatched blocks are those designed by the team which built the computer while the open boxes represent standard IBM ASIC library components

the many results obtained one might note those for the neutral K meson mixing parameter  $B_K$  [8], the nucleon axial coupling  $g_A$  [9] and the decay amplitude  $f_+$  of a K meson into a pion and lepton pair [10], to mention a few examples. In addition a wide range of topics have been studied by members of the USQCD collaboration using the DOE-funded QCDOC.

The QCDSP and QCDOC machines have also exerted influence on high-performance computing technology. The award of the Gordon Bell prize for cost performance at SC98 to the BNL QCDSP machine helped attract attention to this economical, low power, and yet broadly applicable architecture. This visibility together with the recruiting by IBM of four physicists closely involved with QCDSP and earlier Columbia University machines, lead to a collaboration between IBM, Columbia University, Edinburgh University, and the RIKEN laboratory in Japan on the QCDOC computer and at the same time, the start of the separate Blue Gene Light project at IBM. Blue Gene Light evolved into the present IBM Blue Gene series of commercial computers which have transformed high-performance computing and dominated the top 500 list.

## Related Entries

- [QCD apeNeXT Machines](#)
- [QCD \(Quantum Chromodynamics\) Computations](#)
- [QCD Machines](#)

## Bibliographic Notes and Further Reading

A thorough account of these two computer architectures can be found in Ref. [11], written after both machines were complete. Addition information about QCDSP appears in Ref. [12]. Earlier accounts of the design and construction these computers appear in the proceedings of the annual International Symposia on Lattice Field Theory and other HPC conferences. For QCDSP see Refs. [13–16] and for QCDOC Refs. [17–22].

## Bibliography

1. Kaplan DB (1992) A method for simulating chiral fermions on the lattice. *Phys Lett* B288:342–347 [[hep-lat/9206013](#)]
2. Shamir Y (1993) Chiral fermions from lattice boundaries. *Nucl Phys* B406:90–106 [[hep-lat/9303005](#)]
3. Blum T et al (2004) Quenched lattice qcd with domain wall fermions and the chiral limit. *Phys Rev* D69:074502 [[hep-lat/0007038](#)]

4. RBC Collaboration, Blum T et al (2003) Kaon matrix elements and cp-violation from quenched lattice qcd. i: the 3-flavor case. *Phys Rev D*68:114506 [hep-lat/0110075]
5. Blum T et al (2002) Non-perturbative renormalisation of domain wall fermions: quark bilinears. *Phys Rev D*66:014504 [hep-lat/0102005]
6. Aoki Y et al (2005) Lattice QCD with two dynamical flavors of domain wall fermions. *Phys Rev D*72:114505 [hep-lat/0411006]
7. Chen P et al (2001) The finite temperature qcd phase transition with domain wall fermions. *Phys Rev D*64:014503, [arXiv:hep-lat/0006010]. 13
8. RBC Collaboration, Antonio DJ et al (2008) Neutral kaon mixing from 2 + 1 flavor domain wall QCD. *Phys Rev Lett* 100:032001 [hep-ph/0702042]
9. RBC+UKQCD Collaboration, Yamazaki T et al (2008) Nucleon axial charge in 2 + 1 flavor dynamical lattice QCD with domain wall fermions. *Phys Rev Lett* 100:171602 [0801.4016]
10. Boyle PA et al (2008) KI3 semileptonic form factor from 2 + 1 flavour lattice QCD. *Phys Rev Lett* 100:141601 [0710.5136]
11. Boyle P et al (2005) Overview of the qcdsp and qcdoc computers. *IBM Res J* 49:351–360
12. Mawhinney RD (1999) The 1 teraflops qcdsp computer. *Parallel Comput* 25:1281 [hep-lat/0001033]
13. Arsenin I et al (1994) A 0.5-teraflops machine optimized for lattice QCD. *Nucl Phys Proc Suppl* 34:820–822
14. Chen D et al (1998) The qcdsp project: a status report. *Nucl Phys Proc Suppl* 60A:241–245
15. Chen D et al (1999) Status of the qcdsp project. *Nucl Phys Proc Suppl* 73:898 [hep-lat/9810004]
16. Christ NH (2000) Computers for lattice qcd. *Nucl Phys Proc Suppl* 83:111–115 [hep-lat/9912009]
17. Chen D et al (2001) QCDOC: a 10-teraflops scale computer for lattice QCD. *Nucl Phys Proc Suppl* 94:825–832 [hep-lat/0011004]
18. Boyle PA et al (2002) Status of the QCDOC project. *Nucl Phys Proc Suppl* 106:177–183 [hep-lat/0110124]
19. Boyle PA et al (2003) Status of and performance estimates for QCDOC. *Nucl Phys Proc Suppl* 119:1041–1043 [hep-lat/0210034]
20. QCDOC Collaboration, Boyle PA, Jung C, Wettig T (2003) The qcdoc supercomputer: hardware, software, and performance. *ECONF* C030324I:THIT003 [hep-lat/0306023]
21. Boyle PA et al (2004) Hardware and software status of QCDOC. *Nucl Phys Proc Suppl* 129:838–843 [hep-lat/0309096]
22. Boyle PA et al (2005) Qcdoc: project status and first results. *J Phys Conf Ser* 16:129–139

## Quadrics

SALVADOR COLL

Universidad Politécnica de Valencia, Valencia, Spain

## Synonyms

[QsNet](#)

## Definition

Quadrics was a company that produced hardware and software for high-performance interconnection networks used in cluster-based supercomputers. Their main product was QsNet, a high-speed interconnection network that was referred to as Quadrics interconnection network. Quadrics, and QsNet, played a significant role in the supercomputing field from 1996, when the company was created, until 2009, when it was officially closed.

## Discussion

### Introduction

Quadrics was created in 1996 as a subsidiary of Finmeccanica, a large Italian industrial group, under the name Quadrics Supercomputers World (QSW). The new company inherited the architecture of the Quadrics parallel computer and the Meiko CS-2 supercomputer [2]. The Quadrics computer was originally developed at the Istituto Nazionale di Fisica Nucleare (INFN) and later commercialized by Alenia Spazio (another Finmeccanica subsidiary). The Meiko CS-2 was a massively parallel supercomputer based on Sun or Fujitsu processors connected by a multistage fat-tree network. The network was implemented by two hardware building blocks, a programmable network interface, called Elan and an 8-way crossbar switch called Elite. That technology was later developed into QsNet by QSW, whose name was shortened to be simply Quadrics on 2002. QsNet proved very successful reaching its peak on June 2003 when six out of the ten fastest supercomputers in the world used the Quadrics interconnect. During 2004, the second generation of the Quadrics network, QsNetII, was deployed. Finally, very close to the completion of the third-generation Quadrics network, QsNetIII, the company was closed on June 2009 and the support was transferred to Vega UK Ltd.

## QsNet

► [Quadrics](#)

## The Quadratics Interconnection Network

The Quadratics network surpassed contemporary high-speed interconnects – such as Gigabit Ethernet [23], GigaNet [26], the Scalable Coherent Interface (SCI) [13], the Gigabyte System Network (HiPPI-6400) [25], and Myrinet [4] – in functionality with an approach that integrated a node’s local virtual memory into a globally shared, virtual-memory space; provided a programmable processor in the network interface that allowed the implementation of intelligent communication protocols; delivered integrated network fault detection and fault tolerance; and supported collective communication operations at hardware level.

The Quadratics interconnection network is a bidirectional multistage interconnection network with a connection pattern among stages known as butterfly. It is based on  $4 \times 4$  switches, and can be viewed as a quaternary fat tree [14]. QsNet is based on two custom ASICs: a communication coprocessor called Elan, integrated into a network interface card, and a high-bandwidth, low-latency, communication switch called Elite. It uses wormhole switching with two virtual channels per physical link, source-based routing and adaptive routing. The main specifications of the two commercialized QsNet generations are summarized in Table 1 for the network interface card, and in Table 2 for the network switches.

QsNet connects Elite switches in a quaternary fat-tree topology, which belongs to the more general class of  $k$ -ary  $n$ -trees [18, 19]. A quaternary fat-tree of dimension  $n$  is composed of  $4^n$  processing nodes and  $n \times 4^{n-1}$  switches interconnected as a butterfly network (see below); it can be recursively built by connecting four quaternary fat trees of dimension  $n - 1$ .

### Fat-Tree ( $k$ -ary $n$ -tree)

A  $k$ -ary  $n$ -tree is composed of two types of vertices:  $N = k^n$  processing nodes and  $nk^{n-1} k \times k$  communication switches (a  $k$ -ary  $n$ -tree of dimension  $n = 0$  is composed of a single processing node). Each node is an  $n$ -tuple  $\{0, 1, \dots, k - 1\}^n$ , while each switch is defined as an ordered pair  $(w, l)$ , where  $w \in \{0, 1, \dots, k - 1\}^{n-1}$  and  $l \in \{0, 1, \dots, n - 1\}$ .

- Two switches  $(w_0, w_1, \dots, w_{n-2}, l)$  and  $(w'_0, w'_1, \dots, w'_{n-2}, l')$  are connected by an edge if and only if  $l' = l + 1$  and  $w_i = w'_i$  for all  $i \neq l$ . The edge is labeled

**Quadratics. Table 1** Quadratics NIC specifications

| QsNet                              | QsNetII                             |
|------------------------------------|-------------------------------------|
| 32-bit RISC thread processor       | 64-bit RISC thread processor        |
| 32-bit virtual addressing          | 64-bit virtual addressing           |
| 16Kbytes data cache                | 32Kbytes data cache                 |
| 64 Mbytes ECC SDRAM                | 64 Mbytes DDR SDRAM                 |
| PCI 64 bit/66 MHz                  | PCI-X 64 bit/133 MHz                |
| $2 \times 400$ MHz byte-wide links | $2 \times 1.33$ GHz byte-wide links |

**Quadratics. Table 2** Quadratics switch specifications

| QsNet                               | QsNetII                             |
|-------------------------------------|-------------------------------------|
| 8 links $\times$ 2 virtual channels | 8 links $\times$ 2 virtual channels |
| 35 ns unblocked latency             | 20 ns unblocked latency             |

with  $w'_l$  on the level  $l$  vertex and with  $w_l$  on the level  $l'$  vertex.

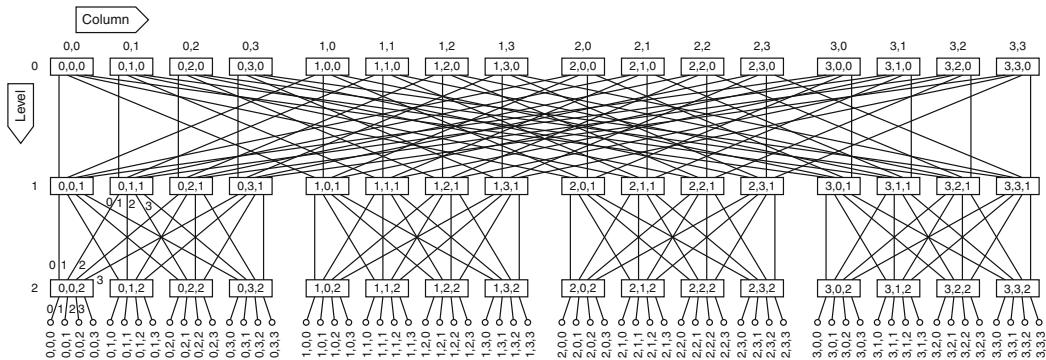
- There is an edge between the switch  $(w_0, w_1, \dots, w_{n-2}, n - 1)$  and the processing node  $p_0, p_1, \dots, p_{n-1}$  if and only if  $w_i = p_i$  for all  $i \in \{0, 1, \dots, n - 2\}$ . This edge is labeled with  $p_{n-1}$  on the level  $n - 1$  switch.

The labeling scheme shown makes the  $k$ -ary  $n$ -tree a delta network [8, 9]: any path starting from a level 0 switch and leading to a given node  $p_0, p_1, \dots, p_{n-1}$  traverses the same sequence of edge labels  $(p_0, p_1, \dots, p_{n-1})$ . An example of such labeling is shown in Fig. 1, for a 64-node QsNet network, that is a 4-ary 3-tree. The above-defined connection pattern among levels is also referred to as butterfly.

### First Generation, QsNet

The Elan3 network interface connects the high-performance, multistage Quadratics network to a processing node containing one or more CPUs. In addition to generating and accepting packets to and from the network, the Elan provides substantial local processing power to implement high-level, message-passing protocols such as MPI. The internal functional structure of the Elan3, shown in Fig. 2, centers around two primary processing engines: the microcode processor and thread processor.

The 32-bit microcode processor supports four threads of execution, where each thread can independently issue pipelined memory requests to the memory system. Up to eight requests can be outstanding at



**Quadrics. Fig. 1** 4-ary 3-tree node, switch, and edge labels

any given time. The scheduling for the code processor enables a thread to wake up, schedule a new memory access on the result of a previous memory access, and go back to sleep in as few as two system-clock cycles.

The four code threads are for

- The inputter, which handles input transactions from the network
- The DMA engine, which generates DMA packets to write to the network, prioritizes outstanding DMAs, and time slices large DMAs to prevent adverse blocking of small DMAs
- Processor-scheduling, which prioritizes and controls the thread processor's scheduling and descheduling
- The command processing, which handles requested operations (commands) from the host processor at the user level

The thread processor is a 32-bit RISC processor that helps implement higher-level messaging libraries without explicit intervention from the main CPU. To better support the implementation of high-level, message-passing libraries without the main CPU's explicit intervention, QsNet augments the instruction set with extra instructions. These extra instructions help construct network packets, manipulate events, efficiently schedule threads, and block save and restore a thread's state when scheduling.

The memory management unit (MMU) translates 32-bit virtual addresses into either 28-bit local SDRAM physical addresses or 48-bit peripheral component interconnect (PCI) physical addresses. To translate these addresses, the MMU contains a 16-entry, fully

associative, translation look-aside buffer (TLB) and a small data-path and state machine used to perform table walks to fill the TLB and save trap information when the MMU experiences a fault.

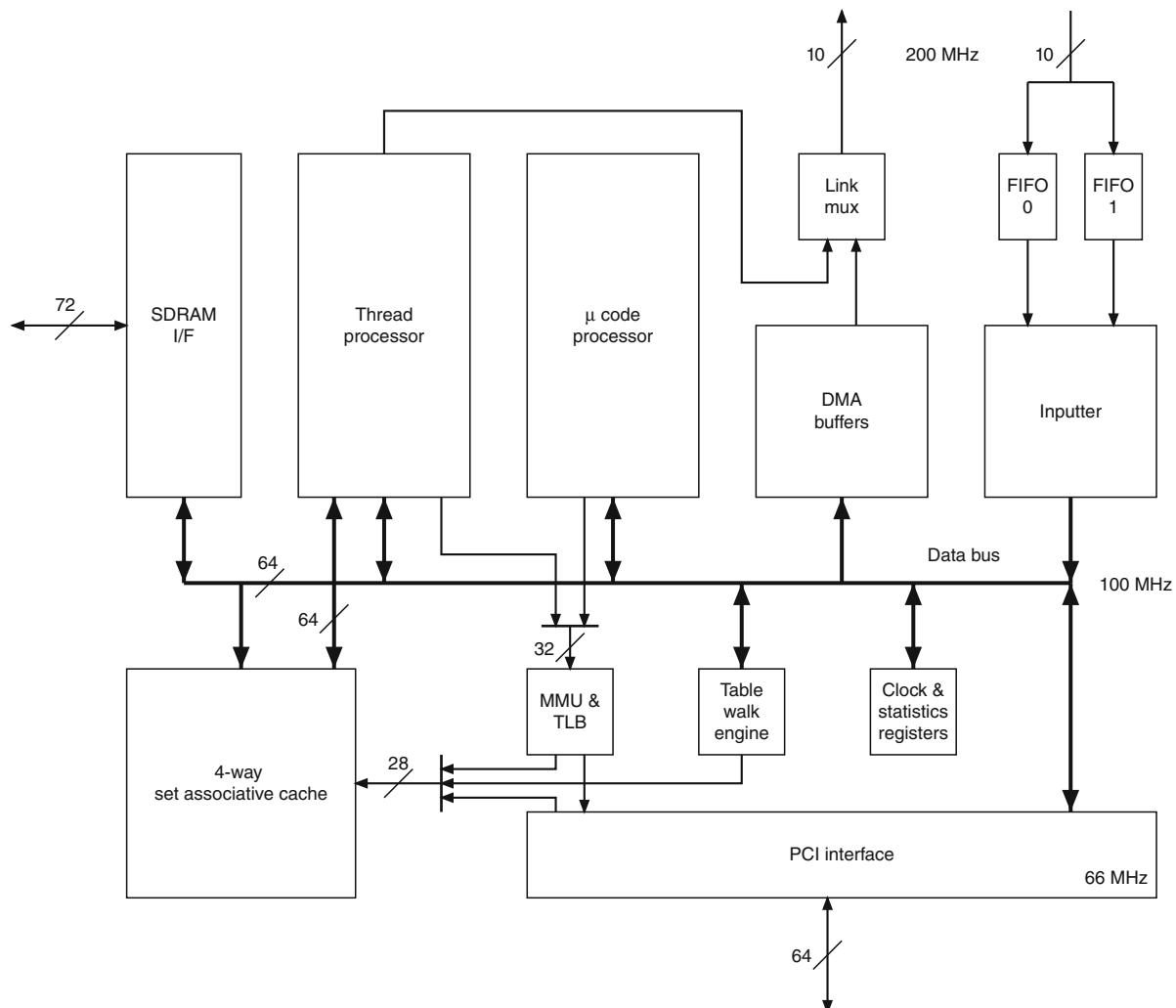
The Elan contains routing tables that translate every virtual processor number into a sequence of tags that determine the network route.

Elan has an 8-Kbyte memory cache (organized as 4 sets of 2Kbytes) and 64-Mbyte SDRAM memory. The cache line size is 32 bytes. The cache performs pipelined fills from SDRAM and can issue multiple cache fills and write backs for different units while still being servicing accesses for units that hit on the cache. The SDRAM interface is 64 bits in length with eight check bits added to provide error-correcting code. The memory interface also contains 32-byte write buffer and read buffers.

The link logic transmits and receives data from the network and generates 9 bits and a clock signal on each half of the clock cycle. Each link provides buffer space for two virtual channels with a 128-entry, 16-bit FIFO RAM for flow control.

The Elite provides the following features:

- Eight bidirectional links supporting two virtual channels in each direction
- An internal  $16 \times 8$  full crossbar switch (the crossbar has two input ports for each input link to accommodate two virtual channels)
- Packet error detection and recovery with cyclic-redundancy check-protected routing and data transactions
- Two priority levels combined with an aging mechanism to ensure fair delivery of packets in the same priority level



**Quadrics. Fig. 2** Elan3 functional units

- Hardware support for collective communication
- Adaptive routing

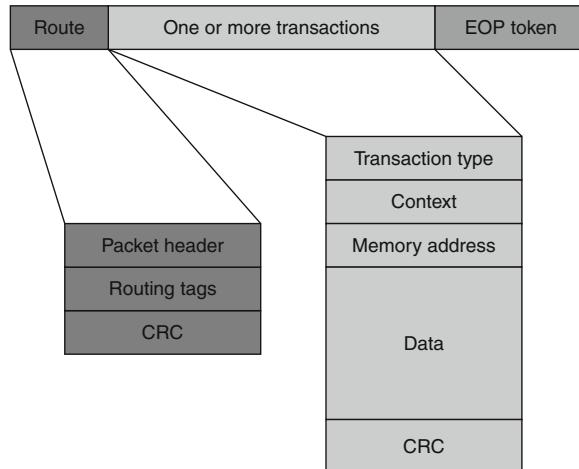
can be connected to two or more networks through multiple network interface cards.

### Multiple Network Rails

One novel solution exploited by Quadrics-based systems to the issues of limited bandwidth availability in network connections, and of fault tolerance, is the use of multiple independent networks (also known as rails) [6]. Support for multirail networks has been included for all the QsNet products. In a multirail system, the network is replicated. In this way, each node

### Packet Routing and Flow Control

Each user- and system-level message is chunked in a sequence of packets by the Elan. An Elan packet contains three main components. The packet starts with the (1) routing information, which determines how the packet will reach the destination. This information is followed by (2) one or more transactions consisting of some header information, a remote memory address, the context identifier and a chunk of data, which can



**Quadrics. Fig. 3** Packet transaction format

be up to 64 bytes in the current implementation. The packet is terminated by (3) an end of packet (EOP) token, as shown in [Fig. 3](#).

Transactions fall into two categories: write block transactions and non-write block transactions. The purpose of a write block transaction is to write a block of data from the source node to the destination node, using the destination address contained in the transaction immediately before the data. A DMA operation is implemented as a sequence of write block transactions, partitioned into one or more packets (a packet normally contains five write block transactions of 64 bytes each, for a total of 320 bytes of data payload per packet). The non-write block transactions implement a family of relatively low-level communication and synchronization primitives. For example, non-write block transactions can atomically perform remote test-and-write or fetch-and-add and return the result of the remote operation to the source, and can be used as building blocks for more sophisticated distributed algorithms.

Elite networks are source routed. The Elan network interface, which resides in the network node, attaches route information to the packet header before injecting the packet into the network. The route information is a sequence of Elite link tags. As the packet moves inside the network, each Elite switch removes the first route tag from the header and forwards the packet to the next

Elite switch in the route or to the final destination. The routing tag can identify either a single link (used for point-to-point communication) or a group of adjacent links (used for collective communication).

The Elan interface pipelines each packet transmission into the network using wormhole flow control. At the link level, the Elan interface partitions each packet into smaller 16-bit units called flow control digits or *flits* [7]. Every packet closes with an end-of-packet token, but the source Elan normally only sends the end-of-packet token after receipt of a packet acknowledgment token. This process implies that every packet transmission creates a virtual circuit between source and destination. It is worth noting that both acknowledgment and EOP can be tagged to communicate control information. So, for example, the destination can notify the successful completion of a remote non-write block transaction without explicitly sending an extra packet.

Minimal routing between any pair of nodes is accomplished by sending the message to one of the nearest common ancestor switches and from there to the destination [9]. In this way, each packet experiences two routing phases: an adaptive ascending phase (in forward direction) to get to a nearest common ancestor, where the switches forward the packet through the least loaded link; and a deterministic descending phase (in backward direction) to the destination.

Network nodes can send packets to multiple destinations using the network's broadcast capability. For successful broadcast packet delivery, the source node must receive a positive acknowledgment from all the broadcast group recipients. All Elan interfaces connected to the network can receive the broadcast packet but, if desired, the sender can limit the broadcast set to a subset of physically contiguous Elans.

### Global Virtual Memory

Elan can transfer information directly between the address spaces of groups of cooperating processes while maintaining hardware protection between these process groups. This capability (called virtual operation) is a sophisticated extension to the conventional virtual memory mechanism that is based on two concepts: Elan virtual memory and Elan context.

**Elan virtual memory.** Elan contains an MMU to translate the virtual memory addresses issued by the various on-chip functional units (thread processor, DMA engine, and so on) into physical addresses. These physical memory addresses can refer to either Elan local memory (SDRAM) or the node's main memory. To support main memory accesses, the configuration tables for the Elan MMU are synchronized with the main processor's MMU tables so that Elan can access its virtual address space. The system software is responsible for MMU table synchronization and is invisible to programmers.

The Elan MMU can translate between virtual addresses in the main processor format (e.g., a 64-bit word, big-endian architecture, such as that of the AlphaServer) and virtual addresses written in the Elan format (a 32-bit word, little-endian architecture). A processor with a 32-bit architecture (e.g., an Intel Pentium) requires only one-to-one mapping. Figure 4 shows a 64-bit processor mapping. The 64-bit addresses starting at 0x1FF0C808000 are mapped to the Elan's 32-bit addresses starting at 0xC808000. This means that the main processor can directly access virtual addresses in the range 0x1FF0C808000 to 0xFFFFFFFFFFF, and Elan can access the same memory with addresses in the 0xC808000 to 0xFFFFFFFF range. In our example, the user can allocate main memory using `malloc`, and the process heap can grow outside the region directly accessible by the Elan, which is delimited by 0x1FFFFFFFFF. To avoid this problem, both the main and Elan memory

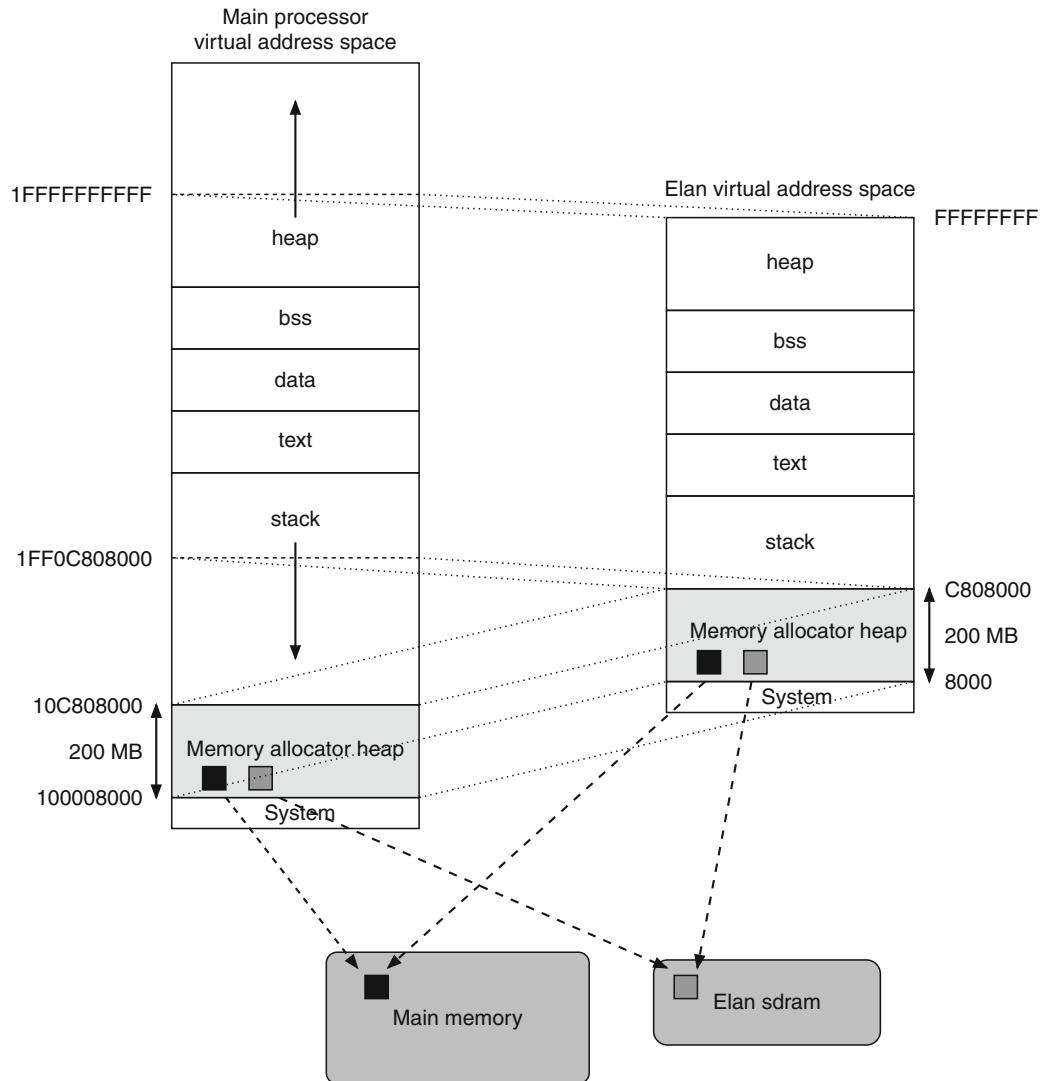
can be allocated using a consistent memory allocation mechanism.

As shown in Fig. 4, the MMU tables can map a common region of virtual memory called the memory allocator heap. The allocator maps, on demand, physical pages (of either main or Elan memory) into this virtual address range. Thus, using allocation functions provided by the Elan library, the user can allocate portions of virtual memory either from main or Elan memory, and the main processor and Elan MMUs can be kept consistent.

**Elan context.** In a conventional virtual-memory system, each user process has an assigned process identification number that selects the MMU table set and, therefore, the physical address spaces accessible to the user process. QsNet extends this concept so that the user address spaces in a parallel program can intersect. Elan replaces the process identification number value with a context value. User processes can directly access an exported segment of remote memory using a context value and a virtual address. Furthermore, the context value also determines which remote processes can access the address space via the Elan network and where those processes reside. If the user process is multithreaded, the threads will share the same context just as they share the same main-memory address space. If the node has multiple physical CPUs, then different CPUs can execute the individual threads. However, the threads will still share the same context.

### Network Fault Detection and Fault Tolerance

QsNet implements network fault detection and tolerance in hardware. (It is important to note that this fault detection and tolerance occurs between two communicating Elans). Under normal operation, the source Elan transmits a packet (i.e., route information for source routing followed by one or more transactions). When the receiver in the destination Elan receives a transaction with an ACK Now flag, it means that this transaction is the last one for the packet. The destination Elan then sends a packet acknowledgment token back to the source Elan. Only when the source Elan receives the packet acknowledgment token does it send an end-of-packet token to indicate the packet transfer's completion. The fundamental rule of Elan network operation



**Quadrics. Fig. 4** Virtual address translation. The *dotted lines* in the figure signify that a segment of memory from one address space maps onto an equally sized segment of memory in another address space

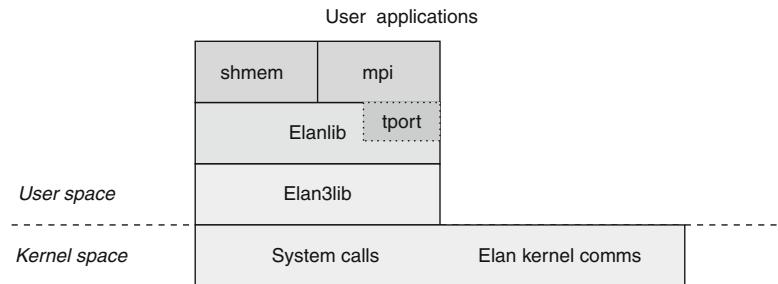
is that for every packet sent down a link, an Elan interface returns a single packet-acknowledgment token. The network will not reuse the link until the destination Elan sends such a token.

If an Elan detects an error during a packet transmission over QsNet, it immediately sends an error message without waiting for a packet-acknowledgment token. If an Elite detects an error, it automatically transmits an error message back to the source and the destination. During this process, the source and destination Elans and the Elites between them isolate the faulty link and/or switch via per-hop fault detection [21]; the

source receives notification about the faulty component and can retry the packet transmission a default number of times. If this is unsuccessful, the source can appropriately reconfigure its routing tables to avoid the faulty component.

### Communication Libraries

With respect to software, QsNet provides several layers of communication libraries that trade off performance with machine independence and programmability. Figure 5 shows the different programming libraries for the Elan network interface. The Elan3lib provides the



**Quadratics. Fig. 5** Elan programming libraries

lowest-level, user space programming interface to the Elan3. At this level, processes in a parallel job can communicate through an abstraction of distributed, virtual, shared memory. Each process in a parallel job is allocated a virtual process identification (VPID) number and can map a portion of its address space into an Elan. These address spaces, taken in combination, constitute a distributed, virtual, shared memory. A combination of a VPID and a virtual address can provide an address for remote memory (i.e., memory on another processing node belonging to a process). The system software and the Elan hardware use VPID to locate the Elan context when they perform a remote communication. Since Elan has its own MMU, a process can select which part of its address space should be visible across the network, determine specific access rights (e.g., write or read only), and select the set of potential communication partners.

Elanlib is a higher-level interface that releases the programmer from the revision-dependent details of Elan and extends Elan3lib with collective communication primitives and point-to-point, tagged message-passing primitives (called tagged message ports or Tports). Standard communication libraries such as that of the MPI-2 standard [12, 24] or Cray SHMEM are implemented on top of Elanlib.

### Elan3lib

The Elan3lib library supports a programming environment where groups of cooperating processes can transfer data directly, while protecting process groups from each other in hardware. The communication takes place at the user level, with no copy, bypassing the operating system. The main features of Elan3lib are the memory

mapping and allocation scheme (described previously), event notification, and remote DMA transfers.

Events provide a general-purpose mechanism for processes to synchronize their actions. Threads running on Elan and processes running on the main processor can use this mechanism. Processes, threads, packets, etc. can access events both locally and remotely. In this way, intranetwork synchronization of processes is possible, and events can indicate the end of a communication operation, such as the completion of a remote DMA. QsNet stores events in Elan memory to guarantee atomic execution of the synchronization primitives. (The current PCI bus implementations cannot guarantee atomic execution, so it is not possible to store events in main memory). Processes can wait for an event by busy-waiting, or polling. In addition, processes can tag an event as block copy. The block-copy mechanism works as follows: A process can initialize a block of data in Elan memory to hold a predefined value. An equivalent-sized block is located in main memory, and both blocks are in the user's virtual address space. When the specified event is set, for example when a DMA transfer has completed, a block copy takes place. That is, the hardware in the Elan, the DMA engine, copies the block in Elan memory to the block in main memory. The user process polls the block in main memory to check its value (by, e.g., bringing a copy of the corresponding memory block into the level-two cache) without polling for this information across the PCI bus. When the value is the same as that initialized in the source block, the process knows that the specified event has occurred.

The Elan supports remote DMA transfers across the network, without any copying, buffering, or operating system intervention. The process that initiates

the DMA fills out a DMA descriptor, which is typically allocated on the Elan memory for efficiency. The DMA descriptor contains source and destination process VPDs, the amount of data, source and destination addresses, two event locations (one for the source and the other for the destination process), and other information that enhances fault tolerance. [Figure 6](#) outlines the typical steps of remote DMA. The command processor referred to in the figure is an Elan microcode thread that processes user commands; it is not a specific microprocessor.

### **Elanlib**

Elanlib is a machine-independent library that integrates the main features of Elan3lib with higher-level collective communication primitives and Tports.

Tports provide basic mechanisms for point-to-point message passing. Senders can label each message with a tag, sender identity, and message size. This information is known as the envelope. Receivers can receive their messages selectively, filtering them according to the sender's identity and/or a tag on the envelope. The Tports layer handles communication via shared memory for processes on the same node. The Tports programming interface is very similar to that of MPI.

Tports implement message sends (and receives) with two distinct function calls: a non-blocking send that posts and performs the message communication, and a blocking send that waits until the matching start send is completed, allowing implementation of different flavors of higher-level communication primitives.

Tports can deliver messages synchronously and asynchronously. They transfer synchronous messages from sender to receiver with no intermediate system buffering; the message does not leave the sender until the receiver requests it. QsNet copies asynchronous messages directly to the receiver's buffers if the receiver has requested them. If the receiver has not requested them, it copies asynchronous messages into a system buffer at the destination.

### **Collective Communication Support**

The basic hardware mechanism that supports collective communication is provided by the Elite switches. The Elite switches can forward a packet to several output ports, with the only restriction that these ports must be

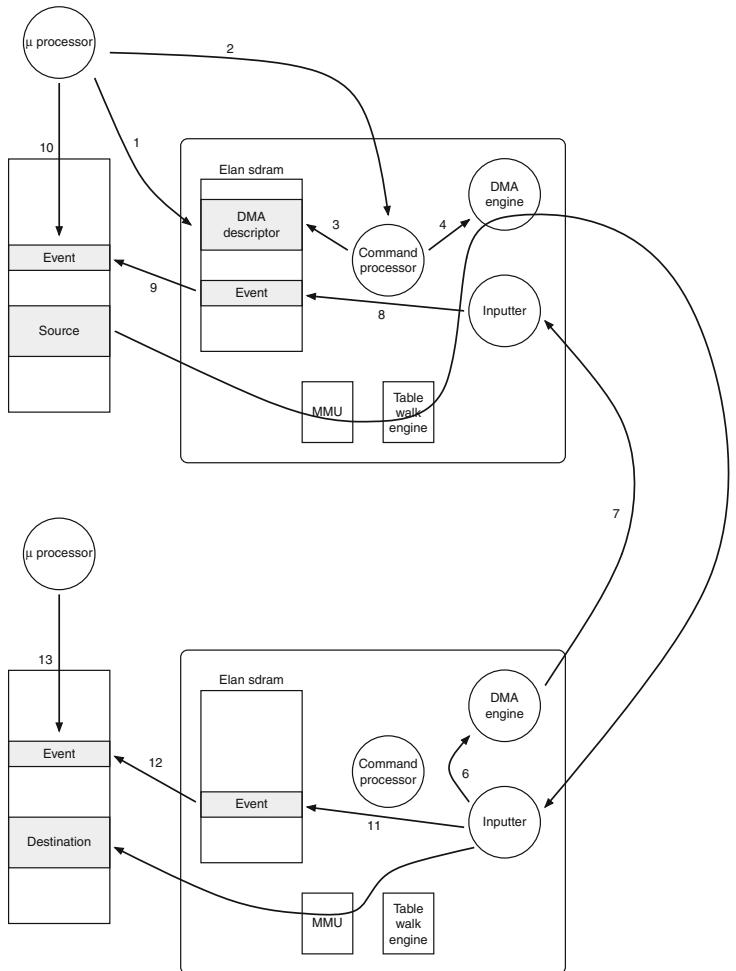
contiguous. In this way, a group of adjacent nodes can be reached by using a single hardware multicast transaction. When the destination nodes are not contiguous, this hardware mechanism is not usable. Instead, a software-based implementation which uses a tree and point-to-point messages, exchanged by the Elans without interrupting their processing nodes, is used. These mechanisms constitute the basic blocks to implement collective communication patterns such as barrier synchronization, broadcast, reduce, and reduce-to-all.

### **Hardware-Based Multicast**

The routing phases for multicast packets differ from those defined for unicast packets. With multicast, during the ascending phase the nearest common ancestor switch for the source node and the destination group is reached. After that, the turnaround routing step is performed and, during the second routing phase, the packet spans the appropriate links to reach all the destination nodes.

The operation of the hardware-based multicast is outlined in [Fig. 7](#). A process in a node injects a multicast packet into the network (see [Fig. 7\(a\)](#)). In [Fig. 7\(b\)](#) the packet reaches the nearest common ancestor switch, and then multiple branches are propagated in parallel. All nodes are capable of receiving a multicast packet, as long as the multicast set is physically contiguous. For a multicast packet to be successfully delivered, a positive acknowledgment must be received from all the recipients of the multicast group. The Elite switches combine the acknowledgments, as pioneered by the NYU Ultra-computer [3, 20], returning a single one to the source (see [Figs. 7\(c\)](#) and [7\(d\)](#)). Acknowledgments are combined in a way that the worst ack wins (a network error wins over an unsuccessful transaction, which on its turn wins over a successful one), returning a positive acknowledgment only when all the partners in the collective communication complete the distributed transaction with success.

The network hardware guarantees the atomic execution of the multicast: either all nodes successfully complete the operation or none. It is worth noting that the multicast packet opens a set of circuits from the source to the destination set, and that multiple transactions (up to 16 in the current implementation) can be pipelined



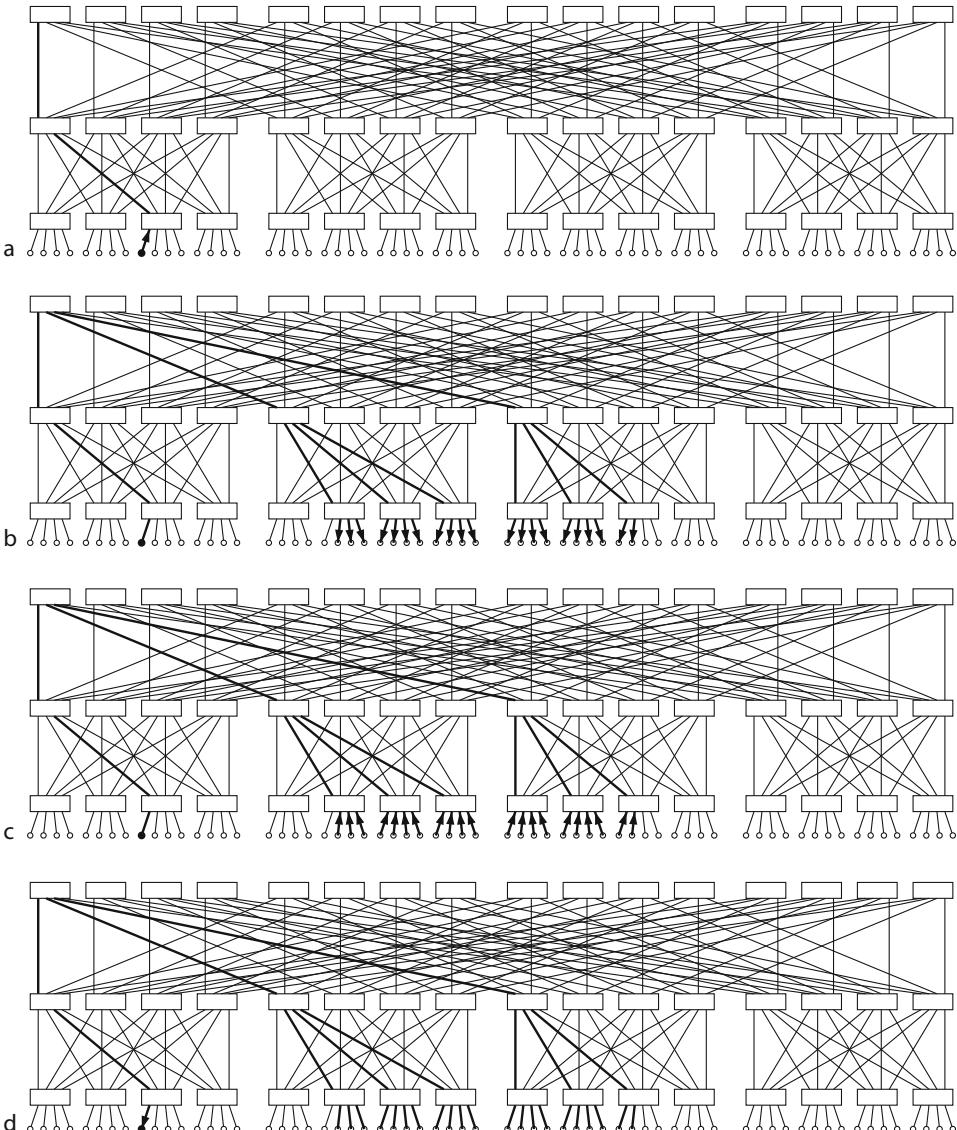
**Quadrics. Fig. 6** Execution of a remote DMA. The sending process initializes the DMA descriptor in the Elan memory (1) and communicates the address of the DMA descriptor to the command processor (2). The command processor checks the correctness of the DMA descriptor (3) and adds it to the DMA queue (4). The DMA engine performs the remote DMA transaction (5). Upon transaction completion, the remote inputter notifies the DMA engine (6), which sends an acknowledgment to the source Elan's inputter (7). The inputters or the hardware in the Elan can notify source (8, 9, 10) and destination (11, 12, 13) events, if needed

within a single packet. For example, it is possible to conditionally issue a transaction based on the result of a previous transaction. This powerful mechanism allows the efficient implementation of sophisticated collective operations and high-level protocols [10, 11].

## Second Generation, QsNetII

The QsNetII network has been designed to optimize the interprocessor communication performance in systems constructed from standard server building blocks. The organization of the network remains the same while

outperforming what it was achieved with QsNet. The network is based on two building blocks: Elan4, a communication processor for the network interface and Elite4, an 8-way high-performance crossbar switch. The network interface incorporates a number of innovative features to minimize latency for short messages, and achieve the maximum bandwidth from a standard PCI-X interface. The network interface has a full 64-bit virtual addressing capability and can perform RDMA operation from user space to user space in 64-bit architectures. As in the QsNet an embedded I/O processor,



**Quadrics. Fig. 7** Hardware-based multicast quadratics. (a) A process in node injects a packet into the network (the ascending routing phase in shown); (b) The packet reaches the destinations passing through a nearest common ancestor switch (descending routing phase shown); (c) The acknowledgments are combined; (d) The issuing process receives a single acknowledgment

which is user programmable can be used to offload asynchronous protocol handling tasks.

As an improvement from the first generation, apart from the technology evolution summarized in [Table 1](#) and [Table 2](#), QsNetII includes specific hardware at Elan level for processing short reads and writes and protocol control to provide very low latencies. Elan4 architecture allows data flow from the PCI-X bus directly to the

output link, thus providing very low latency and high bandwidth.

## Related Entries

- [All-to-All](#)
- [Broadcast](#)
- [Buses and Crossbars](#)
- [Clusters](#)

- [Collective Communication](#)
- [Collective Communication, Network Support for](#)
- [Flow Control](#)
- [Interconnection Networks](#)
- [Meiko](#)
- [MPI \(Message Passing Interface\)](#)
- [Myrinet](#)
- [Network Interfaces](#)
- [Network of Workstations](#)
- [Networks, Fault-Tolerant](#)
- [Networks, Multistage](#)
- [OpenSHMEM - Toward a Unified RMA Model](#)
- [Routing \(Including Deadlock Avoidance\)](#)
- [SCI \(Scalable Coherent Interface\)](#)
- [Scheduling Algorithms](#)
- [Shared-Memory Multiprocessors](#)
- [Switch Architecture](#)
- [Switching Techniques](#)
- [Top500](#)
- [Ultracomputer, NYU](#)

## Bibliographic Notes and Further Reading

As indicated in the introduction, QsNet inherited the architecture of the Meiko CS-2 supercomputer whose design is described in [2]. Many of the performance characteristics of QsNet rely on the fat-tree topology used in the network implementation. Fat-tree networks were first introduced by Charles E. Leiserson in 1985 [14], and later used in the Connection Machine CM5 supercomputer [15]. In [18], Fabrizio Petrini and Marco Vanneschi presented the first formal model of  $k$ -ary  $n$ -trees based on a recursive definition of fat-tree networks built with constant arity switches.

A description and evaluation of point-to-point communication in QsNet is reported in [16], while extensions of the previous paper with an evaluation of permutation patterns, scalability of uniform traffic, analysis of I/O traffic, and collective communication are presented in [5, 17]. In [11], the QsNet high-performance collective communication primitives are shown to be a key factor to provide resource-management functions, orders of magnitude faster than previously reported results.

A novel technique introduced by Quadrics is the use of independent network rails to overcome bandwidth limitations and enhance fault tolerance. In [6], various

venues of exploiting multiple rails are presented and analyzed.

Finally, QsNetII is presented and evaluated in [1] while QsNetIII, that was never commercialized, is described in [22].

## Bibliography

1. Beecroft J, Addison D, Hewson D, McLaren M, Roweth D, Petri尼 F, Nieplocha J (2005) QsNetII: defining high-performance network design. *IEEE Micro* 25(4):34–47
2. Beecroft J, Homewood M, McLaren M (1994) Meiko cs-2 interconnect elan-elite design. *Parallel Comput* 20(10–11):1627–1638
3. Bell G (1992) Ultracomputers: a teraflop before its time. *Communications of the ACM* 35(8):27–47
4. Boden NJ, Cohen D, Felderman RE, Kulawik AE, Seitz CL, Seizovic JN, Su W-K (1995) Myrinet: a gigabit per-second local area network. *IEEE Micro* 15(1):29–36
5. Coll S, Duato J, Mora F, Petri尼 F, Hoisie A (2003) Collective communication patterns on the Quadrics network. In: Getov V, Gerndt M, Hoisie A, Malony A, Miller B, (eds) *Performance analysis and grid computing*, Chapter I, Kluwer Academic, Norwell, MA, pp 93–107
6. Coll S, Frachtenberg E, Petri尼 F, Hoisie A, Gurvits L (2003) Using multirail networks in high-performance clusters. *Concurrency Computat Pract Exper* 15(7–8):625–651
7. Dally WJ, Seitz CL (1987) Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans Comput* C-36(5):547–553
8. Dally WJ, Towles B (2004) Principles and practices of interconnection networks. Morgan Kaufmann, San Francisco, CA
9. Duato J, Yalamanchili S, Ni L (2002) *Interconnection networks: an engineering approach*. Morgan Kaufmann, San Francisco, CA
10. Fernández J, Frachtenberg E, Petri尼 F (2003) BCS-MPI: a new approach in the system software design for large-scale parallel computers. In: ACM/IEEE SC2003, Phoenix, Arizona
11. Frachtenberg E, Petri尼 F, Fernandez J, Pakin S, Coll S (2002) Storm: lightning-fast resource management. In: IEEE/ACM SC2002, Baltimore, MD
12. Gropp W, Huss-Lederman S, Lumsdaine A, Lusk E, Nitzberg B, Saphir W, Snir M (1998) *MPI - the complete reference*, vol 2, the MPI extensions. The MIT Press, Cambridge, London
13. Hellwagner H (1999) The SCI Standard and Applications of SCI. In: Hellwagner H, Reinfeld A (eds) *SCI: scalable coherent interface*, vol 1291 of lecture notes in computer science. Springer, Berlin, pp 95–116
14. Leiserson CE (1985) Fat-Trees: universal networks for hardware efficient supercomputing. *IEEE Trans Comput* C-34(10):892–901
15. Leiserson CE et al (1996) The network architecture of the connection machine CM-5. *J Parallel Distrib Comput* 33(2):145–158
16. Petri尼 F, Vanneschi M (1997)  $k$ -ary  $n$ -trees: high performance networks for massively parallel architectures. In: *Proceedings of the 11th international parallel processing symposium, IPPS'97*, Geneva, Switzerland, pp 87–93 April 1997

17. Petrini F, Vanneschi M (1998) Performance analysis of wormhole routed k-ary n-trees. *Int J Found Comput Sci* 9(2):157–177
18. Petrini F, Feng W-C, Hoisie A, Coll S, Frachtenberg E (2002) The Quadrics network: high-performance clustering technology. *IEEE Micro*, 22(1):46–57
19. Petrini F, Frachtenberg E, Hoisie A, Coll S (2003) Performance evaluation of the quadrics interconnection network. *Cluster Comput* 6(2):125–142
20. Pfister GF, Norton VA (1985) “Hot Spot” contention and combining in multistage interconnection networks. *IEEE Trans Comput C-34(10)*:943–948
21. Quadrics Supercomputers World Ltd (1999) Elan programming manual. Bristol, England, UK
22. Roweth D, Jones T (2008) QsNetIII an adaptively routed network for high performance computing. In: Proceedings of the 2008 16th IEEE symposium on high performance interconnects, HOTI ’08 Washington, DC, pp 157–164
23. Seifert R (1998) Gigabit ethernet: technology and applications for high speed LANs. Addison-Wesley Boston, MA
24. Snir M, Otto S, Huss-Lederman S, Walker D, Dongarra J (1998) MPI - the complete reference, vol 1, the MPI core. The MIT Press, Cambridge, London
25. Tolmie D, Boorman TM, DuBois A, DuBois D, Feng W, Philp I (1999) From HiPPI-800 to HiPPI-6400: a changing of the guard and gateway to the future. In: Proceedings of the 6th international conference on parallel interconnects (PI’99), Anchorage, AK
26. Vogels W, Follett D, Hsieh J, Lifka D, Stern D (2000) Tree-saturation control in the AC3 velocity cluster. In: Proceedings of the eighth symposium on high performance interconnects (HOTI’00), Stanford University, Palo Alto CA

---

## Quantum Chemistry

- [NWChem](#)

---

## Quantum Chromodynamics (QCD) Computations

- [QCD \(Quantum Chromodynamics\) Computations](#)

---

## Quicksort

- [Sorting](#)

# R

## Race

### ► Race Conditions

## Race Conditions

CHRISTOPH VON PRAUN

Georg-Simon-Ohm University of Applied Sciences,  
Nuremberg, Germany

### Synonyms

Access anomaly; Critical race; Determinacy race;  
Harmful shared-memory access; Race; Race hazard

### Definition

A race condition occurs in a parallel program execution when two or more threads access a common resource, e.g., a variable in shared memory, and the order of the accesses depends on the timing, i.e., the progress of individual threads.

The disposition for a race condition is in the parallel program. In different executions of the same program on the same input access events that constitute a race can occur in different order, which may but does not generally result in different program behaviors (non-determinacy).

Race conditions that can cause unintended non-determinacy are programming errors. Examples of such programming errors are violations of atomicity due to incorrect synchronization.

### Discussion

#### Introduction

Race conditions are common on access to shared resources that facilitate inter-thread synchronization or communication, e.g., locks, barriers, or concurrent data structures.

Such structures are called *concurrent objects* [7], since they operate correctly even when being accessed concurrently by multiple threads.

The term *concurrent* refers to the fact that the real-time order in which accesses from different threads execute is not predetermined by the program. Concurrent may but does not necessarily mean *simultaneous*, i.e., that the execution of accesses overlaps in real time.

Concurrent objects can be regarded as *arbiters* that decide on the outcome of race conditions. Threads that participate in a race when accessing a concurrent object will find agreement about the outcome of the race; hence access to concurrent objects is a form of inter-thread *synchronization*.

A common and intuitive correctness criterion for the behavior of a concurrent object is *linearizability* [8], which informally means that accesses from different threads behave as if the accesses of all threads were executed in some total order that is compatible with the program order within each thread and the real-time order.

Linearizability is not a natural property of concurrent systems. Ordinary load and store operations on weakly-ordered shared memory, e.g., can expose many more behaviors than permitted by the strong rules of linearizability. Hence when implementing concurrent objects on such systems, specific algorithms or hardware synchronization instructions are required to ensure correct, i.e., linearizable, operation.

Race conditions on access to resources that are not designed as concurrent objects, e.g., shared memory locations with ordinary load and store operations, are commonly considered as programming errors. An example of such errors are *data races*. Informally, a data race occurs when multiple threads access the same variable in shared memory; the accesses may occur simultaneously and at least one access modifies the variable.

## Model and Formalization

The definition and formalization of the terminology is adopted from Netzer and Miller [18].

**Events** An *event* represents an execution instance of a statement or statement sequence in the program. Every event specifies the set of shared memory locations that are read and/or written. In this entry, events typically refer to individual read or write operations that access a single shared memory location.

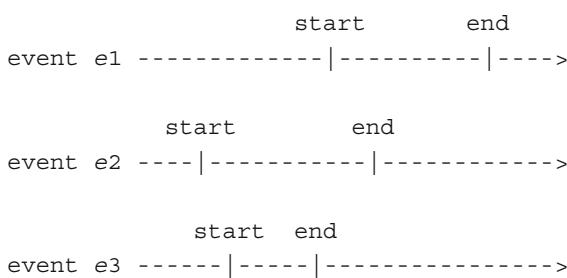
**Simultaneous events** Events do not necessarily occur instantaneously, and in parallel programs, events can be unordered.

A simple model that relates the occurrence of events to wall clock time is as follows: The timing of an event is specified by its *start* and *end* instants. Events  $e_1, e_2$  are *simultaneous* if the start of  $e_1$  occurs after the start of  $e_2$  but before the end of  $e_2$ , or vice versa. [Figure 1](#) illustrates the concept.

**Temporal ordering relation** Events that are not simultaneous are ordered by a *temporal ordering relation*  $\xrightarrow{T}$ , such that  $e_1 \xrightarrow{T} e_2 \Leftrightarrow \text{end}(e_1) < \text{start}(e_2)$ . In the execution of a sequential program  $\xrightarrow{T}$  is a total order, for parallel programs  $\xrightarrow{T}$  is typically a partial order.

For clarity of the presentation and without limiting generality, the temporal ordering relations in the examples of this entry are total orders.

**Shared data dependence relation** Another relation among events is the *shared data dependence relation*  $\xrightarrow{D}$  that specifies how events communicate through memory.  $e_1 \xrightarrow{D} e_2$  holds if  $e_2$  reads a value that was written by  $e_1$ . This relation serves to distinguish executions



**Race Conditions. Fig. 1**  $e_2$  is simultaneous to  $e_1$  and  $e_3$

that have the same temporal ordering but differ in the communication among simultaneous events.

**Program execution** A *program execution* is a triplet  $P = \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$ .

**Feasible program executions** To characterize a race condition in a program execution  $P$ , it is necessary to contrast  $P$  with other possible executions  $P'$  of the same program on the same input.  $P'$  can differ from  $P$  in the temporal ordering or shared data dependence relation (or both). Moreover,  $P'$  should be a *prefix* of  $P$ , which means that each thread in  $P'$  performs the same or an initial sequence of events as the corresponding thread in  $P$ . In other words, the projection for each thread in  $P'$  is the same or a prefix of the projection of the corresponding thread in  $P$ . The set of such program executions  $P'$  is called the set of *feasible program executions*  $X_f$ . A feasible execution is one that adheres to the control and data dependencies prescribed by the program.

It is reasonable to contrast  $P$  with execution prefixes  $P'$ , since the focus of interest is on the earliest point of the program execution  $P$  where a race condition occurred and thus non-determinacy may have been introduced. The execution that follows from that point on may contain further race conditions or erroneous program behavior but these may merely be a consequence of the initial race condition.

$X_f$  contains, e.g., program executions  $P'$  that differ from  $P$  only in the temporal ordering relation. In other words, the timing in  $P'$  can deviate from the timing in  $P$  but not such that any shared data dependencies among events from different threads are resolved differently. Thus  $P$  and  $P'$  have the same functional behavior. An execution that differs from  $P$  in [Fig. 2](#) in such a way is, e.g.,  $P' = \langle \{e_1, e_2, \dots, e_5\}, e_4 \xrightarrow{T'} e_1 \xrightarrow{T'} e_2 \xrightarrow{T'} e_3 \xrightarrow{T'} e_5, \emptyset \rangle$ : Reordering the read of shared variable  $x$  does not alter the shared data dependence relation.

A somewhat more substantial departure from  $P$  but still within the scope of feasible program executions, are executions where shared data dependencies among events from different threads are resolved differently. An execution that differs from  $P$  in [Fig. 2](#) in such a way is, e.g.,  $P'' = \langle \{e_1, e_2, e_3, e_4\}, e_1 \xrightarrow{T''} e_2 \xrightarrow{T''} e_3 \xrightarrow{T''} e_4, e_3 \xrightarrow{D''} e_4 \rangle$ : The update of  $x$  (event  $e_3$ ) occurs in this execution before the read of  $x$  in event  $e_4$ . This induces a new data

|                       |                   |
|-----------------------|-------------------|
| initially $x = 0$     |                   |
| thread-1              | thread-2          |
| (1) $r_1 = x$         | (4) $r_2 = x$     |
| (2) if ( $r_1 == 0$ ) | (5) $x = r_2 + 1$ |
| (3) $x = r_1 + 1$     |                   |

**Race Conditions.** Fig. 2 Feasible execution:

$P = \langle \{e_1, e_2, \dots, e_5\}, e_1 \xrightarrow{T} e_4 \xrightarrow{T} e_2 \xrightarrow{T} e_3 \xrightarrow{T} e_5, \emptyset \rangle$ . Events  $e_i$  correspond to the execution of statements marked (i). There are no shared data dependencies in this execution

dependence. It is sufficient that  $P''$  specifies a prefix of  $P$ , namely, up to the event where the change in the shared data dependence manifests.

Finally, an execution of the program in Fig. 2 that is not feasible:  $P''' = \langle \{e_1, e_2, e_3, e_4, e_5\}, e_4 \xrightarrow{T'''} e_5 \xrightarrow{T'''} e_1 \xrightarrow{T'''} e_2 \xrightarrow{T'''} e_3, e_5 \xrightarrow{P'''} e_1 \rangle$ . This execution is not possible, since  $e_3$  would not be executed if  $e_0$  read a value different from 0.

**Conflicting events** A pair of events is called *conflicting*, if both events access the same shared resource and at least one access modifies the resource [21].

### General Races and Data Races

Given a program execution  $P = \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$ , let  $X_f$  be the set of feasible executions corresponding to  $P$ .

**General race** A *general race* exists between conflicting events  $a, b \in E$ , if there is a  $P' = \langle E', \xrightarrow{T'}, \xrightarrow{D'} \rangle \in E$ , such that  $a, b \in E'$  and

1.  $b \xrightarrow[T']{} a$  if  $a \xrightarrow[T]{} b$ , or
2.  $a \xrightarrow[T']{} b$  if  $b \xrightarrow{} a$ , or
3.  $a \not\xrightarrow{} b$ .

In other words, there is some order among events  $a$  and  $b$  but the order is not predetermined by the program (cases 1 and 2), or executions are feasible where  $a$  and  $b$  occur simultaneously (case 3).

**Data race** A *data race* is a special case of a general race. A data race exists between conflicting memory accesses  $a, b \in E$ , if there is a  $P' = \langle E', \xrightarrow{T'}, \xrightarrow{D'} \rangle \in X$ , such that  $a, b \in E'$  and  $a \xrightarrow[T']{} b$ . In other words, executions are feasible where  $a$  and  $b$  occur simultaneously.

Programmers commonly use *interleaving semantics* when reasoning about concurrent programs. Interleaving semantics mean that the possible behaviors of a parallel program correspond to the sequential execution of events interleaved from different threads. This property of an execution is also called *serializability* [3]. The resulting intuitive execution model is also known as sequential consistency [9].

The motivation for the distinction between data races and general races is to explicitly term those race conditions, where the intuitive reasoning according to interleaving semantics may fail; these are the data races. A common situation where interleaving semantics fails in the presence of data races are concurrent accesses to shared memory in systems that have a memory consistency model that is weaker than sequential consistency [1].

A data race  $\langle a, b \rangle$  means that events  $a$  and  $b$  have the potential of occurring simultaneously in some execution. Simultaneous access can be prevented by augmenting the access sites in the program code with some form of explicit concurrency control. That way, every data race can be turned into a general race that is not a data race.

**Race conditions and concurrent objects** For linearizable concurrent objects, the distinction between data races and general races among events is immaterial. This is because linearizability requires that events on concurrent objects that are simultaneous behave as if they occurred in some order.

### Feasible and Apparent Races

**Feasible races** So far, the definition of a race condition in  $P$  is based on the set of *feasible* program executions  $X_f$ . Thus, locating a race condition requires that the feasibility of a program execution  $P'$ , i.e.,  $P' \in X_f$ , is assessed and that in turn requires a detailed analysis of shared data and control dependencies. This problem NP-hard [16, 17]. Race conditions defined on the basis of feasible program executions are called *feasible general races* and *feasible data races* respectively.

**Apparent races** Due to the difficulty of verifying the feasibility of a program execution  $P$ , common race detection methods widen the set of program executions to consider. The set  $X_a \supseteq X_f$  contains all executions

|                      |                      |          |
|----------------------|----------------------|----------|
| initially $x, y = 0$ | thread-1             | thread-2 |
| (1) $x = 1$          | (3) $r1 = y$         |          |
| (2) $y = 1$          | (4) if ( $r1 == 1$ ) |          |
|                      | (5) $r2 = x$         |          |

**Race Conditions.** Fig. 3 This program does not have explicit synchronization that would restrict the temporal ordering  $\xrightarrow{T'}$ . Thus the execution  $P' = \langle \{e_1, \dots, e_5\}, e_3 \xrightarrow{T} e_4 \xrightarrow{T} e_5 \xrightarrow{T} e_1 \xrightarrow{T} e_2, \emptyset \rangle$  is in  $X_a$ .  $P'$  is however not feasible, since control dependence does not allow execution of  $e_5$  if the read in  $e_3$  returns 0

$P'$  that are prefixes of  $P$  with the following restriction: The temporal ordering  $\xrightarrow{T'}$  obeys the program order and inter-thread dependencies due to explicit synchronization; such relation is also known as the happened-before relation [8]. It is not required that  $P'$  be feasible with respect to shared data dependencies or control dependencies that result from those. Figure 3 gives an example.

Race conditions that are determined using a superset of feasible program executions, e.g.,  $X_a$ , are called *apparent general races* and *apparent data races* respectively. Note that this definition of race conditions depends on the specifics of the synchronization mechanism.

The notion of feasible races is more restricting than the notion of apparent races. This means that an algorithm that detects apparent races considers executions that are not feasible and therefore may report race conditions that could never occur in a real program execution. Such reports are called *spurious races*.

Figure 4 shows another example to illustrate the notion of apparent races further. This program does not have explicit synchronization and has read and write accesses to shared variables  $x$  and  $y$ .

Perhaps surprisingly, the program in Fig. 4 does neither have an apparent nor a feasible data race. The reason is that there is no program execution  $P$  that contains event  $e_3$  or  $e_6$ . Hence the sets  $X_f$  or  $X_a$ , which are defined based on  $P$ , cannot contain those events either and hence neither  $X_f$  nor  $X_a$  would lead us to determine a feasible respectively apparent race condition.

|                      |                      |          |
|----------------------|----------------------|----------|
| initially $x, y = 0$ | thread-1             | thread-2 |
| (1) $r1 = x$         | (4) $r2 = y$         |          |
| (2) if ( $r1 == 1$ ) | (5) if ( $r2 == 1$ ) |          |
| (3) $y = 1$          | (6) $x = 1$          |          |

**Race Conditions.** Fig. 4 Program with read and write accesses to shared variables that does neither have a feasible nor an apparent data race

Some of the literature does not explicitly distinguish apparent and feasible races, i.e., the term “race” may refer to either of the two definitions, depending on the context.

**Actual data races** An *actual data race* exists in execution  $P = \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$ , if events  $a, b \in E$  are conflicting and both events occur simultaneously, i.e.,  $a \xrightarrow{T'} b$ .

### Race Conditions as Programming Errors

Some race conditions are due to programming errors and some are part of the intended program behavior. The distinction between the two is commonly made along the distinction of data races and general races: data races are programming errors, whereas general races that are not data races are considered as “normal” behavior. While such distinction is oftentimes true, exceptions exist and are common.

First, there are correct programs that entail the occurrence of data races at run time. In non-blocking concurrent data structures [13], e.g., data races are intended and considered by the algorithmic design. The development and implementation of such algorithms that “tolerate” race conditions is extremely difficult, since it requires a deep understanding of the shared memory consistency model. Moreover, the implementation of such algorithms is typically not portable, since different platforms may have different shared memory semantics (see ▶“Data-Race-Free Programs”).

Data races that are not manifestations of program bugs are called *benign data races*.

Second, there are incorrect programs that have executions with general races that are not data races. An example for such programming error is a *violation of atomicity* [6]: Accesses to a shared resource occur in

separate critical sections, although the programmer's intent was to have the accesses in the same atomicity scope.

**Non-determinacy** Race conditions can but do not necessarily introduce non-determinacy in parallel programs. For a detailed discussion of this aspect, refer to the entry ▶ “[Determinacy](#)”.

### Data-Race-Free Programs

Data races are an important concept to pinpoint certain behaviors of parallel programs that are manifestations of programming errors. But beyond this role, the concept of data races is also important for system architects who design shared *memory consistency models*.

Intuitively, a shared memory consistency model defines the possible values that a read may return considering prior writes. Consider the program execution in [Fig. 5](#).

Despite the fact that this execution is correct according to the memory consistency model of common multiprocessor architectures [\[23\]](#), such program behavior is difficult to analyze and explain. The reason is, that the program execution is not *serializable*, i.e., there is no interleaving of events from different threads that obeys program order and that would justify the results returned by the reads [\[3\]](#).

Why should such behavior be permitted? For optimization of the memory access path, a processor or compiler can reorder and overlap the execution of memory accesses such that the temporal ordering relation  $\xrightarrow{T}$ , and with it the shared data dependence relation  $\xrightarrow{D}$ , can differ from the program order. For the program in [Fig. 5](#), one of the processors hoists the read before the writes, thus permitting the result.

To allow such optimization on the one hand and to give programmers an intuitive programming abstraction on the other hand, computer architects have

| thread-1        | thread-2        |
|-----------------|-----------------|
| (1) write(x, 1) | (4) write(y, 5) |
| (2) write(x, 2) | (5) write(y, 7) |
| (3) read(y, 5)  | (6) read(x, 1)  |

**Race Conditions. Fig. 5** Program execution with a data race. This execution is not serializable

phrased a minimum guarantee that shared memory consistency models should adhere to: *Data-race-free* programs should only have executions that could also occur on sequentially consistent hardware. In other words, the behavior of parallel programs that are *data-race-free* follows the intuitive interleaving semantics. So what means *data-race-free*?

**Data-race-free in computer architecture** Adve and Hill [\[2\]](#) define a data race as a property of a single program execution: A pair of conflicting accesses participates in a data race, if their execution is not ordered by the happened-before relation [\[8\]](#). The happened-before relation is defined as the combination of program order and the synchronization order due to synchronization events from different threads. This definition allows, unlike the definition according to Netzer and Miller [\[18\]](#), to validate the presence or absence of data race by inspecting a single program execution. An execution is *data-race-free*, if it does not have a data race. A program is data-race-free, if all possible executions are data-race-free. “Possible executions” are thereby implicitly defined as any executions permitted by the processor, for which the memory model is defined.

**Data-race-free in parallel programming languages** The developers of parallel programming languages also specify the semantics of shared memory using a memory model [\[10, 15\]](#). Like memory models at the hardware level, the minimum requirement is that *data-race-free programs* can only have behaviors that could result from sequentially consistent executions.

The definition of what constitutes a data race is however more complex, since building the model entirely around the happened-before relation is not sufficient. This is because the starting point for the definition of data races are programs, not program executions. It is necessary to explicitly define which executions are *feasible* for a given program.

**Causality** Manson et al. [\[10\]](#) define a set requirements called *causality* that a program execution has to meet to be *feasible*. As an example, consider the program in [Fig. 4](#). The *causality* requirement serves to rule out the feasibility of executions that contain events  $e_3$  or  $e_6$ . Intuitively, causality requires that there is a noncyclic

chain of “facts” that justify the execution of a statement. The writes  $e_3, e_6$  may only execute if the reads  $e_1, e_4$  returned the value 1. That in turn requires, that the writes executed. This kind of cyclic justification order is forbidden by the Java memory model. This program is hence data-race-free since executions that include the writes  $e_3, e_6$  are not feasible due to a violation of the causality rule.

More generally, the causality requirement reflects on the overall definition of “data-race-free” as follows: Programs that have no data races *in sequentially consistent executions* are called data-race-free. Execution of such programs are always sequentially consistent.

**Alternatives to causality** Causality has led to significant complexity in the description of the Java memory model and it is not evident that the rules for causality are complete. Hence alternative proposals have been made to specify feasible program executions, e.g., by Saraswat et al. [20].

### Effects of Data Races on Program Optimization

Midkiff and Padua [14] observed that program transformations that preserve the semantics of sequential programs may not be correct for parallel programs with data races. Thus, the presence of data races inhibits the optimization potential of compilers for parallel programs.

### Race Conditions in Distributed Systems

Race conditions can also occur in distributed systems, e.g., when multiple client computers communicate with a shared server. Conceptually, the shared server is a concurrent object that gives certain guarantees on the behavior of concurrent accesses. Typically, concurrency control at the level of the communication protocol or at the level of the server implementation will ensure that semantics follow certain intuitive and well-understood correctness conditions such as linearizability.

In MPI programs [12], e.g., a race condition may occur at a receive operation (`MPI_Recv`) where the source of the message to be received can be arbitrary (`MPI_ANY_SOURCE`). The order in which concurrent messages from different sources are received can be different in different program runs. The receive operation

serializes the arrival of consecutive messages and hence exercises concurrency control.

### Related Entries

- ▶ [Determinacy](#)
- ▶ [Memory Models](#)
- ▶ [Race Detection Techniques](#)

### Bibliographic Notes and Further Reading

Race conditions and data races have been discussed in early literature about the analysis and debugging of parallel programs [4, 5, 11, 19, 22]. An informal characterization of the term “race” was, e.g., given by Emrath and Padua [5]: *A race exists when two statements in concurrent threads access the same variable in a conflicting way and there is no way to guarantee execution ordering.*

A formal definition is given by Netzer and Miller [18], which is based on the existence of program executions with certain properties as discussed by this entry. This definition however does not lend itself to the design of a practical algorithm that validates the presence or absence of a race condition in a given program execution (see the related entry on Race Detection Techniques).

Hence for practical applications, slightly different definitions of data races are common.

In the context of shared memory models, the interest is mainly in characterizing the absence of data races, hence the definition of data-race-free program executions and data-race-free programs in [2, 10, 15].

Most definitions for data races in the context of static analysis tools make a conservative approximation of the program’s data-flow and control-flow and thus are akin to Netzer and Miller’s concept of apparent data races. For dynamic data-race detection, the approximations made for the definition of what constitutes a data race may not be conservative any longer: While the goal of dynamic race detection tools is to detect all feasible races on the basis of a single-input-single-execution (SISE) [4], there are feasible data races according to [18] that may be overlooked according to these definitions.

### Bibliography

1. Adve S, Gharachorloo K (1996) Shared memory consistency models: a tutorial. *IEEE Comput* 29:66–76

2. Adve S, Hill M (June 1993) A unified formalization of four shared-memory models. *IEEE Trans Parallel Distrib Syst* 4(6):613–624
3. Arvind, Maessen J-W (2006) Memory model = instruction reordering + store atomicity. *SIGARCH Comput Archit News* 34(2):29–40
4. Dinning A, Schonberg E (December 1991) Detecting access anomalies in programs with critical sections. In: Proceedings of the ACM/ONR workshop on parallel and distributed debugging, pp 85–96
5. Emrath PA, Padua DA (January 1989) Automatic detection of nondeterminacy in parallel programs. In: Proceedings of the ACM workshop on parallel and distributed debugging, pp 89–99
6. Flanagan C, Freund SN (2004) Atomizer: a dynamic atomicity checker for multithreaded programs. In: POPL'04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on principles of programming languages, ACM, New York, pp 256–267
7. Herlihy MP, Wing JM (July 1990) Linearizability: a correctness condition for concurrent objects. *ACM Trans Program Lang Syst (TOPLAS)* 12:463–492
8. Lamport L (July 1978) Time, clock and the ordering of events in a distributed system. *Commun ACM* 21(7):558–565
9. Lamport L (July 1997) How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans Comput* 46(7):779–782
10. Manson J, Pugh W, Adve S (2005) The Java memory model. In: Proceedings of the symposium on principles of programming languages (POPL'05). ACM, New York, pp 378–391
11. Mellor-Crummey J (May 1993) Compile-time support for efficient data race detection in shared-memory parallel programs. In: Proceedings of the workshop on parallel and distributed debugging, ACM, New York, pp 129–139
12. Message Passing Interface Forum (June 1995) MPI: a message passing interface standard. <http://www.mpi-forum.org/>
13. Michael MM, Scott ML (1996) Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC'96: Proceedings of the 15th annual ACM symposium on principles of distributed computing, ACM, New York, pp 267–275
14. Midkiff S, Padua D (August 1990) Issues in the optimization of parallel programs. In: Proceedings of the international conference on parallel processing, pp 105–113
15. Nelson C, Boehm H-J (2007) Sequencing and the concurrency memory model (revised). The C++ Standards Committee, Document WG21/N2171 J16/07-0031
16. Netzer R, Miller B (1990) On the complexity of event ordering for shared-memory parallel program executions. In: Proceedings of the international conference on parallel processing, Pennsylvania State University, University Park. Pennsylvania State University Press, University Park, pp 93–97
17. Netzer R, Miller B (July 1991) Improving the accuracy of data race detection. Proceedings of the ACM SIGPLAN symposium on principles and practice of parallel programming PPoPP, published in ACM SIGPLAN NOTICES 26(7):133–144
18. Netzer R, Miller B (March 1992) What are race conditions? Some issues and formalizations. *ACM Lett Program Lang Syst* 1(1): 74–88
19. Nudler I, Rudolph L (1988) Tools for the efficient development of efficient parallel programs. In: Proceedings of the 1st Israeli conference on computer system engineering
20. Saraswat VA, Jagadeesan R, Michael M, von Praun C (2007) A theory of memory models. In: PPoPP'07: Proceedings of the 12th ACM SIGPLAN symposium on principles and practice of parallel programming, ACM, New York, pp 161–172
21. Shasha D, Snir M (April 1988) Efficient and correct execution of parallel programs that share memory. *ACM Trans Program Lang Syst* 10(2):282–312
22. Sterling N (January 1993) WARLOCK: a static data race analysis tool. In: USENIX Association (ed) Proceedings of the USENIX winter 1993 conference, San Diego, pp 97–106
23. Weaver DL, Germond T (1991) The SPARC architecture manual (version 9)

## Race Detection Techniques

CHRISTOPH VON PRAUN

Georg-Simon-Ohm University of Applied Sciences,  
Nuremberg, Germany

### Synonyms

Anomaly detection

### Definition

Race detection is the procedure of identifying race conditions in parallel programs and program executions. Unintended race conditions are a common source of error in parallel programs and hence race detection methods play an important role in debugging such errors.

### Discussion

#### Introduction

The following paragraphs summarize the notion of race conditions and data races. An elaborate discussion and detailed definitions are given in the corresponding related entry on “►race conditions.”

*Race conditions.* A widely accepted definition of race conditions is given by Netzer and Miller [47]: A *general race*, or *race* for short, is a pair of conflicting accesses to a shared resource for which the order of accesses is not guaranteed by the program, i.e., the accesses may execute in either order or simultaneously. A subclass of general races are *data races*, for which conflicting memory accesses may occur simultaneously.

*Data races.* The bulk of literature on *race detection* discusses methods for the detection of *data races*. Most data races are programming errors, typically due to *omitted synchronization*. Data races can potentially cause the atomicity of critical sections to fail.

*Violations of atomicity.* Race conditions that are not data races are common in parallel program executions. Some of such races are however programming errors related to *incorrect use of synchronization*. A common example are accesses to a shared resource that occur in separate critical sections, although the programmer's intent was to have the accesses in the same atomicity scope. Such programming errors are commonly called *violations of atomicity* [23]. Taken literally, this terminology seems perhaps too general, since data races, which are non implied by this term, can also cause the atomicity of a critical section to fail.

*Detection accuracy.* An ideal race detection should have two properties: (1) A race condition is reported if the program or a program execution, whatever is checked, has some race condition. This property is called *soundness*. Notice that if there are several race conditions, soundness does not require that all of them are reported. (2) Every reported race condition is a genuine race condition in a feasible program execution. This property is called *completeness* [24].

*Detection methods.* *Static methods* analyze the program code and do not require that the program executes. Sound and complete detection of feasible race conditions in a parallel program by static analysis is in general undecidable due to a theoretical result by Ramalingam [56].

*Dynamic methods.* analyze the event stream from a single program execution, and thus determine the occurrence of a race condition in one program execution. Such analysis can be sound and complete. Naturally, dynamic methods cannot make general statements about the disposition of a race condition in a program. However, the accuracy of a dynamic analysis can be more or less sensitive to the timing of threads. Ideally, a dynamic race detection algorithm has the *single input, single execution* (SISE) property [16], which means that “*A single execution instance is sufficient to determine the existence of an anomaly (data race) for a given input.*” In other words, the detection accuracy should be oblivious to the timing of the execution.

When designing a static or dynamic race detection method, trade-offs between performance, resource usage, and accuracy are made. In practice, it is not uncommon that useful tools are unsound, incomplete, or both with respect to the detection of feasible race conditions.

## Data Race Detection

There are two principal approaches to detect data races: *happened-before-based* (HB-based) and *lockset-based* methods.

### HB-Based Data Race Detection

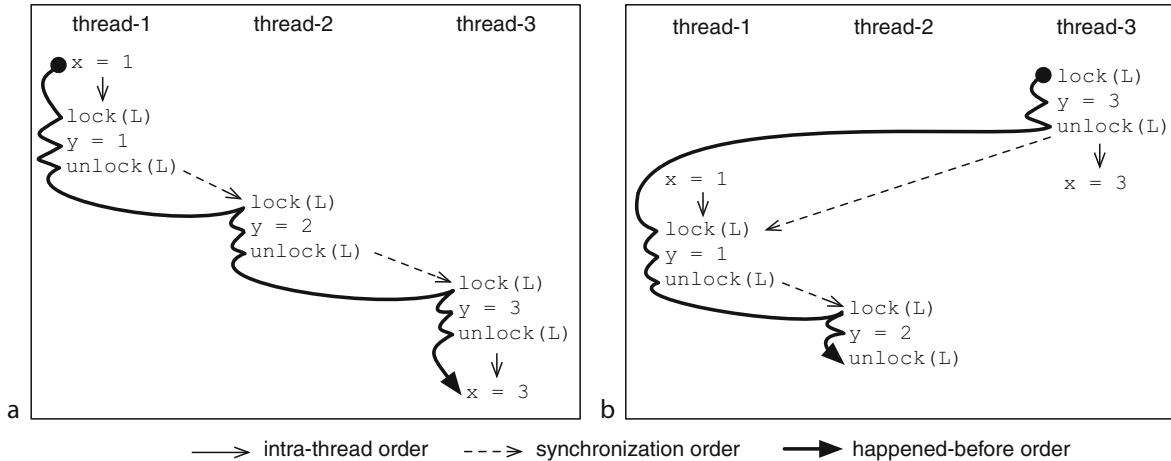
HB-based data race detection operates typically at run-time, thus validating the occurrence of data races in a program execution.

*Data race definition.* A data race  $\langle a, b \rangle$  is deemed to have occurred in a program execution, if  $a$  and  $b$  are conflicting accesses to the same resource, and the accesses are *causally unordered*. The causal order among run-time events in a concurrent systems is thereby defined through the *happened-before relation* by Lamport [31].

This definition assesses the existence of a data race on one particular program execution and its HB-relation. Note that this is different from the definition of feasible and apparent data races by Netzer and Miller [47], since their definition assesses the occurrence of a data race on a set of program executions. Thus, it is possible that an execution is classified as “data-race-free” according to the above (HB-based) defintion but it is not data-race-free according to Netzer and Miller’s defintion [47]. An example of such a program execution, which is detailed in a later paragraph, is given in Fig. 1.

*Happened-before relation.* A pair of events  $\langle a, b \rangle$  is ordered according to the happened-before relation (HB-relation), if  $a$  and  $b$  occur in the same thread and are ordered according to the sequential execution of that thread, or  $a$  and  $b$  are synchronization events in different threads that are ordered due to the synchronization order. Overall, the happened-before relation is a transitive partial order.

The notion of synchronization order depends on the underlying programming model and has been originally described for point-to-point communication in



**Race Detection Techniques.** Fig. 1 In execution (a), the assignments to variable  $x$  in thread-1 and thread-3 are ordered by the HB-relation. There is however a feasible data race, since these assignments may occur simultaneously as, e.g., in execution (b)

distributed systems [31]. The synchronization order has also been defined for synchronization operations on multiprocessor architectures [2, 60], for fork-join synchronization [36], and critical sections [16].

*Algorithmic principles.* HB-based race detectors perform three tasks: (1) track the HB-relation within each thread; (2) keep an access history as sequence of logical timestamps for each shared resource; (3) validate that, for every resource, critical accesses are ordered by the HB-relation.

Various HB-based race detection algorithms have been proposed; they differ in the precise methods and run-time organization of computing the three tasks.

*Accuracy.* If the HB-relation is recorded precisely, HB-based data race detection on a specific program execution can be complete and also sound according to the HB-based definition of data races. However, HB-based data race detection is not sound with respect to data races according to Netzer and Miller's definition [45]: Fig. 1 illustrates two executions of the same program, one where the assignments to  $x$  are ordered by the HB-relation (a), and one where the assignments occur simultaneously (b). An HB-based race detector applied to execution (a) would not report the data race. In other words, HB-based race detection does not have the SISE property.

*Encodings and algorithms for computing the HB-relation.* The HB-relation can be represented and

recorded precisely during a program execution with standard *vector clocks* [19, 33]. Race detectors that build on standard vector clocks are, e.g., the family of Djit detectors by Pozniansky and Schuster [52, 53]. The algorithm requires that maximum number of threads is specified in advance.

A theoretical result from Charron-Bost [11] implies that the precise computation of vector clocks values for  $n$  events in an execution with  $p$  processors requires  $O(np)$  time and space. Due to this result, the encoding of the HB-relation with vector clocks is considered to be impractical [48].

*Performance optimizations.* There are two main sources of overhead in HB-based race detection: First, the tracking of logical time, usually by vector clocks or variations. Second, access operations to shared memory and synchronization operations are instrumented to record access histories and verify orderings.

Flanagan and Freund [22] optimize vector clocks with an adaptive representation of timestamps as follows: The majority of memory accesses target thread-local, lock-protected or shared-read data; for such "safe" access patterns a very efficient and compact timestamp representation is used. This representation is adaptively enhanced to a full vector clock when the access patterns deviate from the safe ones; then, access ordering can be validated according to the HB-relation.

Variants of vector clocks with alternative encodings of the HB-relation have been developed to improve the efficiency of tracking logical time and checking the ordering among pairs of accesses. Examples for such encodings are the *English-Hebrew labeling* (EH) by Nudler and Rudolph [50], *task recycling* (TR) by Dinning and Schonberg [15], and *offset-span labeling* (OS) by Mellor-Crummey [36]. These algorithms make trade-offs at different levels: For example, OS does, unlike TR, avoid global communication when computing the timestamps; this makes the algorithm practical for distributed environments. However, OS is limited to nested fork-join synchronization patterns.

The implementation of encodings can be optimized further for efficiency, sometimes however at the cost of reducing the accuracy of the detection. Dinning and Schonberg [15], e.g., limit the size of access histories per variable. Some data races may not be reported due to this optimization, which seemed to occur infrequently in their experiments.

A similar optimization is described by Choi and Min [13]: Their algorithm records only the most recent read and write timestamp for a shared variable; as a consequence, data races that occurred in an execution may not be reported. They argue however that this optimization preserves the soundness of the detection, i.e., if there is some data race according to the HB-definition, it is reported. An iterative debugging process based on deterministic replay will eventually identify all data races and thus lead to a program execution that is data-race-free (again according to the HB-definition). The significant contribution of Choi and Min [13] is their systematic and provably correct debugging methodology for data races.

Compilers can reduce the overhead of race detection by pruning the instrumentation of accesses that are provably not involved in data races, such as access to thread-local data. Moreover, accesses that lead to *redundant access events* [7, 37] at run-time may also be recognized by a compiler. Static analyses of concurrency properties support this process (see “Static Analysis”).

Even if redundant accesses cannot be safely recognized by a compiler, techniques have been developed by Choi et al. [14] to recognize and filter redundant access events at run-time.

Another run-time technique to reduce the overhead of dynamic data race detection is to include only small fractions of the overall program execution (sampling) in the checking process [34]. Perhaps surprisingly, the degradation of accuracy due to an incomplete event trace is moderate and acceptable in the light of significant performance gains.

*Race detection in distributed systems.* Race detection in distributed systems refers to HB-based data race detection in distributed shared memory (DSM) systems. Perkovic and Keleher, e.g., [51] present an optimized DSM protocol where the tracking of HB-information is piggybacked on the coherence protocol. Another DSM race detection technique is presented by Richards and Larus [57].

## Lockset-Based Data Race Detection

Lockset-based data race detection is a technique tailored to programs that use critical sections as their primary synchronization model. Instead of validating the absence of data races directly, the idea of lockset-based data race detection is to validate that a program or program execution adheres to a certain programming policy, called *locking discipline* [59].

*Locking discipline.* A simple locking policy could, e.g., state that threads that access a common memory location must hold a mutual exclusion lock when performing the access. Compliance with this locking discipline implies that executions are data-race-free. The validation of the locking discipline is done with static or dynamic program analysis, or combinations thereof.

The key ideas of lockset-based data race detection are due to Savage et al. [59], although the concept of *lock covers*, which is related to locksets is discussed earlier by Dinning and Schonberg [16].

*Algorithmic principles.* Checking a locking discipline works as follows: Each thread tracks at run-time the set of locks it currently holds. Conceptually, each variable in shared memory has a shadow location that holds a lockset. On the first access to a shared variable, the shadow memory is initialized with the lockset of the current thread. On subsequent accesses, the lockset in shadow memory is updated by intersecting it with the lockset of the accessing thread. If the intersection is

empty and the variable has been accessed by different threads, a potential data race is reported.

*Accuracy.* Lockset-based detection is sound. In fact, the validation of a locking discipline has the SISE property, i.e., all race conditions that occur in a certain execution are reported. The detection is however incomplete, since accesses that violate the locking discipline, may be ordered by other means of synchronization.

The principles of lockset-based race detection have been refined with two goals in mind: To increase the accuracy (reduce overreporting) and to improve the performance of dynamic checkers.

*Increasing the accuracy.* There are several common parallel programming idioms that are data-race-free but violate the simple locking discipline stated before. Examples are initialization without lock protection, read-sharing, or controlled data handoff. To accommodate these patterns, the simple locking policy can be extended as follows: An abstract state machine is associated with each shared memory location. The state machine is designed to identify safe programming idioms in the access stream and to avoid overreporting.

For example, read and write accesses of the initializing thread can proceed safely, even without locking. If other threads read the variable and only read accesses follow, a read-sharing pattern is deemed to have occurred and no violation of the locking policy is reported, even if the accessing threads do not hold a common lock at the time of the accesses [59]. Subsequent work has picked up these ideas and refined the state model to capture more elaborate sharing patterns that are safe [29, 67].

The state model can introduce unsoundness, namely, in cases where the timing of threads in an execution lets an access sequence look like a safe programming idiom, where indeed an actual data race occurred. In practice, the benefits of reduced overreporting outweigh the drawback due to potential unsoundness.

*Hybrid data race detection.* Some methods for dynamic data race detection [16, 50, 52, 70] combine the lockset algorithm [59] with checks of Lamport's happened-before relation. Such a procedure mitigates the shortcomings of either approach and improves the accuracy of the detection.

*Performance optimizations.* Dynamic tools can be supported by a compiler that reduces instrumentation and thus the frequency of run-time checks. For example, accesses that are known at compile-time not to participate in data races don't have to be instrumented [14, 50, 67].

Another strategy to reduce the run-time overhead is to track the lockset and state information in shadow memory not per fixed-size variable but at a larger granularity, e.g., at the granularity of programming language objects [67] or minipages [71].

*Application to non-blocking synchronization models.* Although most work has studied programs with blocking synchronization (locks), the principles are also applicable to optimistic synchronization[55, 69].

## Dynamic Methods

So far, data race detection has been described as a dynamic program analysis. Dynamic methods are also called *trace-based*, since subject of the analysis is an execution trace.

*Architecture.* There are different styles for organizing the data collection and analysis phase:

- *Post-mortem methods* record a complete set of relevant run-time events in a persistent trace for *offline analysis*. This method was common in early race detection systems, e.g., by Allan and Padua [4]. Due to the possibly large size of the trace, this method is limited to short program runs.
- *Online methods*, also called *on-the-fly detection*, record events temporarily and perform the data race analysis entwined with the actual program execution. Much of the recorded information can be discarded as the analysis proceeds. Depending on the algorithm, this may but does not necessarily compromise the accuracy of the detection. Most dynamic data race tools choose this architecture, e.g., Mellor-Crummey [36] for an HB-based analysis and Savage et al. [59] for a method based on locksets. In distributed systems, online methods are also the preferred analysis architecture.

*Implementation techniques.* Dynamic race detection is an aspect that is crosscutting and orthogonal to the remaining functionality of a software. Several

techniques have been used to incorporate race detection in a software system:

- *Program instrumentation* augments memory and synchronization accesses with checking code. Instrumentation can occur at compile-time [59], or at run-time [14].
- Race detection as part of a software-based *protocol implementation* in distributed shared memory systems [48, 51, 57].
- *Hardware extensions* for HB-based [3, 40, 42, 54] or lock-set-based [72] race detection.

*Limitations.* An inherent limitation of dynamic race detection is that it is based on information from a single program execution.

*This limitation is twofold.* First, not all possible control paths may have been exercised on the given input. Second, possible race conditions may have been covert in the specific thread schedule of the execution trace, as e.g., in example in Fig. 1. This second limitation, called *scheduling dependence*, can be mitigated or entirely avoided by so-called *predictive analyses* [12, 25, 61, 68]. The key idea of predictive (dynamic) analyses is to consider not only the order of events recorded in a specific execution trace but also permutations thereof. This technique increases the coverage of the dynamic analysis of concurrent program executions, since it is capable to expose thread interleavings (event orders) that have not been exercised.

## Static Methods

Static program analysis, does not require that the program is executed.

*Pragmatic methods.* A pragmatic approach to find programming errors is to identify situations in the source code, where the program deviates from common programming practice. Naturally, this approach lends itself to find deviations from common programming idioms in concurrent programs that may lead to unintended race conditions. This approach is called pragmatic, since the analysis is neither sound nor complete with respect to identifying data races. In practice, pragmatic analysis methods have turned out to be very effective with the additional benefit that in most cases, analyses do not require sophisticated data-flow or concurrency analysis and hence can be very efficient.

Findbugs by Ayewah et al. [5] is a tool for pragmatic analyses of Java programs. Hovenmeyer and Pugh [28] describe numerous analyses for concurrency related errors. One programming idiom for concurrent program in Java is, e.g., that accesses to a shared mutable variable are consistently protected by synchronized blocks. Violations of this programming practice are described as *bug pattern* called “inconsistent synchronization.” Inconsistent synchronization occurs, if a class contains mixed (synchronized and unsynchronized) accesses to a field variable and no more than one third of the accesses (writes weighed higher than reads) occur outside a synchronized block.

RacerX [18] is a static data race and deadlock detection system targeted to check large operating system codes. The tool builds a call graph and verifies a locking discipline along a calling context sensitive traversal of the code. The system does however not have a pointer analysis and approximates aliasing through variable types. Moreover, the system uses a heuristic to classify sections of code that are sequential and those that are concurrent. These approximations facilitate the analysis of very large codes (>500 K lines of code). As the analysis issues a significant number of spurious reports, a clever ranking scheme is used to prioritize the large numbers of reports according to their likelihood of being an actual bug.

*Methods based on data-flow analysis.* May-happen-in-parallel analysis (MHP) is the foundation of many compile-time analyses for concurrent programs. MHP analysis approximates the order of statements executed by different threads and computes the may-happen-in-parallel relation among statements. MHP analysis in combination with a compile-time model of program data can serve as the foundation of compile-time data race detection.

Bristow [9] used an *inter-process precedence graph* for determining anomalies in programs with post-wait synchronization. Taylor [65] and Duesterwald and Soffa [17] extend this work and define a model for parallel tasks in Ada programs with rendez-vous synchronization. The program representation in [17] is modular and enables an efficient analysis of programs with procedures and recursion based on a data-flow framework. Masticola and Ryder [35] generalize and improve the approach of [17]. Naumovic et al. [44] compute the potential concurrency in Java programs at the

level of statements. The authors have shown that the precision of their data-flow algorithm is optimal for most of the small applications that have been evaluated. The approach requires that the number of real threads in the system is specified as input to the analysis. The combination of MHP information with a model of program data (heap shape and reference information) could be used to determine conflicting data accesses. This approach is discussed by Midkiff, Lee, Padua, and Sura [38, 64].

Static race detection for Java programs has been developed, e.g., by Choi et al. [14] and Naik et al. [43]. Both systems are based on a whole program analysis to determine an initial set of potentially conflicting object accesses. This set is pruned (refined) by several successive analysis steps, which are alias analysis, thread escape analysis, and lockset analysis. The Chord checker by Naik et al. [43] is object context-sensitive [32, 39], and this feature is found to be essential, in combination with a precise, inclusion-based alias analysis, to achieve a high detection accuracy; object context sensitivity incurs however a significant scalability cost. Chord sacrifices soundness since it approximates lock identity through *may alias* information when determining common lock protection.

*Type-based methods.* Type systems can model and express data protection and locking policies in data and method declarations. Compliance of data access and method invocations with the declared properties can be checked mostly statically. The main advantage of the type-based approach is its modularity, which makes it, in contrast to a whole program analysis, well amenable to treat incomplete and large programs. The type systems that can prove data-race-freedom have either been proposed as extensions to existing programming languages by Bacon et al. [6] or Boyapati and Rinard [8]. Flanagan and Freund [1, 20] present a type system that is able to specify and check lock-protection of individual variables. In combination with an annotation generator [21], they applied the type checker to Java programs of up to 450 KLOC. The annotation generator is able to recognize common locking patterns and further uses heuristics to classify as benign certain accesses without lock protection. The heuristics are effective in reducing the number of spurious warnings; some are however unsound, which has not been a problem for the benchmarks investigated in [21].

*Model checking.* The principle of model checking is to explore every possible control flow-path and variable value assignment for undesired program behavior. Since this procedure is obviously intractable, models of data and program are explored instead. An additional source of complexity in parallel versus sequential programs is the timing of threads resulting in myriads of possible interleaving of actions from different threads.

The main challenge of model checking for concurrency errors, such as data races, is hence to reduce the state space to be explored. One idea is to consider only those states and transitions of individual threads that operate on shared data, and are thus visible to other threads. Another idea is to aggregate the possible interleavings, e.g., by modeling multiple threads and their transitions in one common state transition space.

Model checking has been applied to the detection of access anomalies in concurrent programs, e.g., in [27, 30, 41, 62, 63, 66]. Stoller’s model checker [63] verifies adherence to a locking policy. Henzinger et al. [27] describe a model checker that identifies conflicting accesses that are not ordered according to the HB-relation. Both works assume an underlying sequentially consistent execution platform. Model checking for executions on weakly-ordered memory systems have been conceived as well by [10], though not particularly for the purpose of data race detection.

## Detection of Determinacy and Atomicity Violations

The emphasis of this entry is on the detection of race conditions that are data races. Related to race conditions are also *violations of determinacy* and *violations of atomicity*. Techniques for their detection are discussed in the corresponding essays on ►determinacy and atomicity.

## Complexity

Netzer and Miller [47] characterize race conditions in terms of feasible programs executions. A program execution is feasible, if the control flow and shared data-dependencies in the execution could actually occur at run-time in accordance with the semantics of the program.

The problem of detecting a race condition is at least as hard as determining if a feasible program execution exists, where a pair of suspect statements occurs

concurrently. Netzer and Miller [46] found that problem to be intractable even if shared data-dependencies are ignored. Helmbold and McDowell [26] confirm and refine this result by restricting programs to certain control-flow and synchronization models. For unrestricted programs, if data-dependencies are not ignored, the problem of deciding if two conflicting statement participate in a race is as hard as the halting problem [26].

In practice, most dynamic and also static data-flow-based race detection analyses have a complexity that is quasi-linear in the number of synchronization and shared resource accesses.

## Related Entries

- [Determinacy](#)
- [Formal Methods-Based Tools for Race, Deadlock, and Other Errors](#)
- [Race Conditions](#)

## Bibliography

1. Abadi M, Flanagan CE, Freund SN (2006) Types for safe locking: static race detection for Java. *Trans Program Lang Syst (TOPLAS)* 28(2):207–255
2. Adve S, Hill M (June 1990) Weak ordering — A new definition. In: Proceedings of the annual international symposium on computer architecture (ISCA'90), pp 2–14
3. Adve S, Hill M, Miller B, Netzer R (May 1991) Detecting data races on weak memory systems. In: Proceedings of the annual international symposium on computer architecture (ISCA'91), pp 234–243
4. Allen TR, Padua DA (August 1987) Debugging fortran on a shared memory machine. In: Proceedings of the international conference on parallel processing, pp 721–727
5. Ayewah N, Hovemeyer D, Morgenthaler JD, Penix J, Pugh W (2008) Using static analysis to find bugs. *IEEE Softw* 25(5):22–29
6. Bacon D, Strom R, Tarafdar A (October 2000) Guava: a dialect of Java without data races. In: Proceedings of the conference on object-oriented programming, systems, languages, and applications (OOPSLA'00), pp 382–400
7. Balasundaram V, Kennedy K (1989) Compile-time detection of race conditions in a parallel program. In: Proceedings of the international conference on supercomputing (ISC'89), pp 175–185
8. Boyapati C, Lee R, Rinard M (November 2002) Ownership types for safe programming: preventing data races and deadlocks. In: Proceedings of the conference on object-oriented programming, systems, languages, and applications (OOPSLA'02), pp 211–230
9. Bristow G, Dreay C, Edwards B, Riddle W (1979) Anomaly detection in concurrent programs. In: Proceedings of the international conference on software engineering (ICSE'79), pp 265–273
10. Burckhardt S, Alur R, Martin MMK (2007) CheckFence: checking consistency of concurrent data types on relaxed memory models. In: PLDI'07: Proceedings of the 2007 ACM SIGPLAN conference on programming language design and implementation. ACM, New York, pp 12–21
11. Charron-Bost B (1991) Concerning the size of logical clocks in distributed systems. *Inf Process Lett* 39(1):11–16
12. Chen F, Serbanuta TF, Rosu G (2008) jpredictor: a predictive runtime analysis tool for java. In: ICSE'08: Proceedings of the 30th international conference on software engineering. ACM, New York, pp 221–230
13. Choi J-D, Min SL (1991) Race frontier: reproducing data races in parallel program debugging. In: PPOPP'91: Proceedings of the third ACM SIGPLAN symposium on principles and practice of parallel programming. ACM, New York, pp 145–154
14. Choi J-D, Lee K, Loginov A, O'Callahan R, Sarkar V, Sridharan M (June 2002) Efficient and precise datarace detection for multithreaded object-oriented programs. In: Conference on programming language design and implementation (PLDI'02), pp 258–269
15. Dinning A, Schonberg E (1990) An empirical comparison of monitoring algorithms for access anomaly detection. In: PPOPP'90: Proceedings of the second ACM SIGPLAN symposium on principles & practice of parallel programming. ACM, New York, pp 1–10
16. Dinning A, Schonberg E (December 1991) Detecting access anomalies in programs with critical sections. In: Proceedings of the ACM/ONR workshop on parallel and distributed debugging, pp 85–96
17. Duesterwald E, Soffa M (1993) Concurrency analysis in the presence of procedures using a data-flow framework. In: Proceedings of the symposium on testing, analysis, and verification (TAV4), pp 36–48
18. Engler D, Ashcraft K (October 2003) RacerX: Effective, static detection of race conditions and deadlocks. In: Proceedings of the symposium on operating systems principles (SOSP'03), pp 237–252
19. Fidge CJ (1988) Timestamp in message passing systems that preserves partial ordering. In: Proceedings of the 11th Australian computing conference, pp 56–66
20. Flanagan C, Freund SN (June 2000) Type-based race detection for Java. In: Proceedings of the conference on programming language design and implementation (PLDI'00), pp 219–229
21. Flanagan C, Freund SN (June 2001) Detecting race conditions in large programs. In: Proceedings of the workshop on program analysis for software tools and engineering (PASTE'01), pp 90–96
22. Flanagan C, Freund SN (2009) FastTrack: Efficient and precise dynamic race detection. In: PLDI'09: Proceedings of the 2009 ACM SIGPLAN conference on programming language design and implementation. ACM, New York, pp 121–133
23. Flanagan C, Qadeer S (June 2003) A type and effect system for atomicity. In: Proceedings of the conference on programming language design and implementation (PLDI'03), pp 338–349
24. Flanagan C, Leino R, Lillibridge M, Nelson G, Saxe J, Stata R (June 2002) Extended static checking for Java. In: Proceedings of the

- conference on programming language design and implementation (PLDI'02), pp 234–245
- 25. Flanagan C, Freund SN, Yi J (2008) Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In: PLDI'08: Proceedings of the 2008 ACM SIGPLAN conference on programming language design and implementation. ACM, New York, pp 293–303
  - 26. Helmbold DP, McDowell CE (September 1994) A taxonomy of race detection algorithms. Technical Report UCSC-CRL-94-35, University of California, Santa Cruz, Computer Research Laboratory
  - 27. Henzinger TA, Jhala R, Majumdar R (2004) Race checking by context inference. In: PLDI'04: Proceedings of the ACM SIGPLAN 2004 conference on programming language design and implementation. ACM, New York, pp 1–13
  - 28. Hovemeyer D, Pugh W (July 2004) Finding concurrency bugs in java. In: Proceedings of the PODC workshop on concurrency and synchronization in Java programs
  - 29. Jannesari A, Bao K, Pankratius V, Tichy WF (2009) Helgrind+: An efficient dynamic race detector. In: Proceedings of the 23rd international parallel & distributed processing symposium (IPDPS'09). IEEE, Rome
  - 30. Kidd N, Reps T, Dolby J, Vaziri M (2009) Finding concurrency-related bugs using random isolation. In: VMCAI'09: Proceedings of the 10th international conference on verification, model checking, and abstract interpretation. Springer-Verlag, Heidelberg, pp 198–213
  - 31. Lamport L (July 1978) Time, clock and the ordering of events in a distributed system. Commun ACM 21(7):558–565
  - 32. Lhoták O, Hendren L (March 2006) Context-sensitive points-to analysis: is it worth it? In: Mycroft A, Zeller A (eds) International conference of compiler construction (CC'06), vol 3923 of LNCS. Springer, Vienna, pp 47–64
  - 33. Mattern F (1988) Virtual time and global states of distributed systems. In: Proceedings of the Parallel and distributed algorithms conference. Elsevier Science, Amsterdam, pp 215–226
  - 34. Marino D, Musuvathi M, Narayanasamy S (2009) Literace: effective sampling for lightweight data-race detection. In: PLDI'09: Proceedings of the 2009 ACM SIGPLAN conference on programming language design and implementation. ACM, New York, pp 134–143
  - 35. Masticola S, Ryder B (1993) Non-concurrency analysis. In: Proceedings of the symposium on principles and practice of parallel programming (PPoPP'93), pp 129–138
  - 36. Mellor-Crummey J (November 1991) On-the-y detection of data races for programs with nested fork-join parallelism. In: Proceedings of the supercomputer debugging workshop, pp 24–33
  - 37. Mellor-Crummey J (May 1993) Compile-time support for efficient data race detection in shared-memory parallel programs. In: Proceedings of the workshop on parallel and distributed debugging, pp 129–139
  - 38. Midkiff S, Lee J, Padua D (June 2001) A compiler for multiple memory models. In: Rec. Workshop compilers for parallel computers (CPC'01)
  - 39. Milanova A, Rountev A, Ryder BG (2005) Parameterized object sensitivity for points-to analysis for Java. ACM Trans Softw Eng Methodol 14(1):1–41
  - 40. Min SL, Choi J-D (1991) An efficient cache-based access anomaly detection scheme. In: ASPLOS-IV: Proceedings of the 4th international conference on architectural support for programming languages and operating systems. ACM, New York, pp 235–244
  - 41. Musuvathi M, Qadeer S, Ball T, Basler G, Nainar PA, Neamtiu I (2008) Finding and reproducing heisenbugs in concurrent programs. In: OSDI'08: Proceedings of the 8th USENIX conference on operating systems design and implementation. USENIX Association, Berkeley, pp 267–280
  - 42. Muzahid A, Suárez D, Qi S, Torrellas J (2009) Sigrace: signature-based data race detection. In: ISCA'09: Proceedings of the 36th annual international symposium on computer architecture. ACM, New York, pp 337–348
  - 43. Naik M, Aiken A, Whaley J (June 2006) Effective static race detection for Java. In: Proceedings of the conference on programming language design and implementation (PLDI'06), pp 308–319
  - 44. Naumovich G, Avrunin G, Clarke L (September 1999) An efficient algorithm for computing MHP information for concurrent Java programs. In: Proceedings of the European software engineering conference and symposium on the foundations of software engineering, pp 338–354
  - 45. Netzer R, Miller B (August 1990a) Detecting data races in parallel program executions. Technical report TR90-894, Department of Computer Science, University of Wisconsin, Madison
  - 46. Netzer R, Miller B (January 1990b) On the complexity of event ordering for shared-memory parallel program executions. Technical report TR 908, Computer Sciences Department, University of Wisconsin, Madison
  - 47. Netzer R, Miller B (March 1992) What are race conditions? Some issues and formalizations. ACM Lett Program Lang Syst 1(1): 74–88
  - 48. Netzer R, Brennan T, Damodaran-Kamal S (1996) Debugging race conditions in message-passing programs. In: SPDT'96: Proceedings of the SIGMETRICS symposium on parallel and distributed tools. ACM, New York, pp 31–40
  - 49. Nudler I, Rudolph L (1988) Tools for the efficient development of efficient parallel programs. In: Proceedings of the 1st Israeli conference on computer system engineering
  - 50. O'Callahan R, Choi J-D (June 2003) Hybrid dynamic data race detection. In: Symposium on principles and practice of parallel programming (PPoPP'03), pp 167–178
  - 51. Perkovic D, Keleher PJ (October 1996) Online data-race detection via coherency guarantees. In: Proceedings of the 2nd symposium on operating systems design and implementation (OSDI'96), pp 47–57
  - 52. Pozniansky E, Schuster A (June 2003) Efficient on-the-y data race detection in multi-threaded c++ programs. In: Proceedings of the symposium on principles and practice of parallel programming (PPoPP'03), pp 179–190
  - 53. Pozniansky E, Schuster A (2007) Multirace: efficient on-the-y data race detection in multithreaded c++ programs: research articles. Concurrency Comput: Pract Exper 19(3):327–340

54. Prvulovic M, Torrellas J (2003) Reenact: using thread-level speculation mechanisms to debug data races in multithreaded codes. *SIGARCH Comput Archit News* 31(2):110–121
55. Rajwar R, Goodman JR (2001) Speculative lock elision: enabling highly concurrent multithreaded execution. In: *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on microarchitecture*. IEEE Computer Society, Washington, DC, pp 294–305
56. Ramalingam G (2000) Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans Program Lang Syst (TOPLAS)* 22:416–430
57. Richards B, Larus JR (1998) Protocol-based data-race detection. In: *SPDT'98: Proceedings of the SIGMETRICS symposium on parallel and distributed tools*. ACM, New York, pp 40–47
58. Savage S, Burrows M, Nelson G, Sobalvarro P, Anderson T (October 1997a) Eraser: a dynamic data race detector for multi-threaded programs. In: *Proceedings of the symposium on operating systems principles (SOSP'97)*, pp 27–37
59. Savage S, Burrows M, Nelson G, Sobalvarro P, Anderson T (1997b) Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on computer systems* 15(4): 391–411
60. Scheurich C, Dubois M (June 1987) Correct memory operation of cache-based multiprocessors. In: *Proceedings of 14th annual symposium on computer architecture, Computer Architecture News*, pp 234–243
61. Sen K, Rosu G, Agha G (2003) Runtime safety analysis of multithreaded programs. In: *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on foundations of software engineering*. ACM, New York, pp 337–346
62. Shacham O, Sagiv M, Schuster A (2005) Scaling model checking of dataraces using dynamic information. In: *PPoPP'05: Proceedings of the tenth ACM SIGPLAN symposium on principles and practice of parallel programming*. ACM, New York, pp 107–118
63. Stoller SD (October 2002) Model-checking multi-threaded distributed Java programs. *Int J Softw Tools Technol Transfer* 4(1): 71–91
64. Sura Z, Fang X, Wong C-L, Midkiff SP, Lee J, Padua DA (June 2005) Compiler techniques for high performance sequentially consistent Java programs. In: *Proceedings of the symposium principles and practice of parallel programming (PPoPP'05)*, pp 2–13
65. Taylor RN (May 1983) A general purpose algorithm for analyzing concurrent programs. *Commun ACM* 26(5):362–376
66. Visser W, Havelund K, Brat G, Park S (2000) Model checking programs. In: *ASE'00: Proceedings of the 15th IEEE international conference on automated software engineering*. IEEE Computer Society, Washington, p 3
67. von Praun C, Gross T (October 2001) Object race detection. In: *Conference on object-oriented programming, systems, languages, and applications (OOPSLA'01)*, pp 70–82
68. Wang L, Stoller SD (2006) Accurate and efficient runtime detection of atomicity errors in concurrent programs. In: *PPoPP'06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, New York, pp 137–146
69. Welc A, Jagannathan S, Hosking AL (June 2004) Transactional monitors for concurrent objects. In: *Proceedings of the European conference on object-oriented programming (ECOOP'04)*, pp 519–542
70. Yu Y, Rodeheffer T, Chen W (October 2003) RaceTrack: Efficient detection of data race conditions via adaptive tracking. In: *Proceedings of the symposium on operating systems principles (SOSP'03)*, pp 221–234
71. Yu Y, Rodeheffer T, Chen W (2005) Racetrack: efficient detection of data race conditions via adaptive tracking. In: *SOSP'05: Proceedings of the 20th ACM symposium on operating systems principles*. ACM, New York, pp 221–234
72. Zhou P, Teodorescu R, Zhou Y (2007) Hard: hardware-assisted lockset-based race detection. In: *HPCA'07: Proceedings of the 2007 IEEE 13th international symposium on high performance computer architecture*. IEEE Computer Society, Washington, DC, pp 121–132

## Race Detectors for Cilk and Cilk++ Programs

JEREMY T. FINEMAN<sup>1</sup>, CHARLES E. LEISERSON<sup>2</sup>

<sup>1</sup>Carnegie Mellon University, Pittsburgh, PA, USA

<sup>2</sup>Massachusetts Institute of Technology, Cambridge, MA, USA

### Synonyms

[Cilkscreen](#); [Nondeterminator](#)

### Definition

The Nondeterminator race detector takes as input an ostensibly deterministic Cilk program and an input data set and makes the following guarantee: it will either determine at least one location in the program that is subject to a determinacy race when the program is run on the data set, or else it will certify that the program always behaves the same on the data set, no matter how it is scheduled. The Cilkscreen race detector does much the same thing for Cilk++ programs. Both can also detect data races, but the guarantee is somewhat weaker.

## Discussion

### Introduction

Many Cilk programs are intended to be deterministic, in that a given program produces the same behavior no matter how it is scheduled. The program may behave nondeterministically, however, if a *determinacy race* occurs: two logically parallel instructions update the same location, where at least one of the two instructions writes the location. In this case, different runs of the program on the same input may produce different behaviors. Race bugs are notoriously hard to detect by normal debugging techniques, such as breakpointing, because they are not easily repeatable. This article describes the *Nondeterminator* and *Cilkscreen* race detectors, which are systems for detecting races in Cilk and Cilk++ programs, respectively.

Determinacy races have been given many different names in the literature. For example, they are sometimes called *access anomalies* [12], *data races* [24], *race conditions* [22], *harmful shared-memory accesses* [32], or *general races* [31]. Emrath and Padua [15] call a deterministic program *internally deterministic* if the program execution on the given input exhibits no determinacy race and *externally deterministic* if the program has determinacy races but its output is deterministic because of the commutative and associative operations performed on the shared locations. The Nondeterminator program checks whether a Cilk program is internally deterministic. Cilkscreen allows for some internal nondeterminism in a Cilk++ program, specifically, nondeterminism encapsulated by “reducer hyperobjects” [18].

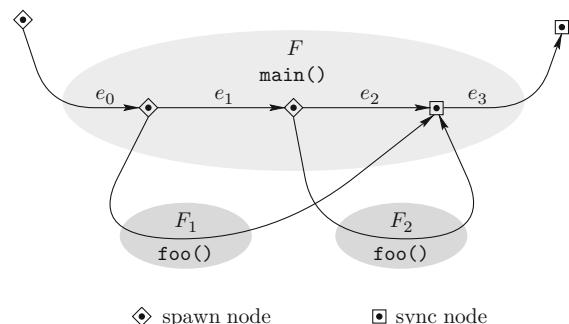
To illustrate how a determinacy race can occur, consider the simple Cilk program shown in Fig. 1. The parallel control flow of this program can be viewed as the directed acyclic graph, or *dag*, illustrated in Fig. 2. The vertices of the dag represent parallel control constructs, and the edges represent *strands*: serial sequences of instructions with no intervening parallel control constructs. In Fig. 2, the strands of the program are labeled to correspond to code fragments from Fig. 1, and the subdags representing the two instances of `foo()` are shaded. In this program, both of the parallel instantiations of the procedure `foo()` update the shared variable `x` in the `x = x + 1` statement. This statement actually causes the processor executing the strand to

```
int x;

cilk void foo()
{
 x = x + 1;
 return;
}

cilk int main() /* F */
{
 x = 0; /* e0 */
 spawn foo(); /* F1 */
 /* e1 */
 spawn foo(); /* F2 */
 /* e2 */
 sync;
 printf("x is %d\n", x); /* e3 */
 return 0;
}
```

**Race Detectors for Cilk and Cilk++ Programs.** Fig. 1 A simple Cilk program that contains a determinacy race. In the comments at the right, the Cilk strands that make up the procedure `main()` are labeled



◊ spawn node      □ sync node

**Race Detectors for Cilk and Cilk++ Programs.** Fig. 2 The parallel control-flow dag of the program in Fig. 1. A spawn node of the dag represents a `spawn` construct, and a sync node represents a `sync` construct. The edges of the dag are labeled to correspond with code fragments from Fig. 1

perform a read from `x`, increment the value, and then write the value back into `x`. Since these operations are not atomic, both might update `x` at the same time. Figure 3 shows how this determinacy race can cause `x` to take on different values if the strands comprising the two instantiations of `foo()` are scheduled simultaneously.

## The Nondeterminator

The Nondeterminator [16] determinacy-race detector takes as input a Cilk program and an input data set and either determines at least one location in the program that is subject to a determinacy race when the program is run on the data set, or else it certifies that the program always behaves the same when run on the data set. If a determinacy race exists, the Nondeterminator localizes the bug, providing variable name, file name, line number, and dynamic context (state of runtime stack, heap, etc.).

The Nondeterminator is not a program verifier, because the Nondeterminator cannot certify that the program is race-free for all input data sets. Rather, it is a debugging tool. The Nondeterminator only checks a program on a particular input data set. What it verifies is that every possible scheduling of the program execution produces the same behavior. If the program relies on any random choices or runtime calls to hardware counters, etc., these values should also be viewed as part of the input data set.

The Nondeterminator is a serial program that operates *on-the-fly*, meaning that it detects races as it simulates the execution of the program, rather than by logging and subsequent analysis. As it executes, it maintains various data structures for determining the existence of determinacy races. An “access history” maintains a subset of strands that access each particular memory location. An “SP-maintenance” data structure maintains the series-parallel (SP) relationships among strands. Specifically, the race detector must determine whether two strands (that access the same memory location) operate logically in parallel (i.e., whether it is possible to schedule both strands at the same time), or whether there is some serial relationship between the strands.

The Nondeterminator was implemented by modifying the ordinary Cilk compiler and runtime system. Each read and write in the user’s program is instrumented by the Nondeterminator’s compiler to perform determinacy-race checking at runtime. The Nondeterminator then takes advantage of the fact that any Cilk program can be executed as a C program. Specifically, if the Cilk keywords are deleted from a Cilk program, a C program results, called the Cilk program’s *serialization* (also called *serial elision* in the literature), whose semantics are a legal implementation of the semantics of the Cilk program [19]. The Nondeterminator executes the user’s program as the serialization would execute, but it performs race-checking actions when reads, writes, and parallel control statements occur.

Since the Nondeterminator was implemented by modifying the Cilk compiler, it is unable to detect races that occur in precompiled library code. In contrast, the Cilk++ race detector, called Cilkscreen, uses binary instrumentation [5, 23, 27]. Cilkscreen dynamically replaces memory accesses in a Cilk++ program binary with instrumented memory accesses, and thus it can operate on executable binaries for which no source code is available. Cilkscreen employs much the same algorithmic technology as the Nondeterminator, however.

## Detecting Races in Programs that Use Locks

The original Nondeterminator [16] is designed to detect determinacy races in a Cilk program. Typically a programmer may add locks to the program to protect against the externally nondeterministic interleaving given in Fig. 3, thus intending and sanctioning nondeterministic executions. Thus, it may not make sense to test the program for determinacy races, but some weaker form of race detection may be desired.

| Case 1        |               |             | Case 2        |              |             |
|---------------|---------------|-------------|---------------|--------------|-------------|
| $F_1$         | $F_2$         | $e_3$       | $F_1$         | $F_2$        | $e_3$       |
| read $x = 0$  |               |             | read $x = 0$  |              |             |
| write $x = 1$ |               |             | read $x = 0$  | read $x = 0$ |             |
|               | read $x = 1$  |             | write $x = 1$ |              |             |
|               | write $x = 2$ |             | write $x = 1$ |              |             |
|               |               | “ $x$ is 2” |               |              | “ $x$ is 1” |

Race Detectors for Cilk and Cilk++ Programs. Fig. 3 An illustration of a determinacy race in the code from Fig. 1.

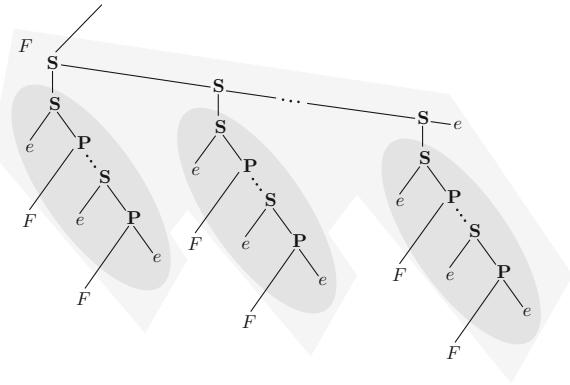
The value of the shared variable  $x$  read and printed by strands  $e_3$  can differ depending on how the instructions in the two instances  $F_1$  and  $F_2$  of the `foo()` procedure are scheduled

The Nondeterminator-2 [7] detects *data races*, which occur when two logically parallel strands holding no locks in common update the same location, where at least one of the two instructions modifies the location. Although the Nondeterminator and Nondeterminator-2 test for different types of races, a significant chunk of the implementations and algorithms used (notably, the SP-maintenance algorithm) remains the same in both debugging tools. This article focuses on determinacy-race detection.

## Series-Parallel Parse Trees

The structure of a Cilk program can be interpreted as a *series-parallel (SP) parse tree* rather than a series-parallel dag. Figure 5 shows a parse tree for the dag in Fig. 2. In the SP parse tree, each internal node is either an *S-node*, denoted by S, or a *P-node*, denoted by P, and each leaf is a strand of the dag. If two subtrees are children of the same S-node, then the strands in the left subtree *logically precede* the strands in the right subtree, and (the subcomputation represented by) the left subtree must execute before (that of) the right subtree. If two subtrees are children of the same P-node, then the strands in the left subtree operate *logically in parallel* with those in the right subtree, and no ordering holds between (the subcomputations represented by) the two subtrees. For two strands  $e$  and  $e'$ , the notation  $e \parallel e'$  means that  $e$  and  $e'$  operate logically in parallel, and the notation  $e < e'$  means that  $e$  logically precedes  $e'$ .

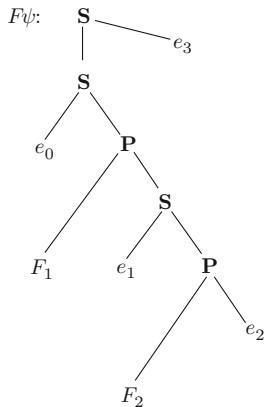
A canonical SP parse tree for a Cilk dag, shown in Fig. 4, can be constructed as follows: First, build a parse tree recursively for each child of the root procedure. Each sync block of the root procedure contains a series of spawns with strands of intervening C code (some of which may be empty). Then, create a parse tree for the sync block alternately applying series and parallel composition to the child parse trees and the root strands. Finally, string the parse trees for the sync blocks together into a *spine* for the procedure by applying a sequence of series compositions to the sync blocks. Sync blocks are composed serially, because a sync statement is never passed until *all* previously spawned subprocedures have completed. The only ambiguities that might arise in the parse tree occur because of the associativity of series composition and the commutativity of parallel composition. If, as shown in Fig. 4, the alternating S-nodes and P-nodes in a sync block always place



**Race Detectors for Cilk and Cilk++ Programs.** Fig. 4 The canonical series-parallel parse tree for a generic Cilk procedure. The notation  $F$  represents the SP parse tree of any subprocedure spawned by this procedure, and  $e$  represents any strand of the procedure. All nodes in the shaded areas belong to the procedure, and the nodes in each oval belong to the same sync block. A sequence of S-nodes forms the spine of the SP parse tree, composing all sync blocks in series. Each sync block contains an alternating sequence of S-nodes and P-nodes. Observe that the left child of an S-node in a sync block is always a strand, and that the left child of a P-node is always a subprocedure

strands and subprocedures on the left, and the series compositions of the sync blocks are applied in order from last to first, then the parse tree is unique. Figure 5 shows such a *canonical* parse tree for the Cilk dag in Fig. 2.

One convenient feature of the SP parse tree is that the logical relationship between strands can be determined by looking at the least common ancestor (lca) of the strands in the parse tree. For example, consider the parse tree in Fig. 5. The least common ancestor of strands  $e_0$  and  $e_3$ , denoted by  $\text{lca}(e_0, e_3)$ , is the root S-node, and hence  $e_0 < e_3$ . In contrast,  $\text{lca}(F_1, F_2)$  is a P-node, and hence  $F_1 \parallel F_2$ . To see that the dag and the parse tree represent the same control structure requires only observing that  $\text{lca}(e_i, e_j)$  is an S-node in the parse tree if and only if there is a directed path from  $e_i$  to  $e_j$  in the corresponding dag. A proof of this fact can be found in [16].



Race Detectors for Cilk and Cilk++ Programs. Fig. 5 The canonical series-parallel parse tree for the Cilk dag in Fig. 2

An execution of a Cilk program can be interpreted as a walk or traversal of the corresponding SP parse tree. The order in which nodes are traversed depends on the scheduler. A partial execution must obey series-parallel relationships, namely, that the tree walk cannot enter the right subtree of an S-node until the left subtree has been fully executed. Both subtrees of a P-node, however, can be traversed in arbitrary order or in parallel. The canonical parse tree is such that an ordinary, left-to-right, depth-first tree walk visits strands in the same order as the program's serialization visits them.

### Access History

The Nondeterminator and Cilkscreen race detectors execute a Cilk program in serial, depth-first order while maintaining two data structures. The SP-maintenance data structure, described later in this article, is used to query the logical relationships among strands. The access history maintains information as to which strands have accessed which memory locations.

At a high level, an *access history* maintains for each shared-memory location two sets of strands that have read from and written to the location. As the Nondeterminator executes, strands are added to and removed from the access history. Whenever an access of memory location  $v$  occurs, each of the strands in  $v$ 's access history are compared against the currently executing strand. If any of these strands operates logically in parallel with the currently executing strand, and one of the accesses is a write, then a race is reported. This query of logical relationships is determined by the SP-maintenance data

```

write a shared location ℓ by strand e :
 if $reader[\ell] \parallel e$ or $writer[\ell] \parallel e$
 then a determinacy race exists
 $writer[\ell] \leftarrow e$

read a shared memory location ℓ by strand e :
 if $writer[\ell] \parallel e$
 then a determinacy race exists
 if $reader[\ell] \prec e$
 then $reader[\ell] \leftarrow e$

```

### Race Detectors for Cilk and Cilk++ Programs. Fig. 6

Pseudocode describing the implementation of read and write operations for a determinacy-race detector. The access history is updated on these operations

structure described in section “Series-Parallel Maintenance.” The main goal when designing an access history is to reduce the number of strands stored; the larger the access history the more series-parallel queries need to be performed.

For a serial determinacy race detector, an access history of size  $O(1)$  per memory location suffices. In particular, the access history associates with each memory location  $\ell$  two values  $reader[\ell]$  and  $writer[\ell]$ , each storing a single strand that has previously read from or written to  $\ell$ , respectively. Specifically,  $writer[\ell]$  stores a unique runtime ID of the strand that has most recently written to  $\ell$ . Similarly,  $reader[\ell]$  stores the ID of some previous reader of  $\ell$ , although it need not be the most recent reader.

The Nondeterminator updates the access history as new memory accesses occur. Each read and write operation of the original program is instrumented to update the access history and discover determinacy races. Figure 6 gives pseudocode describing the instrumentation of read and write operations. A race occurs if a strand  $e$  writes a location  $\ell$  and discovers that either the previous reader or the previous writer of  $\ell$  operates logically in parallel with  $e$ . Similarly, a race occurs whenever  $e$  reads a location  $\ell$  and discovers that the previous writer operates logically in parallel with  $e$ . Whenever a location  $\ell$  is written, the access history  $writer[\ell]$  is updated to be the current strand  $e$ . The read history  $reader[\ell]$  is updated to  $e$  when a read occurs, but only if the previous reader operates logically in series with  $e$ .

The cost of maintaining the access history is  $O(1)$  plus the cost of  $O(1)$  series-parallel queries, for each memory access. A correctness proof of the access history can be found in [16]. The proof involves showing that an update (or lack of an update) to  $\text{reader}[\ell]$  does not discard any important information. Specifically, consider three strands  $e_1$ ,  $e_2$ , and  $e_3$  that occur in that order in the serial execution, where  $e_1$  is the current value of  $\text{reader}[\ell]$ ,  $e_2$  is the currently executing strand reading  $\ell$ , and  $e_3$  is some future strand. If  $e_1 \prec e_2$  and  $e_1 \parallel e_3$ , then  $e_2 \parallel e_3$ . Thus, a test for a conflict between  $e_2$  and  $e_3$  produces the same result as a test for a conflict between  $e_1$  and  $e_3$ . On the other hand, if  $e_1 \parallel e_2$  and  $e_2 \parallel e_3$ , then  $e_1 \parallel e_3$ , and so keeping  $e_1$  gives at least as much information as  $e_2$ . Since updates to  $\text{reader}[\ell]$  do not lose information, the access history properly detects a race whenever the current writer operates logically in parallel with any previous reader. The full proof considers the other two race cases separately (the current writer operates logically in parallel with a previous writer, or the current reader operates logically in parallel with a previous writer), showing that either a race is reported on the access or an earlier writer-writer race was reported.

### Data-Race Detection

In contrast to the Nondeterminator, which detects determinacy races, the Nondeterminator-2 [7] and Cilkscreen can also detect data races. The difference between the Nondeterminator and these other two debugging programs lies in how they deal with the access history. The Nondeterminator-2 and Cilkscreen tools report data races for locking protocols in which every lock is acquired and released within a single strand and cannot be held across parallel control constructs. Strands may hold multiple locks at one time, however. These tools do not detect deadlocks, but only whether during a serial left-to-right depth-first execution, two logically parallel strands holding no locks in common access a shared variable, where at least one of the strands modifies the location.

The *lock set* of an access (`read` or `write`) is the set of locks held by the strand performing the access when the access occurs. If the lock sets of two parallel accesses to the same location have an empty intersection, and at least one of the accesses is a `write`, then a data race exists. To simplify the race-detection algorithm, a small trick avoids the extra condition that “at least one of the

```

ACCESS(ℓ) in strand e with lock set H :
 for each $\langle e', H' \rangle \in \text{lockers}[\ell]$
 do if $e' \parallel e$ and $H' \cap H = \emptyset$
 then a data race exists
 redundant \leftarrow FALSE
 for each $\langle e', H' \rangle \in \text{lockers}[\ell]$
 do if $e' \prec e$ and $H' \supseteq H$
 then $\text{lockers}[\ell] \leftarrow \text{lockers}[\ell] - \{\langle e', H' \rangle\}$
 if $e' \parallel e$ and $H' \subseteq H$
 then redundant \leftarrow TRUE
 if redundant = FALSE
 then $\text{lockers}[\ell] \leftarrow \text{lockers}[\ell] \cup \{\langle e, H \rangle\}$

```

### Race Detectors for Cilk and Cilk++ Programs. Fig. 7

Pseudocode describing the implementation of read and write operations for a data-race detector. The access history is updated on these operations using the lock-set algorithm

accesses is a `write`.” The idea is to introduce a *fake lock* for read accesses called the `R-LOCK`, which is implicitly acquired immediately before a `read` and released immediately afterward. The `R-LOCK` behaves from the race detector’s point of view just like a normal lock, but during an actual computation, it is never actually acquired and released (since it does not actually exist). The use of `R-LOCK` allows the condition for a data race to be stated more succinctly: *If the lock sets of two parallel accesses to the same location have an empty intersection, then a data race exists.* By this condition, a data race (correctly) does not exist for two read accesses, since their lock sets both contain the `R-LOCK`.

The access history for the algorithm for detecting data races records a set of readers and writers for each memory location and their lock sets. For a given location  $\ell$ , the entry  $\text{lockers}[\ell]$  stores a list of *lockers*: strands that access  $\ell$ , each paired with the lock set that was held during the access. If  $\langle e, H \rangle \in \text{lockers}[\ell]$ , then location  $\ell$  is accessed by strand  $e$  while it holds the lock set  $H$ . Figure 7 gives pseudocode for the data-race-detection algorithm employed by the Nondeterminator-2 and Cilkscreen.

This algorithm for data-race detection does not provide as strong a guarantee as the algorithm for determinacy-race detection. Programming with locks introduces intentional nondeterminism which may not be exposed during a serial left-to-right depth-first execution. Thus, although the algorithm always finds a data race if one is exposed, some data races may not

be exposed. Nevertheless, for *abelian* programs, where critical sections protected by the same lock “commute” – produce the same effect on memory no matter in which order they are executed – a guarantee can be provided. For a computation generated by a deadlock-free abelian program running on a given input, the algorithm guarantees to find a data race if one exists, and otherwise guarantee that all executions produce the same final result. A proof of the correctness of this assertion can be found in [7].

The cost of querying the access history for data-race detection is much more than the  $O(1)$  cost for determinacy-race detection. There are at most  $n^k$  entries for each memory location, where  $n$  is the number of locks in the program and  $k \leq n$  is the number of locks held simultaneously. Thus, the running time of the data-race detector may increase proportionally to  $n^k$  in the worst case. In practice, however, few locks are held simultaneously, and the running time tends to be only slightly worse than for determinacy-race detection.

## Series-Parallel Maintenance

The main structural component of the two Nondeterminator programs and Cilkscreen is their algorithm for maintaining series-parallel (SP) relationships among strands. These race detectors execute the program in a left-to-right depth-first order while maintaining the logical relationships between strands. As new strands are encountered, the SP-maintenance data structure is updated. Whenever the currently executing strand accesses a memory location, the Nondeterminator queries the SP-maintenance data structure to determine whether the current strand operates in series or in parallel with strands (those in the access history) that made earlier accesses to the same memory location.

Figure 8 compares the serial space and running times of several SP-maintenance algorithms. The SP-bags and SP-order algorithms, described in this section, are employed by various implementations of the Nondeterminator. The English-Hebrew [32] and Offset-Span [24] algorithms are used in different race detectors and displayed for comparison. The modified SP-bags entry reflects a different underlying data structure than the SP-bags entry, but the algorithm for both is the same. As the number of queries performed by a determinacy race detector can be as large as  $O(T)$  for a program that runs serially in  $T$  time, keeping the query time small

| Algorithm             | Space per node | Time per                |                         |
|-----------------------|----------------|-------------------------|-------------------------|
|                       |                | Strand creation         | Query                   |
| English-Hebrew [32]   | $\Theta(f)$    | $\Theta(1)$             | $\Theta(f)$             |
| Offset-Span [24]      | $\Theta(d)$    | $\Theta(1)$             | $\Theta(d)$             |
| SP-bags [16]          | $\Theta(1)$    | $\Theta(\alpha(v,v))^*$ | $\Theta(\alpha(v,v))^*$ |
| Modified SP-bags [17] | $\Theta(1)$    | $\Theta(1)^*$           | $\Theta(1)$             |
| SP-order [4]          | $\Theta(1)$    | $\Theta(1)$             | $\Theta(1)$             |

$f$  = number of forks/spawns in the program  
 $d$  = maximum depth of nested parallelism  
 $v$  = number of shared locations being monitored

## Race Detectors for Cilk and Cilk++ Programs. Fig. 8

Comparison of serial, SP-maintenance algorithms. An asterisk (\*) indicates an amortized bound. The function  $\alpha$  is Tarjan’s functional inverse of Ackermann’s function

is desirable, and hence SP-bags and SP-order are more appealing algorithms for serial race detectors. The situation is exacerbated in a data-race detector where the number of queries can be larger than  $T$ .

## SP-Bags

Since the logical relationship between strands can be determined by looking at their least common ancestor in the SP parse tree, a straightforward approach for SP-maintenance would maintain this tree explicitly. Querying the relationship between strands can then be implemented naively by climbing the tree starting from each strand until converging at their least common ancestor. This approach yields expensive queries, as the worst-case cost is proportional to the height of the tree, which is not bounded.

SP-bags [16] is indeed based on finding the least common ancestor, but it uses a clever algorithm that yields far better queries. The algorithm itself is an adaptation of Tarjan’s offline least-common-ancestors algorithm.

The SP-bags algorithm maintains a collection of disjoint sets, employing a “disjoint sets” or “union-find” data structure that supports the following operations:

1.  $\text{MAKE-SET}(x)$  creates a new set whose only member is  $x$ .
2.  $\text{UNION}(x,y)$  unites the sets containing  $x$  and  $y$ .
3.  $\text{FIND}(x)$  returns a representative for the set containing  $x$ .

A classical disjoint-set data structure with “union by rank” and “path compression” heuristics [9, 34, 35] supports  $m$  operations on  $n$  elements in  $O(m\alpha(m, n))$  time, where  $\alpha$  is Tarjan’s functional inverse of Ackermann’s function [9, 35].

As the program executes, for each active procedure, the SP-bags algorithm maintains two *bags* (unordered sets) with the following contents at any given time:

- The *S-bag*  $S_F$  of a procedure  $F$  contains the descendant procedures of  $F$  that logically precede the currently executing strand. (The descendant procedures of  $F$  include  $F$  itself.)
- The *P-bag*  $P_F$  of a procedure  $F$  contains the completed descendant procedures of  $F$  that operate logically in parallel with the currently executing strand.

The S- and P-bags are represented using a disjoint sets data structure as described above.

The SP-bags algorithm is given in Fig. 9. As the Cilk program executes in a serial, depth-first fashion, the SP-bags algorithm performs additional operations whenever one of the three following actions occurs: *spawn*, *sync*, *return*, resulting in updates to the S- and P-bags. Whenever a new procedure  $F$  (entering the left subtree of a P-node) is spawned, new bags are created. The bag  $S_F$  is initially set to contain  $F$ , and  $P_F$  is set to be empty. Whenever a procedure  $F'$  returns to its parent procedure  $F$  (completing the walk of the left subtree of a P-node), the contents of  $S_{F'}$  are unioned

```

spawn a procedure F:
 $S_F \leftarrow \text{MAKE-SET}(F)$
 $P_F \leftarrow \emptyset$

return from a procedure F' to parent F :
 $P_F \leftarrow \text{UNION}(P_F, S_{F'})$
 $S_{F'} \leftarrow \emptyset$

sync in a procedure F :
 $S_F \leftarrow \text{UNION}(S_F, P_F)$
 $P_F \leftarrow \emptyset$

```

**Race Detectors for Cilk and Cilk++ Programs.** Fig. 9 The SP-bags algorithm. Whenever one of three actions occurs during the serial, depth-first execution of a Cilk parse tree, the operations in the figure are performed. These operations cause SP-bags to manipulate the disjoint sets data structure

into  $P_F$ , since the descendants of  $F'$  can execute in parallel with the remainder of the sync block in  $F$ . When a *sync* occurs (traversing a spine node in the parse tree) in  $F$ , the bag  $P_F$  is emptied into  $S_F$ , since all of  $F$ ’s executed descendants precede any future strands in  $F$ .

As a concrete example, consider the program given in Fig. 5. The execution order of strands and procedures is  $e_0, F_1, e_1, F_2, e_2, e_3$ . Figure 10 shows the state of the S- and P-bags during each step of the execution. As the S- and P-bags are only specified for active procedures, the number of bags changes during the execution.

To determine whether a previously executed procedure (or strand)  $F$  is logically in series or in parallel with the currently executing strand, simply check whether  $F$  belongs to an S-bag by looking at the identity of *FIND-SET*( $F$ ).

Figure 10 illustrates the correctness of SP-bags for the program Fig. 1. For example, when executing  $F_2$ , all previously executed instructions in  $F$  (namely,  $e_0$  and  $e_1$ ) serially precede  $F_2$ , and  $F$  does indeed belong to an S-bag  $S_F$ . Moreover, the procedure  $F_1$  operates logically in parallel with  $F_2$ , and  $F_2$  belongs to the P-bag  $P_F$ . For a proof of correctness, which is omitted here, refer to [16].

Combining SP-bags with the access history yields an efficient determinacy race detector. Consider a Cilk program that executes in time  $T$  on one processor and references  $v$  shared memory locations. The SP-bags algorithm can be implemented to check this program for determinacy races in  $O(T\alpha(v, v))$  time.

A proof of this theorem, which is provided in [16], is conceptually straightforward. The number of *MAKE-SET*, *UNION*, and *FIND-SET* operations is at most  $O(T)$ , yielding a total running time of  $O(T\alpha(m, n))$  for some value of  $m$  and  $n$ . It turns out that these values can be reduced to  $v$  when garbage collection is employed.

The theoretical running time of SP-bags can be improved by replacing the underlying disjoint sets data structure. Since the *UNIONS* are structured nicely, Gabow and Tarjan’s data structure [20], which features  $O(1)$  amortized time per operation, can be employed. The SP-bags algorithm can be implemented to check a program for determinacy races in  $O(T)$  time, where the program executes in time  $T$  on one processor. A full discussion of this improvement appears in [17].

| Execution step | State of the Bags       |                      |                     |                       |
|----------------|-------------------------|----------------------|---------------------|-----------------------|
| $e_0$          | $S_F = \{F\}$           | $P_F = \emptyset$    |                     |                       |
| $F_1$          | $S_F = \{F\}$           | $P_F = \emptyset$    | $S_{F_1} = \{F_1\}$ | $P_{F_1} = \emptyset$ |
| $e_1$          | $S_F = \{F\}$           | $P_F = \{F_1\}$      |                     |                       |
| $F_2$          | $S_F = \{F\}$           | $P_F = \{F_1\}$      | $S_{F_2} = \{F_2\}$ | $P_{F_2} = \emptyset$ |
| $e_2$          | $S_F = \{F\}$           | $P_F = \{F_1, F_2\}$ |                     |                       |
| $e_3$          | $S_F = \{F, F_1, F_2\}$ | $P_F = \emptyset$    |                     |                       |

Race Detectors for Cilk and Cilk++ Programs. Fig. 10 The state of the SP-bags algorithm when run on the parse tree from Fig. 5

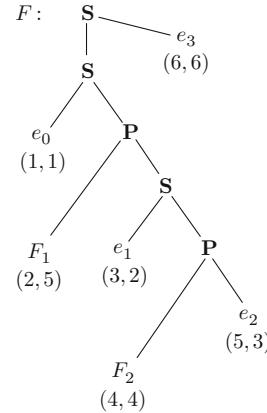
### SP-Order

SP-order uses two total orders to determine whether strands are logically parallel, an *English order* and a *Hebrew order*. In the English order, the nodes in the *left* subtree of a P-node precede those in the *right* subtree of the P-node. In the Hebrew order, the order is reversed: the nodes in the *right* subtree of a P-node precede those in the *left*. In both orders, the nodes in the left subtree of an S-node precede those in the right subtree of the S-node.

Figure 11 shows English and Hebrew orderings for the strands in the parse tree from Fig. 5. Notice that if  $x$  belongs in the left subtree of an S-node and  $y$  belongs to the right subtree of the same S-node, then  $E[x] < E[y]$  and  $H[x] < H[y]$ . In contrast, if  $x$  belongs to the left subtree of a P-node  $y$  belongs to the right subtree of the same P-node, then  $E[x] < E[y]$  and  $H[x] > H[y]$ .

The English and Hebrew orderings capture the SP relationships in the parse tree. Specifically, if one strand  $x$  precedes another strand  $y$  in both orders, then  $x < y$ , since  $\text{lca}(x, y)$  is an S-node, and hence there is a directed path from  $x$  to  $y$  in the dag. If  $x$  precedes  $y$  in one order but  $x$  follows  $y$  in the other, then  $x \parallel y$ , since  $\text{lca}(x, y)$  is a P-node, and hence there is no directed path from one to the other in the dag. For example, in Fig. 11, the fact  $e_0$  precedes  $e_2$  can be determined by observing  $E[e_0] < E[e_2]$  and  $H[e_0] < H[e_2]$ . Similarly, it follows that  $F_1$  and  $F_2$  are logically parallel since  $E[F_1] < E[F_2]$  and  $H[F_1] > H[F_2]$ . The following lemma, proved in [4], shows that this property always holds.

**Lemma 1** Let  $E$  be an English ordering of strands of an SP-parse tree, and let  $H$  be a Hebrew ordering. Then, for any two strands  $x$  and  $y$  in the parse tree,  $x < y$  if and only if  $E[x] < E[y]$  and  $H[x] < H[y]$ . Equivalently,  $x \parallel y$  if and only if  $E[x] < E[y]$  and  $H[x] > H[y]$ , or if  $E[x] > E[y]$  and  $H[x] < H[y]$ .



Race Detectors for Cilk and Cilk++ Programs. Fig. 11 An English ordering  $E$  and a Hebrew ordering  $H$  for strands in the parse tree of Fig. 5. Under each strand/procedure  $e$  is an ordered pair  $(E[e], H[e])$  giving its rank in each of the two orders. Equivalently, the English order is  $e_0, F_1, e_1, F_2, e_2, e_3$  and the Hebrew order is  $e_0, e_1, e_2, F_2, F_1, e_3$

Labeling a static SP parse tree with an English-Hebrew ordering is straightforward. To compute the English ordering, perform a depth-first traversal visiting left children of both P-nodes and S-nodes before visiting right children (an *English walk*). Assign label  $i$  to the  $i$ th strand visited. To compute the Hebrew ordering, perform a depth-first traversal visiting right children of P-nodes before visiting left children but left children of S-nodes before visiting right children (a *Hebrew walk*). Assign labels to strands as before, in the order visited.

In race-detection applications, however, these orderings must be generated on-the-fly before the entire parse tree is known. If the parse tree unfolds according to an English walk, as in the Nondeterminator, then computing the English ordering is trivial. Unfortunately, computing the Hebrew ordering during an English walk is problematic. In a Hebrew ordering, the

label of a strand in the left subtree of a P-node depends on the number of strands in the right subtree. This number is unknown, while performing an English walk, until the right subtree has unfolded completely.

Nudler and Rudolph [32], who introduced English-Hebrew labeling for race detection, addressed this problem by using large static strand labels. In particular, the number of bits in a label in their scheme can grow linearly in the number of P-nodes in the SP parse tree. Although they gave a heuristic for reducing the size of labels, manipulating large labels is the performance bottleneck of their algorithm.

The solution in SP-order is to employ order-maintenance data structures [3, 10, 11, 37] to maintain the English and Hebrew orders dynamically rather than using the static labels described above. An order-maintenance data structure is an abstract data type that supports the following operations:

- OM-PRECEDES( $L, x, y$ ): Return TRUE if  $x$  precedes  $y$  in the ordering  $L$ . Both  $x$  and  $y$  must already exist in the ordering  $L$ .
- OM-INSERT( $L, x, y_1, y_2, \dots, y_k$ ): In the ordering  $L$ , insert new elements  $y_1, y_2, \dots, y_k$ , in that order, immediately after the existing element  $x$ . Any previously existing elements that followed  $x$  now follow  $y_k$ .

The OM-PRECEDES operation can be supported in  $O(1)$  worst-case time. The OM-INSERT operation can be supported in  $O(1)$  worst-case time for each node inserted. In the efficient order-maintenance data structures, labels are assigned to each element of the data structure, and the relative ordering of two elements (OM-PRECEDES) is determined by comparing these labels. Keeping the labels small guarantees good query cost. The labels change dynamically as insertions occur.

The SP-order data structure consists of two order-maintenance data structures to maintain English and Hebrew orderings. (In fact, the English ordering can be maintained implicitly during a left-to-right tree walk. For conceptual simplicity, however, both orderings are represented with the data structure here.) With the data structure chosen, the implementation of SP-order is remarkably simple. When first traversing an S-node  $S$  with left child  $x$  and right child  $y$ , perform OM-INSERT( $Eng, S, x, y$ ) to insert  $x$  then  $y$  after  $S$  in the English ordering  $Eng$ . Also perform

OM-INSERT( $Heb, S, x, y$ ) to insert these children in the same order after  $S$  in the Hebrew ordering  $Heb$ . When first traversing a P-node  $P$  with left child  $x$  and right child  $y$ , also perform OM-INSERT( $Eng, P, x, y$ ) as before, but perform OM-INSERT( $Heb, P, y, x$ ) to insert these children in the opposite order in the Hebrew ordering.

To determine whether a previously executed strand  $e'$  is logically in series or in parallel with the currently executing strand  $e$ , simply check whether OM-PRECEDES( $Heb, e', e$ ). (It is always the case that OM-PRECEDES( $Eng, e', e$ ), as the program executes in the order of an English walk.) If OM-PRECEDES( $Heb, e', e$ ), then  $e' < e$ . If not, then  $e' \parallel e$ .

While the preceding description of SP-order with respect to the parse tree fully describes the algorithm, it is somewhat opaque with respect to the effect on Cilk code and the Cilk compiler. Figure 12, which is analogous to Fig. 9 for the SP-bags algorithm, describes the SP-order algorithm with respect to Cilk keywords. The main point of this figure is to show that, like SP-bags, incorporating SP-order into a Cilk program is fairly lightweight. In contrast to SP-bags, whose data structural elements are procedures, SP-order builds data structures over strands. As such, new objects or IDs are created whenever new strands are encountered. On a

spawn a child procedure  $F'$  of  $F$ :

create strand IDs for:

$e_0$ : the first strand in  $F'$

$e_c$ : the continuation strand in  $F$

OM-INSERT( $Eng, current[F], e_0, e_C$ )

OM-INSERT( $Heb, current[F], e_c, e_0$ )

return from a procedure  $F'$  to parent  $F$ :

nothing

following a spawn of procedure  $F$  or a sync in  $F$ :

create strand ID for:

$e_S$ : the strand following the next sync in  $F$

OM-INSERT( $Eng, current[F], e_S$ )

OM-INSERT( $Heb, current[F], e_S$ )

**Race Detectors for Cilk and Cilk++ Programs. Fig. 12** The SP-order algorithm. Whenever a spawn, sync, or return occurs during the serial, depth-first execution of the Cilk parse tree, the operations in the figure are performed. These operations cause SP-order to manipulate the order maintenance data structures. The value  $current[F]$  denotes the currently executing strand in procedure  $F$

spawn, strands are created for both the spawned procedure and the continuation strand within the parent procedure. These strands are then inserted into *Eng* and *Heb* in opposite orders, as they operate logically in parallel. Whenever beginning to execute a sync block in procedure *F*, either after spawning *F* or sync’ing within *F*, a new strand  $e_S$  is created to represent the start of the next sync block. This  $e_S$  is then inserted after the currently executing strand in both *Eng* and *Heb*, as it operates logically after the currently executing strand. Unlike SP-bags, no data structural changes occur in SP-order when return’ing from a procedure.

One interesting feature of SP-order is that elements (strands) are only added to the data structure. No elements are ever reordered once added. Unreferenced elements may be removed as part of garbage collection. The (incomplete) English and Hebrew orderings at any time are thus consistent with the *a posteriori* orderings, with not-yet-encountered strands removed. The correctness of SP-order, the proof of which is given in [4], is arguably easier to convince oneself of than the proof of SP-bags.

## Making the Nondeterminator Run in Parallel

The two Nondeterminators and Cilkscreen are serial race detectors. Even though these are debugging tools intended for parallel programs, they themselves run serially. This section discusses how to make these race detectors run in parallel. Both the access history and SP-maintenance algorithms must be augmented to permit a parallel algorithm. Moreover, contention due to concurrent updates to these data structures needs to be addressed.

### A Parallel Access History

The access history described in section “Access History” is appealing because it requires only a single reader and writer for a determinacy-race detector, thus keeping the cost of maintaining and querying against the access history low. This strategy, however, is inherently serial. If only a single reader and writer are recorded, a parallel race detector may not discover certain races in the program [17]. Fortunately, Mellor-Crummey [24] shows that recording two readers and writers suffices to guarantee correctness in a parallel execution, which increases the number of SP-maintenance queries by

only a factor of 2. A full description of such a parallel access history and its correctness can be found in [17, 24].

The other challenge in parallelizing the access history is in performing concurrent updates to the data structure. Specifically, two parallel readers may attempt to update the same  $reader[\ell]$  at the same time. (If the goal is to just report a single race in the program, updating  $writer[\ell]$  in parallel is not as much of an issue, as discovering such a parallel update indicates a race on its own. If the goal is to report one race for each memory location, as in the Nondeterminator, then concurrent updates to  $writer[\ell]$  also matter here.) Some extra machinery is required to guarantee the correct value is recorded when concurrent updates occur. As discussed by Fineman [17], concurrent updates by  $P$  processors can be resolved in  $O(t \lg P)$  worst-case time, where  $t$  is the serial cost of an access-history update:  $t = O(1)$  for a determinacy-race detector, and  $t = O(n^k)$  for a data-race detector.

### Parallel SP-Maintenance

The SP-bags algorithm is inherently serial, as the correctness relies on an execution order corresponding to an English walk of the parse tree. Correctness of SP-order, on the other hand, does not rely on any particular execution order. SP-order thus appears like a natural choice for a parallel algorithm. The main complication arises in performing concurrent updates to the underlying order-maintenance data structures of SP-order. The obvious approach for parallelizing SP-order directly requires locking the order-maintenance data structures whenever performing an update. As with the access history, a lock may cause all the other  $P - 1$  processors to waste time waiting for the lock, thus causing a  $\Theta(P)$  overhead per lock acquisition. For programs with short strands, a  $\Theta(P)$  overhead per strand creation results in no improvement as compared to a serial execution.

The SP-ordered-bags algorithm, previously called SP-hybrid [4, 17], uses a two-tiered algorithm with a global tier and a local tier to overcome the scalability problems with lock synchronization. The global tier uses a parallel SP-order algorithm with locks, and the local tier uses the serial SP-bags algorithm without locks. By bounding the number of updates to the global tier, the locking overhead is reduced.

SP-ordered-bags leverages the fact that the Cilk scheduler executes a program mostly in an English order. The only exception occurs on steals when the thief processor causes a violation in the English ordering. The thief then continues executing its local subtree serially in an English order. Thus, any serial SP-maintenance algorithm, like SP-bags, is viable within each of these local serially executing subtrees. (SP-bags is also desirable as it interacts well with the global tier; details are omitted here.) Moreover, each local SP-bags data structure is only updated by the single processor executing that subtree, and thus no locking is required. The local tier permits SP queries within a single local subcomputation.

The global tier is responsible for arranging the local subcomputations to permit SP-queries among these subcomputations. SP-ordered-bags employs the parallel SP-order algorithm to arrange the subcomputations. Although the global tier is locked whenever an update occurs, updates occur only when the Cilk scheduler causes a steal. Since the Cilk scheduler causes at most  $O(PT_\infty)$  steals, where  $T_\infty$  is the span of the program being executed, updates to the global tier add only  $O(PT_\infty)$  to the running time of the program.

Combining the local and global tiers together into SP-ordered-bags yields a parallel SP-maintenance algorithm that runs in  $O(T_1/P + PT_\infty)$  time on  $P$  processors, where  $T_1$  and  $T_\infty$  are the work and span of the underlying program being tested. A detailed discussion of SP-ordered-bags, including correctness and performance proofs, can be found in [4, 17].

## Related Entries

- ▶ [Cilk](#)
- ▶ [Race Conditions](#)
- ▶ [Race Detection Techniques](#)

## Bibliographic Notes and Further Reading

Static race detectors [1, 2, 6, 15, 25, 36] analyze the text of the program to determine whether a race occurs. Static analysis tools may be able to determine whether a memory location can be involved in a race for any input. These tools are inherently conservative, however, sometimes reporting races that do not exist, since the static debuggers cannot fully understand the control-flow and synchronization semantics of a program. For example,

dynamic control-flow constructs (e.g., a fork statement) inside `if` statements or loops are particularly difficult to deal with. Mellor-Crummey [25] proposes using static tools as a way of pruning the number of memory locations being monitored by a dynamic race detector.

Dynamic race detectors execute the program given a particular input. Some dynamic race detectors perform a post-mortem analysis based on program-execution logs [8, 14, 21, 26, 28–30], analyzing a log of program-execution events after the program has finished running. On-the-fly race detectors, like the Nondeterminators and Cilkscreen, report races during the execution of the program. Both dynamic approaches are similar and use some form of SP-maintenance algorithm in conjunction with an access history. On-the-fly race detectors benefit from garbage collection, thereby reducing the total space used by the tool. Post-mortem tools, on the other hand, must keep exhaustive logs.

Netzer and Miller [31] provide a common terminology to unify previous work on dynamic race detection. A *feasible* data race is a race that can actually occur in an execution of the program. Netzer and Miller show that locating feasible data races in a general program is NP-hard. Instead, most race detectors, including the ones in this article, deal with the problem of discovering *apparent* data races, which is an approximation of the races that may actually occur. These race detectors typically ignore data dependencies that may make some apparent races infeasible, instead considering only explicit coordination or control-flow constructs (like forks and joins). As a result, these race detectors are conservative and report races that may not actually occur.

Dinning and Schonberg’s “lock-covers” algorithm [13] detects apparent races in programs that use locks. Cheng et al. [7] generalize this algorithm and improve its running time with their All-Sets algorithm, which is the one used in the Nondeterminator-2 and Cilkscreen.

Savage et al. [33] give an on-the-fly race detector called Eraser that does not use an SP-maintenance algorithm, and hence would report races between strands that operate in series if applied to Cilk. Their Eraser tool works on programs that have static threads (i.e., no nested parallelism) and enforces a simple locking discipline. A shared variable must be protected by a particular lock on every access, or they report a race. The Breddy

algorithm [7] employs SP maintenance while enforcing an “umbrella” locking discipline, which generalizes Eraser’s locking discipline. Breddy was incorporated into the Nondeterminator-2, but although it theoretically runs faster than the All-Sets algorithm in the worst case, most users preferred the All-Sets algorithm for the completeness of coverage, and the typical running time of All-Sets is not onerous.

Nudler and Rudolph [32] introduced the English-Hebrew labeling scheme for their SP-maintenance algorithm. Each strand is assigned two static labels, similar to the labeling described for SP-order. They do not, however, use a centralized data structure to reassign labels. Instead, label sizes grow proportionally to the maximum concurrency of the program. Mellor-Crummey [24] proposed an “offset-span labeling” scheme, which has label lengths proportional to the maximum nesting depth of forks. Although it uses shorter label lengths than the English-Hebrew scheme, the size of offset-span labels is not bounded by a constant as it is SP-order. Both of these approaches perform local decisions on strand creation to assign static labels. Although these approaches result in no locking or synchronization overhead for SP-maintenance, and are inherently parallel algorithms, the large labels can drastically increase the work of the race detector.

Dinning and Schonberg’s “task recycling” algorithm [12] uses a centralized data structure to maintain series-parallel relationships. Each strand (block) is given a unique task identifier, which consists of a task and a version number. A task can be reassigned (recycled) to another strand during the program execution, which reduces the total amount of space used by the algorithm. Each strand is assigned a parent vector that contains the largest version number, for each task, of its ancestor strands. To query the relationship between an active strand  $e_1$  and a strand  $e_2$  recorded in the access history, task recycling simply compares the version number of  $e_2$ ’s task against the version number stored in the appropriate slot in  $e_1$ ’s parent vector, which is a constant-time operation. The cost of creating a new strand, however, can be proportional to the maximum logical concurrency. Dinning and Schonberg’s algorithm also handles other coordination between strands, like barriers, where two parallel threads must reach a particular point before continuing.

## Bibliography

1. Appelbe WF, McDowell CE (1985) Anomaly reporting: a tool for debugging and developing parallel numerical algorithms. In: Proceedings of the 1st international conference on supercomputing systems. IEEE, pp 386–391
2. Balasundaram V, Kennedy K (1986) Compile-time detection of race conditions in a parallel program. In: Proceedings of the 3rd international conference on supercomputing, ACM Press, New York, pp 175–185
3. Bender MA, Cole R, Demaine ED, Farach-Colton M, Zito J (2002) Two simplified algorithms for maintaining order in a list. In: Proceedings of the European symposium on algorithms, pp 152–164
4. Bender MA, Fineman JT, Gilbert S, Leiserson CE (2004) On-the-fly maintenance of series-parallel relationships in fork-join multi-threaded programs. In: Proceedings of the sixteenth annual ACM symposium on parallel algorithms and architectures, Barcelona, Spain, June 2004, pp 133–144
5. Bruening D (2004) Efficient, transparent, and comprehensive runtime code manipulation. Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology
6. Callahan D, Sublok J (1988) Static analysis of low-level synchronization. In: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on parallel and distributed debugging, ACM Press, New York, pp 100–111
7. Cheng G-I, Feng M, Leiserson CE, Randall KH, Stark AF (1988) Detecting data races in Cilk programs that use locks. In: Proceedings of the ACM symposium on parallel algorithms and architectures, June 1988, pp 298–309
8. Choi J-D, Miller BP, Netzer RHB (1991) Techniques for debugging parallel programs with flowback analysis. ACM Trans Program Lang Syst 13(4):491–530
9. Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to algorithms, 3rd edn. MIT Press, Cambridge
10. Dietz PF (1982) Maintaining order in a linked list. In: Proceedings of the ACM symposium on the theory of computing, May 1982, pp 122–127
11. Dietz PF, Sleator DD (1987) Two algorithms for maintaining order in a list. In: Proceedings of the ACM symposium on the theory of computing, May 1987, pp 365–372
12. Dinning A, Schonberg E (1990) An empirical comparison of monitoring algorithms for access anomaly detection. In: Proceedings of the ACM SIGPLAN symposium on principles and practice of parallel programming, pp 1–10
13. Dinning A, Schonberg E (1991) Detecting access anomalies in programs with critical sections. In: Proceedings of the ACM/ONR workshop on parallel and distributed debugging, May 1991, ACM Press, pp 85–96
14. Emrath PA, Ghosh S, Padua DA (1989) Event synchronization analysis for debugging parallel programs. In: Proceedings of the 1989 ACM/IEEE conference on supercomputing, November 1989, pp 580–588
15. Emrath PA, Padua DA (1988) Automatic detection of nondeterminacy in parallel programs. In: Proceedings of the workshop

- on parallel and distributed debugging, Madison, Wisconsin, May 1988, pp 89–99
16. Feng M, Leiserson CE (1997) Efficient detection of determinacy races in Cilk programs. In: Proceedings of the ACM symposium on parallel algorithms and architectures, June 1997, pp 1–11
  17. Fineman JT (2005) Provably good race detection that runs in parallel. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, August 2005
  18. Frigo M, Halpern P, Leiserson CE, Lewin-Berlin S (2009) Reducers and other cilk++ hyperobjects. In: Proceedings of the twenty-first annual symposium on parallelism in algorithms and architectures, pp 79–90
  19. Frigo M, Leiserson CE, Randall KH (1998) The implementation of the Cilk-5 multithreaded language. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation, pp 212–223
  20. Gabow HN, Tarjan RE (1985) A linear-time algorithm for a special case of disjoint set union. *J Comput System Sci* 30(2): 209–221
  21. Helmbold DP, McDowell CE, Wang J-Z (1990) Analyzing traces with anonymous synchronization. In: Proceedings of the 1990 international conference on parallel processing, August 1990, pp II70–II77
  22. Steele GL Jr (1990) Making asynchronous parallelism safe for the world. In: Proceedings of the seventeenth annual ACM symposium on principles of programming languages, ACM Press, pp 218–231
  23. Luk C-K, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K (2005) Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation, ACM Press, New York, pp 190–200
  24. Mellor-Crummey J (1991) On-the-fly detection of data races for programs with nested fork-join parallelism. In: Proceedings of supercomputing, pp 24–33
  25. Mellor-Crummey J (1993) Compile-time support for efficient data race detection in shared-memory parallel programs. In: Proceedings of the ACM/ONR workshop on parallel and distributed debugging, San Diego, California, May 1993. ACM Press, pp 129–139
  26. Miller BP, Choi J-D (1988) A mechanism for efficient debugging of parallel programs. In: Proceedings of the 1988 ACM SIGPLAN conference on programming language design and implementation, Atlanta, Georgia, June 1988, pp 135–144
  27. Nethercote N, Seward J (2007) Valgrind: A framework for heavyweight dynamic binary instrumentation. In: Proceedings of the ACM SIGPLAN 2007 conference on programming language design and implementation, ACM, San Diego, June 2007, pp 89–100
  28. Netzer RHB, Ghosh S (1992) Efficient race condition detection for shared-memory programs with post/wait synchronization. In: Proceedings of the 1992 international conference on parallel processing, St. Charles, Illinois, August 1992
  29. Netzer RHB, Miller BP (1990) On the complexity of event ordering for shared-memory parallel program executions. In: Proceedings of the 1990 international conference on parallel processing, August 1990, pp II:93–97
  30. Netzer RHB, Miller BP (1991) Improving the accuracy of data race detection. In: Proceedings of the third ACM SIGPLAN symposium on principles and practice of parallel programming, New York, NY, USA. ACM Press, pp 133–144
  31. Netzer RHB, Miller BP (1992) What are race conditions? *ACM Lett Program Lang Syst* 1(1):74–88
  32. Nudler I, Rudolph L (1986) Tools for the efficient development of efficient parallel programs. In: Proceedings of the first Israeli conference on computer systems engineering, May 1986
  33. Savage S, Burrows M, Nelson G, Sobalvarro P, Anderson T (1997) Eraser: a dynamic race detector for multi-threaded programs. In: Proceedings of the sixteenth ACM symposium on operating systems principles (SOSP), ACM Press, New York, pp 27–37
  34. Tarjan RE (1975) Efficiency of a good but not linear set union algorithm. *J ACM* 22(2):215–225
  35. Tarjan RE (1983) Data structures and network algorithms. Society for Industrial and Applied Mathematics, Philadelphia
  36. Taylor RN (1983) A general-purpose algorithm for analyzing concurrent programs. *Commun ACM* 26(5):361–376
  37. Tsakalidis AK (1984) Maintaining order in a generalized linked list. *Acta Inform* 21(1):101–112

## Race Hazard

### ► Race Conditions

## Radix Sort

### ► Sorting

## Rapid Elliptic Solvers

EFSTRATIOS GALLOPOULOS  
University of Patras, Patras, Greece

## Synonyms

Fast poisson solvers

## Definition

Direct numerical methods for the solution of linear systems obtained from the discretization of certain partial

differential equations, typically elliptic and separable, defined on rectangular domains in  $d$  dimensions, with sequential computational complexity  $O(N \log_2 N)$  or less, where  $N$  is the number of unknowns.

## Discussion

Mathematical models in many areas of science and engineering are often described, in part, by elliptic partial differential equations (PDEs), so their fast and reliable numerical solution becomes an essential task. For some frequently occurring elliptic PDEs, it is possible to develop solution methods, termed Rapid Elliptic Solvers (RES), for the linear systems obtained from their discretization that exploit the problem characteristics to achieve (almost linear) complexity  $O(N \log N)$  or less (all logarithms are base 2) for systems of  $N$  unknowns. RES are direct methods, in the sense that in the absence of roundoff they give an exact solution and require about the same storage as iterative methods. When well implemented, RES can solve the problems they are designed for faster than other direct or iterative methods [7, 32]. The downside is their limited applicability; since RES impose restrictions on the PDE, the domain of definition and their performance and ease of implementation may also depend on the boundary conditions and the size of the problem.

Major historical milestones were the paper [27], by Hyman, where Fourier analysis and marching were proposed to solve Poisson's equation as a precursor of methods that were analyzed more than a decade later; and the paper by Bickley and McNamee [3, Sect. 3], where the inverse of the special block tridiagonal matrices occurring when solving Poisson's equation and a first version of the Matrix Decomposition algorithm were described. Reference [22] by Hockney with its description of the FACR(1) algorithm and the solution of tridiagonal systems with cyclic reduction (in collaboration with Golub) is widely considered to mark the beginning of the modern era of RES. That and the paper by Buzbee, Golub, and Nielson [9], with its detailed analysis of the major RES were extremely influential in the development of the field. The evolution to methods of low sequential complexity was enabled by two key developments: the advent of the FFT and fast methods for the manipulation of Toeplitz matrices.

Interest in the design and implementation of parallel algorithms for RES started in the early 1970s, with the

papers of Buzbee [10] and Sameh et al. [45] and has been attracting the attention of computational scientists ever since. These parallel RES can solve the linear systems under consideration in  $O(\log N)$  parallel operations on  $O(N)$  processors instead of the fastest but impractical algorithm for general linear systems that requires  $O(\log^2 N)$  parallel operations on  $O(N^4)$  processors. Parallel implementations were discussed as early as 1973 for the Illiac IV (see ►Illiac IV) [14] and subsequently for most important high-performance computing platforms, including vector processors, vector multiprocessors, shared memory symmetric multiprocessors, distributed memory multiprocessors, SIMD and MIMD processor arrays, clusters of heterogeneous processors, and in Grid environments. References for specific systems can be found for the Cray-1 [49, 51]; the ICL DAP [25]; Alliant FX/8 [18, 31]; Caltech and Intel hypercubes [13, 36, 41, 50]; Thinking Machines CM-2 [35]; Denelcor HEP [8]; the University of Illinois Cedar machine [16, 20]; Cray X-MP [49]; Cray Y-MP [12]; Cray T3E [21, 42]; Grid environments [52]; Intel multicore processors and processor clusters [28]; GPUs [43]. Regarding the latter, it is worth noting the extensive studies conducted at Yale on an early GPU, the FPS-164 [38]. Proposals for special-purpose hardware are in [53]. RES were included in the PELLPACK parallel problem-solving environment for elliptic PDEs [26]. More information can be found in the annotated bibliography provided in [17]. As will become clear, RES are an important class of structured matrix computations whose design and parallelization depends on special mathematical transformations not usually applicable when dealing with direct matrix computations. This is a topic with many applications and the subject of intensive investigation in computational mathematics.

## Problem Formulation

RES are primarily applicable for solving numerically separable elliptic PDEs whose discretization (cf. [5, 29, 55]) leads to block Toeplitz tridiagonal linear systems of order  $N = mn$ ,

$$\mathcal{A}U = F, \text{ where } \mathcal{A} = \text{trid}_n[W, T, W], \\ \text{and } W, T \in \mathbb{R}^{m \times m}, \quad (1)$$

where  $W, T$  are symmetric, simultaneously diagonalizable and thus  $WT = TW$ . Symbol  $\text{trid}_n[C, A, B]$  denotes

block Toeplitz tridiagonal or Toeplitz tridiagonal matrices, where  $C, A, B$  are the (matrix or scalar) elements along the subdiagonals, diagonals, and superdiagonals, respectively, and  $n$  is their number.

A simple but very common problem is Poisson's equation on a rectangular region. After suitable discretization, e.g., using an  $m \times n$  rectangular grid, (1) becomes

$$\begin{pmatrix} T & -I & & \\ -I & T & -I & \\ & \ddots & \ddots & \ddots \\ & & -I & T & -I \\ & & & -I & T \end{pmatrix} \begin{pmatrix} U_1 \\ U_2 \\ \vdots \\ U_n \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \\ \vdots \\ F_n \end{pmatrix}, \quad (2)$$

where  $T = \text{trid}_m[-1, 4, -1]$  is Toeplitz symmetric and tridiagonal. Vectors  $U_i = [U_{i,1}, \dots, U_{i,m}]^\top \in \mathbb{R}^m$  for  $i = 1, \dots, n$  contain the unknowns at the  $i$ th grid row, and  $F_i = [F_{i,1}, \dots, F_{i,m}]^\top$  the values of the right-hand side  $F$  including the boundary terms and scalings related to the discretization. This is the problem for which RES are eminently applicable and mostly discussed in the literature of RES, hence the term "fast Poisson solvers." The homogeneous case,  $f \equiv 0$ , is Laplace's equation, that lends itself to even more specialized fast methods (not discussed here). RES can also be designed to solve the general separable problem

$$-(a(x)u_{xx} + b(x)u_x) - (d(y)u_{yy} + e(y)u_y) + (c(x) + \tilde{c}(y))u = f(x, y) \quad (3)$$

whose discrete form is

$$\begin{pmatrix} T + \alpha_1 I & -\beta_1 I & & \\ -\gamma_2 I & T + \alpha_2 I & -\beta_2 I & \\ & \ddots & \ddots & \ddots \\ & & -\gamma_{n-1} I & T + \alpha_{n-1} I & -\beta_{n-1} I \\ & & & -\gamma_n I & T + \alpha_{n-1} I \end{pmatrix} \begin{pmatrix} U_1 \\ U_2 \\ \vdots \\ U_n \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \\ \vdots \\ F_n \end{pmatrix}, \quad (4)$$

for scalars  $\alpha_j, \beta_j, \gamma_j$ . This is no longer block Toeplitz, but the diagonal and off diagonal blocks have special structure.

Once the equations become nonseparable and/or the domain irregular, RES are not directly applicable and other methods are preferred. All is not lost, however, because frequently RES could be used as building

blocks of more general solvers, e.g., in domain decomposition and preconditioning.

Sequential RES are much faster than direct solvers such as sparse Cholesky elimination (see ► [Sparse Direct Methods](#)). This advantage carries over to parallel RES. Other methods that can be used but are addressed elsewhere in this volume are multigrid methods (see ► [Algebraic Multigrid](#)) and preconditioned conjugate gradients (see ► [Iterative Methods](#)) that are the basis for software PDE packages such as PETSc.

## Mathematical Preliminaries and Notation

The following concepts from matrix analysis are essential to describe and analyze RES.

1. Kronecker products of matrices and their properties.
2. Chebyshev polynomials (first and second kind)  $C_d, S_d$ , and modified Chebyshev polynomials (second kind)  $\tilde{C}_d$ .
3. The analytic expression for the eigenvalues and eigenvectors of the symmetric tridiagonal Toeplitz matrix  $T = \text{trid}_m[-1, \alpha, -1]$ : Specifically

$$\lambda_j = \alpha - 2 \cos\left(\frac{\pi j}{m+1}\right), \quad q_j = \sqrt{\frac{2}{m+1}} \left[ \sin\left(\frac{\pi j}{m+1}\right), \dots, \sin\left(\frac{\pi jm}{m+1}\right) \right]^\top. \quad (5)$$

If  $Q = [q_1, \dots, q_m]$  then the product  $Qy$  for any  $y \in \mathbb{R}^m$  has elements  $\sqrt{\frac{2}{m+1}} \sum_{j=1}^m \sin\left(\frac{\pi ij}{m+1}\right)$  for  $i = 1, \dots, m$  and so is the *discrete sine transform* (DST) of  $y$  times a scaling factor. Also  $Q^\top = Q$  and  $Q^\top Q = I$ . The DST can be computed in  $O(m \log m)$  operations using FFT-type methods. Similar results for the eigenstructure also hold for slightly modified matrices that occur when the boundary conditions for the continuous problem are not strictly Dirichlet.

4. The fact (see [1, 3, 37]) that for any nonsingular  $T \in \mathbb{R}^{m \times m}$ , the matrix  $\mathcal{A} = \text{trid}_n[-I, T, -I]$  is nonsingular and  $\mathcal{A}^{-1}$  can be written as a block matrix with general term

$$(\mathcal{A}^{-1})_{ij} = \begin{cases} S_n^{-1}(T)S_{i-1}(T)S_{n-j}(T), & j \geq i, \\ S_n^{-1}(T)S_{j-1}(T)S_{n-i}(T), & i \geq j. \end{cases} \quad (6)$$

5. The vec operator and its inverse, unvec <sub>$n$</sub> . Also the *vec-permutation* matrix  $\Pi_{m,n} \mathbb{R}^{mn \times mn}$ , that is the

unique matrix such that  $\text{vec}(A) = \Pi_{m,n} \text{vec}(A^\top)$ , where  $A \in \mathbb{R}^{m \times n}$ . The matrix is orthogonal and  $\Pi_{m,n}(I_n \otimes \tilde{T}_m) = (\tilde{T}_m \otimes I_n)\Pi_{m,n}$ .

## Algorithmic Infrastructure

Most RES make use of the finite difference analogue of a key idea in the separation of variables method for PDEs, namely, that under certain conditions, some multidimensional problems can be solved by solving several one-dimensional problems [34]. In line with this, it turns out that RES for (2) make extensive use of and their performance greatly depends on two basic kernels: (1) Tridiagonal linear system solvers and (2) fast Fourier and trigonometric transforms. These kernels are also used extensively in collective form, that is for given tridiagonal  $T \in \mathbb{R}^{m \times m}$  and any  $Y \in \mathbb{R}^{m \times s}$ , solve  $TX = Y$  for  $s \geq 1$ . In many cases, shifted or multiply shifted tridiagonal systems with one or multiple right-hand sides must be solved. Also if the  $m \times m$  matrix  $Q$  represents the discrete sine, cosine or Fourier transform of length  $m$ , then RES call for the fast computation of  $QY$  as well as the inverse transforms  $Q^{-1}Y$ . The best known parallel techniques for solving tridiagonal systems are *recursive doubling*, *cyclic reduction*, and *parallel cyclic reduction*, that all need  $O(\log m)$  parallel arithmetic operations on  $O(m)$  processors, e.g., [23, 30, 56]. For more flexibility, it is possible to combine those or Gaussian elimination with *divide-and-conquer* methods. Fourier-based RES also require fast implementations of the discrete cosine transform and the discrete Fourier transform to handle Neumann and periodic boundary conditions. Many publications on RES (e.g., [23, 38, 45, 47]) include extensive discussion of these building blocks. It is common for the tridiagonal matrices to have additional properties that can be used for faster processing. They are usually symmetric positive definite, Toeplitz, and have the form  $\text{trid}_m[-1, \tau, -1]$ , where  $\tau \geq 2$ . When  $\tau > 2$ , they are diagonally dominant in which case it is possible to terminate early Gaussian elimination (as implemented by O'Donnell et al. in [38]) and cyclic reduction (as proposed by Hockney in [22]), at negligible error. ScaLAPACK includes efficient tridiagonal and narrow-banded solvers, but the Toeplitz structure is not exploited (see ►ScaLAPACK). A detailed description of the fast Fourier and other discrete transforms used in RES can be found in [54].

Chapter (see ►FFTW) documents very efficient algorithms for the single and multiple FFTs needed for kernels of type (2). Parallel algorithms for the fast transforms required in the context of the RES have  $O(\log m)$  complexity on  $m$  processors.

## Matrix Decomposition

MD refers to a large class of methods to solve systems with matrices such as (4). As Buzbee noted in [10]: “It seldom happens that the application of  $L$  processors would yield an  $L$ -fold increase in efficiency relative to a single processor, but that is the case with the MD algorithm.” Sameh et al. in [45] provided the first detailed study of parallel MD for Poisson’s equation. MD can be succinctly described, as shown early on by Lynch et al. and Egerváry, using the compact Kronecker product representation of  $\mathcal{A}$  (see [34] and extensive discussion and references in [2]). Of greatest relevance here is the case that  $\mathcal{A} = (I_n \otimes \tilde{T}_m + \tilde{T}_n \otimes I_m)$ , where any of  $\tilde{T}_m, \tilde{T}_n$  is diagonalizable using matrices expressed by Fourier or trigonometric transforms. Denoting by  $Q_x \in \mathbb{R}^{m \times m}$  (resp.  $Q_y \in \mathbb{R}^{n \times n}$ ) the matrix of eigenvectors of  $\tilde{T}_m$  (resp.  $\tilde{T}_n$ ), then  $\mathcal{A}U = F$  can be rewritten as

$$(I_n \otimes Q_x^\top)(I_n \otimes \tilde{T}_m + \tilde{T}_n \otimes I_m)(I_n \otimes Q_x) \\ (I_n \otimes Q_x^\top)U = (I_n \otimes Q_x^\top)F$$

and equivalently as

$$(I_n \otimes \tilde{\Lambda}_m + \tilde{T}_n \otimes I_m)(I_n \otimes Q_x^\top)U = (I_n \otimes Q_x^\top)F.$$

Applying the similarity transformation with the vector-permutation matrix  $\Pi_{m,n}$ , the system becomes

$$\underbrace{(\tilde{\Lambda}_m \otimes I_n + I_m \otimes \tilde{T}_n)}_{\mathcal{B}}(\Pi_{m,n}(I_n \otimes Q_x^\top)U) = \Pi_{m,n}(I_n \otimes Q_x^\top)F.$$

Matrix  $\mathcal{B}$  is block diagonal, so these are  $m$  independent tridiagonal systems of order  $n$  each. Thus

$$U = (I_n \otimes Q_x)\Pi_{m,n}^\top \mathcal{B}^{-1} \Pi_{m,n}(I_n \otimes Q_x^\top)F. \quad (7)$$

When either of  $\tilde{T}_m$  or  $\tilde{T}_n$  is diagonalizable with trigonometric or Fourier transforms, as in the matrix of (2) for the Poisson equation, where they both are of the form  $\text{trid}[-1, 2, -1]$ , and matrix (4) when the coefficients are constant in one direction, MD will also be called Fourier MD and can be applied at total cost  $O(N \log N)$ . For example, if  $Q_x$  can be applied in  $O(m \log m)$  operations, the overall cost becomes  $O(n m \log m)$  for Fourier MD. From (7) emerge the three major computational

phases of Fourier MD for (2). In phase (I), the term  $(I_n \otimes Q_x^\top) F$  is computed, which amounts to the DST of  $n$  vectors  $F_i, i = 1, \dots, n$  to compute the  $\hat{F}_i = Q_x^\top F_i$ . In phase (II),  $m$  independent tridiagonal systems with coefficient matrices  $B_j = \text{trid}_n[-1, \lambda_j^{(m)}, -1]$  are solved, where  $\lambda_j^{(m)}$  is the  $j$ th eigenvalue of  $T$  in (2). The right-hand sides are the  $m$  columns of  $[\hat{F}_1, \dots, \hat{F}_n]^\top$ . In terms of the vec- constructs, these are the  $m$  contiguous, length- $n$  subvectors of the  $mn$  length vector  $\Pi_{m,n} \text{vec}[\hat{F}_1, \dots, \hat{F}_n]$ . The respective solutions  $\hat{U}_j, j = 1, \dots, m$  are stacked and reordered into  $\bar{U} = \Pi_{m,n}^\top \text{vec}[\hat{U}_1, \dots, \hat{U}_m]$  so that phase (III) consists of independent application of a scaled, length  $m$  DST of each column of  $[\bar{U}_1, \dots, \bar{U}_m] = \text{unvec}_n \bar{U}$ . Opportunities for parallelism abound, and if  $mn$  processors are available and each tridiagonal system is solved using a parallel algorithm of logarithmic complexity, then MD can be accomplished in  $O(\log mn)$  parallel operations. There are several ways of organizing the computation. When there are  $n = m$  processors, phases (I) and (III) can be performed independently in  $m \log m$  steps. This will leave the data in the middle phase distributed across the processors so that the multiple tridiagonal systems will be solved with a parallel algorithm that would require significant data movement. Instead, one can transpose the output data from phase (I) and also at the end of phase (II) so that all data is readily accessible from any processor that needs it. This approach has the advantage that it builds on mature numerical software for uniprocessors. It requires, however, the use of efficient algorithms for matrix transposition.

MD algorithms have been designed and implemented on vector and parallel architectures and are frequently used as the baseline in evaluating new Poisson solvers; cf. [10, 12–14, 21, 25, 35, 36, 38, 41, 45, 51] and [17] for more details. More applications of MD can be found in the survey [2] by Bialecki et al.

### Complete Fourier Transform

The idea of CFT was discussed by Hyman in [27] and described in the context of the tensor product methods of Lynch et al. (cf. [5, 34]). For low cost, this requires that both  $\tilde{T}_m$  and  $\tilde{T}_n$  are diagonalizable with Fourier or trigonometric transforms, as is the case for (2). Then the solution of (2) is

$$U = (Q_y \otimes Q_x)(I_n \otimes \tilde{\Lambda}_m + \tilde{\Lambda}_n \otimes I_m)^{-1}(Q_y^\top \otimes Q_x^\top) F,$$

where matrix  $(I_n \otimes \tilde{\Lambda}_m + \tilde{\Lambda}_n \otimes I_m)$  is diagonal. Multiplication by  $Q_y^\top \otimes Q_x^\top$  amounts to performing  $n$  independent DSTs of length  $m$  and  $m$  independent DSTs of length  $n$ ; similarly for  $Q_y \otimes Q_x$ . The middle phase is an element-by-element division by the diagonal of  $I_n \otimes \tilde{\Lambda}_m + \tilde{\Lambda}_n \otimes I_m$  that contains the eigenvalues of  $\mathcal{A}$ .

The parallel computational cost is  $O(\log m + \log n)$  operations on  $O(mn)$  processors. Hockney and Jesshope [23], Swarztrauber and Sweet [47], among others, studied CFT for various parallel systems. In the latter, CFT was found to have lower computational and communication complexity than MD and BCR for systems with  $O(mn)$  processors connected in a hypercube. Specific parallel implementations were described by O'Donnell et al. in [38] for the FPS-164 attached array processor where CFT had lower performance than MD and FACR(1), by Cote in [13] for Intel hypercubes; see also [17].

### Block Cyclic Reduction

BCR is a solver for (1) that generalizes the (scalar) cyclic reduction of Hockney and Golub. It is also more general than Fourier MD because it does not require knowledge of the eigenstructure of  $T$  neither uses fast transforms. A detailed analysis is found in [9]. BCR was key to the design of FISHPAK, the influential numerical library by Swarztrauber, Sweet, and Adams.

It is outlined next for  $n = 2^k - 1$  blocks, though, as shown by Sweet, the method can be modified to handle any  $n$ . For steps  $r = 1, \dots, k-1$ , adjacent blocks of equations are combined in groups of three to eliminate two blocks of unknowns; in the first step, for instance, unknowns from even-numbered blocks are eliminated and a reduced system  $\mathcal{A}^{(1)} = \text{trid}_{2^{k-1}-1}[-I, A^2 - 2I, -I]$ , containing approximately half the blocks remains. Setting  $T^{(0)} = T$ , and  $T^{(r)} = (T^{(r-1)})^2 - 2I$ , at the  $r$ th step the reduced system is

$$\text{trid}_{2^{k-r}-1}[-I, T^{(r)}, -I] \mathbf{U}^{(r)} = \mathbf{F}^{(r)},$$

where  $\mathbf{F}^{(r)} = \text{vec}[F_{2^r,1}, \dots, F_{2^r,(2^{k-r}-1)}]$ . These are computed by

$$F_{2^r,j} = F_{2^{r-1}(j-1)} + F_{2^{r-1}(j+1)} + T^{(r-1)} F_{2^{r-1}j}. \quad (8)$$

Matrix  $T^{(r)}$  can be written in terms of Chebyshev polynomials,  $T^{(r)} = 2C_{2^r}\left(\frac{T}{2}\right) = \tilde{C}_{2^r}(T)$ . The roots are known analytically,  $\rho_i^{(r)} = 2\cos\left(\frac{(2i-1)\pi}{2^{r+1}}\right)$ , therefore the product form of  $T^{(r)}$  is also available.

After  $r = k - 1$  steps, only the system  $T^{(k)}U_{2^{k-1}} = F_{2^{k-1}}^{(r)}$  remains. After this is solved, a back substitution phase recovers the remaining unknowns. Each reduction step requires  $2^{k-r} - 1$  independent matrix-vector multiplications with  $T^{(r-1)}$ . All multiplications with  $T^{(r-1)}$  are done using its product form so that the total cost is  $O(nm \log n)$  operations without recourse to fast transforms. Even with unlimited parallelism, the systems composing  $T^{(r)}$  must be solved in sequence, which creates a parallelization bottleneck. Also the algorithm as stated (sometimes called CORF [9]) is not numerically viable because the growing discrepancy in the magnitude of the terms in (8) leads to excessive roundoff.

Both problems can be overcome. Stabilization is achieved by means of a clever modification due to Buneman and analyzed in detail in [9], in which the unstable recurrence (8) is replaced with a coupled recurrence for a pair of new vectors, that can be computed stably, by exchanging tridiagonal matrix-vector multiplications with tridiagonal linear system solves. The number of sequential operations remains  $O(mn \log n)$ .

The parallel performance bottleneck can be resolved using *partial fractions*, a versatile tool for parallel processing first used, ingeniously, by H.T. Kung in [33] to enable the evaluation of arithmetic expressions such as  $x^n, \prod_{i=1}^n (x + \rho_i)$  and Chebyshev polynomials in few parallel divisions and  $O(\log n)$  parallel additions. For scalars, these findings are primarily of theoretical interest since the expressions can also be computed in  $O(\log n)$  multiplications. The idea becomes much more attractive for matrices (the problem of computing powers of a matrix is mentioned in [33] but not pursued any further) as is the case of BCR in the CORF or the stable variants of Buneman. Specifically, the parallelization bottleneck that occurs when solving

$$T^{(r)}X_r = Y_r, \text{ where } T^{(r)} = \prod_{i=1}^{2^r} (T - \rho_i^{(r)} I) \text{ and} \\ X_r, Y_r \in \mathbb{R}^{m \times (2^{k-r}-1)}$$

for large  $r$  is resolved because, for each  $r$ , the roots  $\rho_i^{(r)}, i = 1, \dots, 2^r$  are distinct and so for any right-hand side  $f \in \mathbb{R}^m$ ,

$$(T^{(r)})^{-1}f = \sum_{i=1}^{2^r} \alpha_i^{(r)} (T - \rho_i^{(r)} I)^{-1} f, \quad (9)$$

where the  $\alpha_i^{(r)}$  are the corresponding partial fraction coefficients that can be easily computed from the values of the derivative of  $\tilde{C}_{2^r}(z)$  at  $z = \rho_i^{(r)}$ . This was described by Sweet (see [48]) and also by Gallopoulos and Saad in [18]. Moreover, when BCR is applied for more general values of  $n$  or other boundary conditions that lead to more general matrix rational functions, partial fractions eliminate the need for operations with the numerator. Expression (9) can be evaluated independently for each right-hand side. Therefore, solving with coefficient matrix  $T^{(r)}$  and  $2^{k-r} - 1$  right-hand sides for  $r = 1, \dots, k - 1$  can be accomplished by solving  $2^r$  independent tridiagonal systems for each right-hand side and then combining the partial solutions by multiplying  $2^{k-r} - 1$  matrices, each of size  $m \times 2^r$  with the vector of  $2^r$  partial fraction coefficients.

The algorithm has parallel complexity  $O(\log n \log m)$ . Even though this is somewhat inferior to parallel MD and CFT, BCR has wider applicability; cf. the survey of Bini and Meini in [4] for many additional uses and parallelization. References [18, 20, 31, 47] discuss the application of partial fraction based parallel BCR and evaluate its performance on vector processors and multiprocessors, multicluster vector multiprocessors, and hypercubes. As shown by Calvetti et al. in [11], the use of partial fractions above is numerically safe for the polynomials that occur when solving Poisson's equation, but caution is required in the general case.

The above techniques for parallelizing MD and BCR were implemented in CRAYFISHPAK, a package that contains most of the functionality of FISHPAK (results for the Cray Y-MP listed in [49]). Beyond BCR, partial fractions can also be used to parallelize the computation important matrix functions, e.g., the matrix exponential.

## FACR

This method, proposed in [22], is based on the fact that at any step of the reduction phase of BCR, the coefficient matrix is  $\mathcal{A}^{(r)} = \text{trid}_{2^{k-r}-1}[-I, T^{(r)}, -I]$ , and that because  $T^{(r)} = 2C_{2^r}(\frac{T}{2})$  has the same eigenvectors as  $T$  and eigenvalues  $2C_{2^r}(\frac{\lambda_j^{(m)}}{2})$ , the reduced system can be solved using Fourier MD. FACR( $l$ ) (acronym for Fourier Analysis Cyclic Reduction) is a hybrid algorithm consisting of the following phases: (1) Perform  $l$  steps of BCR to obtain  $\mathcal{A}^{(l)}$  and  $F^{(l)}$ . (2) Use MD to

solve  $\mathcal{A}^{(l)} U^{(l)} = F^{(l)}$ . (3) Use  $l$  steps of back substitution to obtain the remaining subvectors of  $U$ . Research of Hockney, Swarztrauber, and Temperton has shown that a value  $l \approx \log \log m$  reduces the number of sequential operations down to  $O(mnl)$ . Therefore, properly designed FACR is faster than MD and BCR. In practice, the best choice for  $l$  depends on the relative performance of the underlying kernels and other characteristics of the computer platform. The selection for vector and parallel architectures has been studied at length by Hockney (cf. [23] and references therein) and Jesshope and by Briggs and Turnbull [8]. A general conclusion is that  $l$  is typically very small (so that BCR can be applied in its simplest CORF form), especially for high levels of parallelism, and that it is worth determining it empirically.

The parallel implementation of all steps can proceed using the techniques deployed for BCR and MD; in fact,  $\text{FACR}(l)$  can be viewed as an alternative to partial fractions to avoid the bottleneck to parallelization that was observed after a few steps of reduction in BCR. For example, one can monitor the number of systems that can be solved in parallel in BCR, and before the number of independent computations is small and no longer acceptable, switch to MD. This approach was analyzed by Briggs and Turnbull in [8] on a shared-memory multiprocessor. Corroborating the complexity estimates above, FACR is frequently found to be faster than MD and CFT; see, e.g., [38] for results on an attached array processor.

## Marching and Other Methods

Marching methods are RES that can be used to solve Poisson's equation as well as general separable PDEs. The idea was already mentioned in [27] and by 1980, as recounted by Hockney [24], a marching algorithm by Lorenz was cautiously cited as the fastest performing RES for Poisson's equation on uniprocessors, provided that it is "used sensibly." The caution is due to numerical instability that can have deleterious effects on accuracy. Bank and Rose in [1] proposed *Generalized Marching* (GM) methods that are stable with operation count  $O(n^2 \log \frac{n}{k})$  when  $m = n$ , where  $k$  is determined by the sought accuracy, instead of the theoretical  $O(n^2)$  for simple but unstable marching.

The key idea in marching methods is that if one knows  $U_n$  in (2), then the remaining  $U_{n-1}, \dots, U_1$  could

be computed from this and the boundary values  $U_{n+1}$  by the recurrence  $U_{j-1} = F_j + U_{j+1} - TU_j$  in  $O(mn)$  operations. Vector  $U_n$  can also be computed in  $O(mn)$  operations in the course of block LU of a reordering of (2) by solving a linear system with a matrix rational function of Chebyshev polynomials like  $S_n$ .

The process is not numerically viable for large  $n$ , thus in GM the domain is partitioned into  $k$  strips that are small enough so that sufficient accuracy is maintained when marching is used to compute the solution in each. This is a form of *domain decomposition*, whose merits for parallel processing are discussed in the next section. The interface values of  $U$  necessary to start the independent marches can be computed independently extending the process described above. Additional parallelism is available from using partial fractions.

When the right-hand-side  $F$  of (1) is sparse and one is interested in only few values of  $U$ , then the cost of computing such a solution with MD is reduced. Based on an idea of Banegas, Vassilevski and Kuznetsov designed variants of cyclic reduction that generate systems where a *partial solution method* is deployed to build the final solution. Parallel algorithms using these ideas were implemented by Petrova (on a workstation cluster) [39] and by Rossi and Toivanen (algorithm PSCR on a Cray T3E) who also showed a radix- $q$  version of cyclic reduction that eliminates a factor of  $q \geq 2$  blocks at a time. As with other RES (e.g., [46]), to solve the general separable system (4), these methods require additional preprocessing to compute eigenvalues and eigenvectors that are not available analytically.

The special form of the inverse blocks of (2) in (6) shows that both the matrix and its inverse are "data sparse," a property that is used to design effective approximate algorithms to solve elliptic equations using an integral equation formulation; cf. [6, 15]. The explicit formula for the inverse and partial fractions can be combined to solve (2) for any rows of the grid in  $O(\log N)$  parallel operations, as proposed in [19]. Marching can also be parallelized by diagonalizing  $T$ . Description of this and other methods are found in the monograph [53] by Vajteršic.

## Domain Decomposition

Domain decomposition (DD) is a methodology for solving elliptic PDEs in which the solution is constructed

by partitioning and solving the problem on subdomains (when these do not overlap, DD is called substructuring) and synthesizing the solution by combining the partial solutions, usually iteratively, until a satisfactory solution is obtained. Whenever the equations and subdomains are suitable, RES can be used as building blocks for DD methods; the algorithm outlined by Buzbee et al. in [9, Sect. 9] is a case in point and confirms that RES were an early motivation behind the development of DD. In some cases, one can dispense of the iterative process altogether and compute the solution to the PDE at sequential cost  $O(N \log N)$ , thus extending RES to irregular domains. DD naturally lends itself for parallel processing (see ▶Domain Decomposition), introducing another layer of parallelism in the RES described so far and thus providing greater flexibility in their mapping on parallel architectures, e.g., when processors are organized in loosely coupled clusters, for better load balancing on heterogeneous processors, etc. One such method based on Fourier MD was proposed in [44] and can be viewed as a special case of the Spike banded solver method developed by Sameh and collaborators (cf. [40]) (▶Spike).

A slightly different formulation of this DD solver is based on reordering the equations in (2) combined with block  $LU$ . This was investigated by Chan and Resasco and then at length with hypercube implementations (cf. [41] and references therein) and by Chan and Fatoohi for vector multiprocessors in [12]. The interface values in domain decomposition can also be computed using the partial solution methodology, e.g., in combination with GM as suggested by Vassilevski and studied by Bencheva in the context of parallel processing (cf. [17]). DD was also used by Barberou to increase the performance of the partial solution method of Rossi and Toivanen on a computational Grid of high-performance systems [52].

## Extensions

The above methods can be extended to solve Poisson's equation in three dimensions. The corresponding matrix will be multilevel Toeplitz: It is block Toeplitz tridiagonal with each block being of the form (2). Sameh in [44] proposed a six phase parallel MD-like algorithm that combines independent FFTs in two dimensions and tridiagonal system solutions along the third dimension. Similar parallel RES for the Intel hypercube were

described in [41, 50] and in [21] for the Cray T3E. The partial solution method was also extended to three-dimensional problems, e.g., [52, cf. comments on solver PDC3D].

The parallel RES methods above can be extended to handle other types of boundary conditions (Dirichlet, Neumann and periodic) and operators (e.g., biharmonic) and provide a blueprint for the design of parallel RES of higher order of accuracy, e.g., the FACR-like method FFT9 of Houstis and Papatheodorou (see [5, 17]).

## Related Entries

- ▶Algebraic Multigrid
- ▶Domain Decomposition
- ▶FFTW
- ▶Illiac IV
- ▶Preconditioners for Sparse Iterative Methods
- ▶ScaLAPACK
- ▶Sparse Direct Methods
- ▶Spike

## Bibliography

1. Bank RE, Rose DJ (1977) Marching algorithms for elliptic boundary value problems. Part I: the constant coefficient case. Part II: the variable coefficient case. SIAM J Numer Anal 14(5):792–829 (Part I); 950–969 (Part II)
2. Bialecki B, Fairweather G, Karageorghis A (2010) Matrix decomposition algorithms for elliptic boundary value problems: a survey. Numer Algorithms. <http://www.springerlink.com/content/g66527532724p607/fulltext.pdf>
3. Bickley WG, McNamee J (1960) Matrix and other direct methods for the solution of systems of linear difference equations. Philos Trans R Soc Lond A: Math Phys Sci 252(1005):69–131
4. Bini DA, Meini B (2008) The cyclic reduction algorithm: from Poisson equation to stochastic processes and beyond. Numer Algorithms 51(1):23–60
5. Birkhoff G, Lynch RE (1984) Numerical solution of elliptic problems. SIAM, Philadelphia
6. Börm S, Grasedyck L, Hackbusch W (2006) Lecture notes 21/2003: hierarchical matrices. Technical report, Max Planck Institut fuer Mathematik in den Naturwissenschaften, Leipzig
7. Botta EFF et al (1997) How fast the Laplace equation was solved in 1995. Appl Numer Math 24(4):439–455
8. Briggs WL, Turnbull T (1988) Fast Poisson solvers for MIMD computers. Parallel Comput 6:265–274
9. Buzbee B, Golub G, Nielson C (1970) On direct methods for solving Poisson's equation. SIAM J Numer Anal 7(4):627–656 (see also the comments by Buzbee in Current Contents, 36, September 1992)

10. Buzbee BL (1973) A fast Poisson solver amenable to parallel computation. *IEEE Trans Comput C-22(8)*:793–796
11. Calvetti D, Gallopoulos E, Reichel L (1995) Incomplete partial fractions for parallel evaluation of rational matrix functions. *J Comput Appl Math* 59:349–380
12. Chan TF, Fatoohi R (1989) Multitasking domain decomposition fast Poisson solvers on the Cray Y-MP. In: Proceedings of the fourth SIAM conference on parallel processing for scientific computing. SIAM, Philadelphia, pp 237–244
13. Cote SJ (1991) Solving partial differential equations on a MIMD hypercube: fast Poisson solvers and the alternating direction method. Technical report UIUCDCS-R-91-1694, University of Illinois at Urbana-Champaign, Urbana
14. Ericksen JH (1972) Iterative and direct methods for solving Poisson's equation and their adaptability to Illiac IV. Technical report UIUCDCS-R-72-574, Department of Computer Science, University of Illinois at Urbana-Champaign
15. Ethridge F, Greengard L (2001) A new fast-multipole accelerated Poisson solver in two dimensions. *SIAM J Sci Comput* 23(3): 741–760
16. Gallivan KA, Heath MT, Ng E, Ortega JM, Peyton BW, Plemmens RJ, Romine CH, Sameh AH, Voigt RG (1990) Parallel algorithms for matrix computations. SIAM, Philadelphia
17. Gallopoulos E. An annotated bibliography for rapid elliptic solvers. <http://scgroup.hpclab.ceid.upatras.gr/faculty/stratis/Papers/myresbib>
18. Gallopoulos E, Saad Y (1989) Parallel block cyclic reduction algorithm for the fast solution of elliptic equations. *Parallel Comput* 10(2):143–160
19. Gallopoulos E, Saad Y (1989) Some fast elliptic solvers for parallel architectures and their complexities. *Int J High Speed Comput* 1(1):113–141
20. Gallopoulos E, Sameh AH (1989) Solving elliptic equations on the Cedar multiprocessor. In: Wright MH (ed) Aspects of computation on asynchronous parallel processors. North-Holland, Amsterdam, pp 1–12
21. Giraud L (2001) Parallel distributed FFT-based solvers for 3-D Poisson problems in meso-scale atmospheric simulations. *Int J High Perform Comput Appl* 15(1):36–46
22. Hockney R (1965) A fast direct solution of Poisson's equation using Fourier analysis. *J Assoc Comput Mach* 12:95–113
23. Hockney R, Jesshope C (1983) Parallel computers. Adam Hilger, Bristol
24. Hockney RW (1980) Rapid elliptic solvers. In: Hunt B (ed) Numerical methods in applied fluid dynamics. Academic, London, pp 1–48
25. Hockney RW (1983) Characterizing computers and optimizing the FACR(l) Poisson solver on parallel unicompilers. *IEEE Trans Comput C-32(10)*:933–941
26. Houstis EN, Rice JR, Weerawarana S, Catlin AC, Papachio P, Wang K-Y, Gaitatzes M (1998) PELLPACK: a problem-solving environment for PDE-based applications on multicomputer platforms. *ACM Trans Math Softw* 24(1):30–73
27. Hyman MA (1951–1952) Non-iterative numerical solution of boundary-value problems. *Appl Sci Res B* 2:325–351
28. Intel Cluster Poisson Solver Library – Intel Software Network. <http://software.intel.com/en-us/articles/intel-cluster-poisson-solver-library/>
29. Iserles A (1996) Introduction to numerical methods for differential equations. Cambridge University Press, Cambridge
30. Johnsson L (1987) Solving tridiagonal systems on ensemble architectures. *SIAM J Sci Statist Comput* 8:354–392
31. Jwo J-S, Lakshmivarahan S, Dhall SK, Lewis JM (1992) Comparison of performance of three parallel versions of the block cyclic reduction algorithm for solving linear elliptic partial differential equations. *Comput Math Appl* 24(5–6):83–101
32. Knightley JR, Thompson CP (1987) On the performance of some rapid elliptic solvers on a vector processor. *SIAM J Sci Statist Comput* 8(5):701–715
33. Kung HT (1976) New algorithms and lower bounds for the parallel evaluation of certain rational expressions and recurrences. *J Assoc Comput Mach* 23(2):252–261
34. Lynch RE, Rice JR, Thomas DH (1964) Tensor product analysis of partial differential equations. *Bull Am Math Soc* 70:378–384
35. McBryan OA (1989) Connection machine application performance. Technical report CH-CS-434-89, Department of Computer Science, University of Colorado, Boulder
36. McBryan OA, Van De Velde EF (1987) Hypercube algorithms and implementations. *SIAM J Sci Stat Comput* 8(2): s227–s287
37. Meurant G (1992) A review on the inverse of symmetric tridiagonal and block tridiagonal matrices. *SIAM J Matrix Anal Appl* 13(3):707–728
38. O'Donnell ST, Geiger P, Schultz MH (1983) Solving the Poisson equation on the FPS-164. Technical report YALE/DCS/TR293, Yale University
39. Petrova S (1997) Parallel implementation of fast elliptic solver. *Parallel Comput* 23(8):1113–1128
40. Polizzi E, Sameh AH (2007) SPIKE: A parallel environment for solving banded linear systems. *Comput Fluids* 36(1):113–120
41. Resasco DC (1990) Domain decomposition algorithms for elliptic partial differential equations. Ph.D. thesis, Yale University, Department of Computer Science
42. Rossi T, Toivanen J (1999) A parallel fast direct solver for block tridiagonal systems with separable matrices of arbitrary dimension. *SIAM J Sci Stat Comput* 20(5):1778–1796
43. Rossinelli D, Bergdorf M, Cottet G-H, Koumoutsakos P (2010) GPU accelerated simulations of bluff body flows using vortex particle methods. *J Comput Phys* 229(9):3316–3333
44. Sameh AH (1984) A fast Poisson solver for multiprocessors. In: Birkhoff G, Schoenstadt A (eds) Elliptic problem solvers II. Academic, New York, pp 175–186
45. Sameh AH, Chen SC, Kuck DJ (1976) Parallel Poisson and biharmonic solvers. *Computing* 17:219–230
46. Swarztrauber PN (1974) A direct method for the discrete solution of separable elliptic equations. *SIAM J Numer Anal* 11(6):1136–1150
47. Swarztrauber PN, Sweet RA (1989) Vector and parallel methods for the direct solution of Poisson's equation. *J Comput Appl Math* 27:241–263

48. Sweet RA (1988) A parallel and vector cyclic reduction algorithm. *SIAM J Sci Statist Comput* 9(4):761–765
49. Sweet RA (1992) Vectorization and parallelization of FISHPAK. In: Dongarra J, Kennedy K, Messina P, Sorensen DC, Voigt RG (eds) Proceedings of the fifth SIAM conference on parallel processing for scientific computing. SIAM, Philadelphia, pp 637–642 (see also <http://www.cisl.ucar.edu/softlib/CRAYFISH.html>)
50. Sweet RA, Briggs WL, Oliveira S, Porsche JL, Turnbull T (1991) FFTs and three-dimensional Poisson solvers for hypercubes. *Parallel Comput* 17:121–131
51. Temperton C (1979) Fast Fourier transforms and Poisson solvers on Cray-1. In: Hockney RW, Jesshope CR (eds) Infotech state of the art report: supercomputers, vol 2. Infotech, Maidenhead, pp 359–379
52. Tromeur-Dervout D, Toivanen J, Garbey M, Hess M, Resch MM, Barberou N, Rossi T (2003) Efficient metacomputing of elliptic linear and non-linear problems. *J Parallel Distribut Comput* 63(5):564–577
53. Vajteršic M (1993) Algorithms for elliptic problems: efficient sequential and parallel solvers. Kluwer, Dordrecht
54. Van Loan C (1992) Computational frameworks for the fast Fourier transform. SIAM, Philadelphia
55. Widlund O (1978) A Lanczos method for a class of nonsymmetric systems of linear equations. *SIAM J Numer Anal* 15(4):801–812
56. Zhang Y, Cohen J, Owens JD (2010) Fast tridiagonal solvers on the GPU. In: Proceedings of the 15th ACM SIGPLAN PPoPP 2010, Bangalore, India, pp 127–136

## Reconfigurable Computer

- Blue CHiP

## Reconfigurable Computers

Reconfigurable computers utilize reconfigurable logic hardware – either standalone, as a part of, or in combination with conventional microprocessors – for processing. Computing using reconfigurable logic hardware that can adapt to the computation can attain only some level of the performance and power efficiency gain of custom hardware but has the advantage of being retargetable for many different applications.

## Bibliography

1. Hauck S, DeHon A (2008) Reconfigurable computing: the theory and practice of FPGA-based computing. Morgan Kaufmann, Burlington

## Reconstruction of Evolutionary Trees

- Phylogenetics

## Reduce and Scan

MARC SNIR

University of Illinois at Urbana-Champaign, Urbana, IL, USA

## Synonyms

Parallel Prefix Sums; Prefix; Prefix Reduction

## Definition

Given  $n$  inputs  $x_1, \dots, x_n$ , and an associative operation  $\otimes$ , a *reduction* algorithm computes the output  $x_1 \otimes \dots \otimes x_n$ . A *prefix* or *scan* algorithm computes the  $n$  outputs  $x_1, x_1 \otimes x_2, \dots, x_1 \otimes \dots \otimes x_n$ .

## Discussion

The reduction problem is a subset of the prefix problem, but as we shall show, the two are closely related. The reduction and scan operators are available in APL [8]: Thus  $+/\!3\ 7\ 1$  computes the value  $11$  and  $+/\!3\ 7\ 1$  computes the vector  $3\ 10\ 11$ . The two primitives appears in other programming languages and libraries, such as MPI [10].

There are two interesting cases to consider:

*Random access:* The inputs  $x_1, \dots, x_n$  are stored in consecutive location in an array.

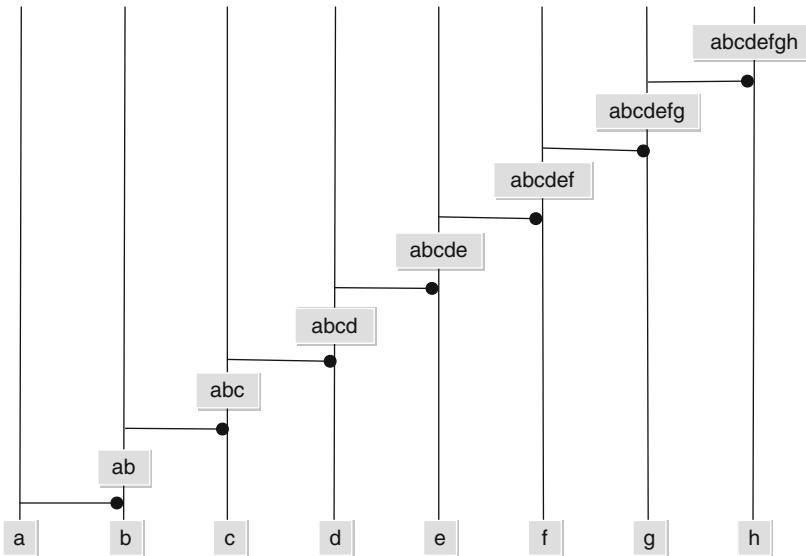
*Linked list:* The inputs  $x_1, \dots, x_n$  are stored in consecutive locations in a linked list; the links connect each element to its predecessor.

We analyze these two cases using a circuit or PRAM model, and ignoring communication costs.

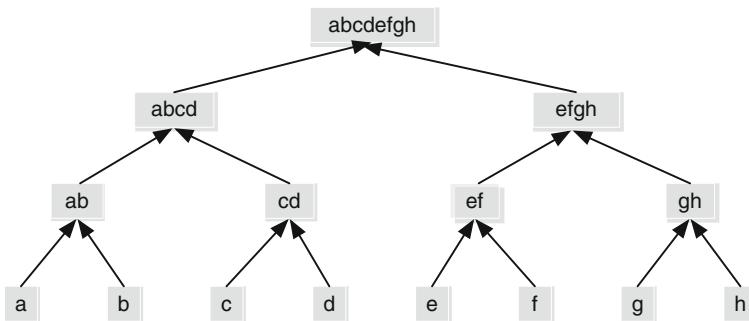
## Random-Access Reduce and Scan

The sequential circuit for a reduction computation is shown in Fig. 1; the corresponding sequential code is shown in Algorithm 1. This circuit has  $W = n - 1$  operations and depth  $D = n - 1$ . It also computes a scan.

One can take advantage of associativity in order to change the order of evaluation, so as to reduce the parallel computation time. An optimal parallel reduction



Reduce and Scan. Fig. 1 Sequential prefix circuit



Reduce and Scan. Fig. 2 Binary tree reduction

algorithm is achieved by using a balanced binary tree for the computation, as shown in Figs. 2 and 3, for  $n = 8$ . The corresponding algorithm is shown in Algorithm 2.

#### Algorithm 1 Sequential reduction and scan

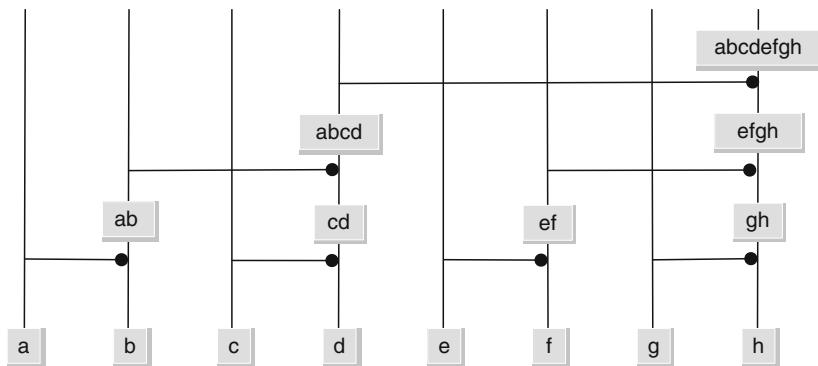
```
for ($i = 2$; $i \leq n$; $i++$)
 $x_i = x_{i-1} \otimes x_i$;
```

#### Algorithm 2 Binary tree reduction

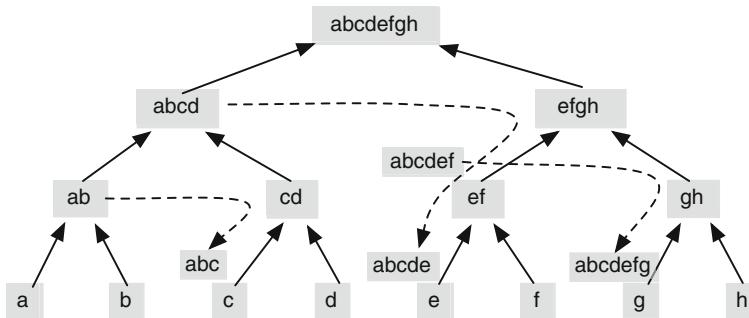
```
for ($i = 1$; $i \leq k$; $i++$)
 forall ($j = 2^i$; $j \leq 2^k$; $j += 2^i$)
 $x_j = x_{j-2^{i-1}} \otimes x_j$;
```

The algorithm has  $W = n - 1$  operations and depth  $D = \lceil \log n \rceil$ , which is optimal; it requires  $\lfloor n/2 \rfloor$  processes. If the number  $p$  of processes is smaller, then the inputs are divided into segments of  $\lceil n/p \rceil$  consecutive inputs; a sequential reduction is applied to each group, followed by a parallel reduction of  $p$  values; we obtain a running time of  $\lceil n/p \rceil - 1 + \log p$ . In particular, the reduction of  $n$  values can be computed in  $O(\log n)$  time, using  $n/\log n$  processes.

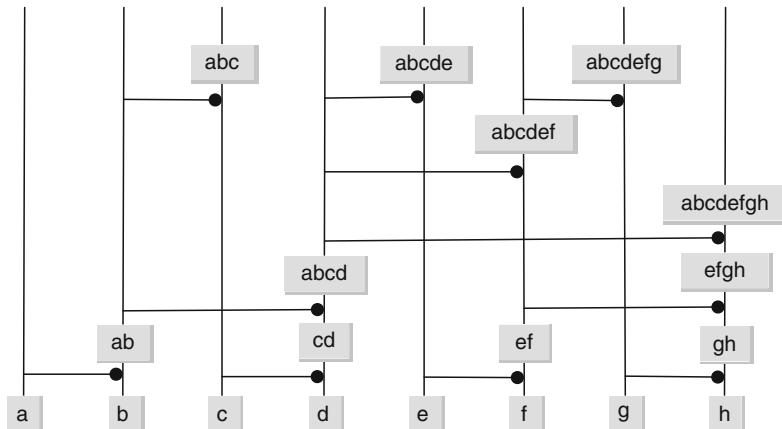
Given a tree that computes the reduction of  $n$  inputs, one can build from it a parallel prefix circuit, as shown in Figs. 4 and 5, for  $n = 8$ : The reduction is computed when values move up the tree, while the remaining prefix values are computed when values move down the tree. The algorithm is shown in Algorithm 3, for  $n = 2^k$ . This



Reduce and Scan. Fig. 3 Binary tree reduction



Reduce and Scan. Fig. 4 Tree prefix computation



Reduce and Scan. Fig. 5 Parallel prefix circuit

algorithm performs  $2n - 2\lfloor \lg n \rfloor - 2 \leq W \leq 2n - \lfloor \lg n \rfloor - 2$  operations and requires  $2\lfloor \lg n \rfloor - 1 \leq D \leq 2\lfloor \lg n \rfloor$  steps, with  $n/2$  processors.

If we have  $p \leq n/2$  processes, then one can compute the parallel prefix using the algorithm shown in

**Algorithm 4:** One computes prefix products sequentially for each sublist, combine the sublist products to compute a global parallel prefix, and then update sequentially elements in each sublist by the product of the previous sublists. The running time is

**Algorithm 3** Parallel prefix algorithm

---

```

for ($i = 1; i \leq k; i++$)
 forall ($j = 2^i; j \leq 2^k; j += 2^i$)
 $x_j = x_{j-2^{i-1}} \otimes x_j;$
for ($i = k - 1; i \geq 1; i--$)
 forall ($j = 2^i + 2^{i-1}; j \leq 2^k; j += 2^i$)
 $x_j = x_{j-2^{i-1}} \otimes x_j;$

```

---

**Algorithm 4** Parallel prefix with limited number of processes

---

```

// We assume that p divides n
forall ($i = 0; i < p; i++$)
 for ($j = i \times n/p + 1; j \leq (i+1) \times n/p; j++$)
 $x_j = x_{j-1} \otimes x_j;$
parallel prefix ($x_{n/p}, x_{2n/p}, \dots, x_n$);
forall ($i = 1; i < p; i++$)
 for ($j = i \times n/p + 1; j \leq (i+1) \times n/p; j++$)
 $x_j = x_{i \times n/p} \otimes x_j;$

```

---

$D = 2n/p + 2\lg p + O(1)$  and the number of operations is  $W = (2 - 1/p)n + O(1)$ .

It follows that the parallel prefix computation can be done in time  $O(\log n)$  using  $n/\log n$  processes.

The last circuit can be improved: Smir shows in [11] that any parallel prefix circuit must satisfy  $W+D \geq 2n-2$  and that this bound can be achieved for any  $2\log n - 2 \leq D \leq n - 1$ . Constructions for  $D$  in the range  $\log n \leq D \leq 2\log n - 2$  are given by Ladner and Fisher in [10]; they achieve  $D = \lceil \log n \rceil + k$  and  $W = (2 + 2^{\{1-k\}})n - o(n)$ . Thus, it is possible to have parallel prefix circuits of optimal depth and linear size.

The upper and lower bounds hold for circuits that only use the operation  $\otimes$ . Bilardi and Preparata study prefix computations in a more general Boolean network model [1]. In this model, prefix can be computed in time  $T$  by a network of size  $S = \Theta((n/T)\lg(n/T))$ , in general, or size  $S = \Theta(n/T)$  if the operation  $\otimes$  satisfies some additional properties, for any  $T$  in a range  $[c\lg n, n]$ .

## Applications

Parallel reduction is a frequent operation in parallel algorithms: It is required for parallel dot products, parallel matrix multiplications, counting, etc.

The parallel prefix algorithm has many applications, too. Blelloch shows in [2] how to use parallel prefix for sorting (quicksort and radix sort) and merging, graph algorithms (minimum spanning trees, connected components, maximum flow, maximal independent set, biconnected components), computational geometry (convex hull, K-D tree building, closest pair in plane, line of sight), and matrix operations. These algorithms use parallel prefix with the operations  $+$  and  $\max$ . Other usages are listed in [3]: addition in arbitrary precision arithmetic, polynomial evaluation, recurrence solution, solution of tridiagonal linear systems, lexical analysis, tree operations, etc. Greenberg et al. show how the use of parallel prefix can speed up discrete event simulations [5].

A few other examples are shown below.

*Enumeration and packing:* One has  $n$  processes and wants to enumerate all processes that satisfy some condition. Process  $i$  sets  $x_i$  to 1 if it satisfies the condition, 0 otherwise. A scan computation will enumerate the processes with a 1, giving each a distinct ordinal number. The same approach can be used to pack all nonzero entries of a vector into contiguous locations. Enumeration is used in radix-sort and quicksort to bin keys.

*Segmented scan:* A segmented parallel prefix operation computes parallel prefix within segments of the full list, rather than on the entire list. Thus  $+ \backslash [3 \ 5 \ | \ 2 \ 4 \ 1 \ | \ 7]$  yields  $[3 \ 8 \ | \ 2 \ 6 \ 7 \ | \ 7]$ .

Let  $\langle S, \otimes \rangle$  be a semi-ring ( $S$  is a set closed under the associative binary operator  $\otimes$ ). Define a new semi-ring  $\langle \hat{S}, \hat{\otimes} \rangle$  whose elements are  $S \cup \{|s : s \in S\}$  and the operation  $\hat{\otimes}$  is defined as follows:

$$\begin{aligned} a \hat{\otimes} b &= a \otimes b \text{ if } a, b \in S \\ |a \hat{\otimes} b &= |c, \text{ where } c = a \otimes b; \\ a \hat{\otimes} |b &= |a \hat{\otimes} |b = |b. \end{aligned}$$

The new operation is associative; a parallel prefix computation with the new operation computes segmented parallel prefixes for the old operation.

*Carry-lookahead adder:* Consider the addition of two binary numbers  $a_n, \dots, a_1$  and  $b_n, \dots, b_1$ . Let  $c_i$  be the carry bit at position  $i$ . Then the addition result is

$s_{n+1}, \dots, s_0$ , where  $s_{n+1} = c_n$ , and  $s_i = a_i \oplus b_i \oplus c_{i-1}$ , for  $i = 1, \dots, n$ .

Define  $p_i = a_i \vee b_i$  (carry propagate) and  $s_i = a_i \wedge b_i$  (carry set). The carry bits for the adder are defined by the recursion

$$\begin{aligned}c_0 &= 0; \\c_i &= p_i c_{i-1} \vee s_i.\end{aligned}$$

This can be written, in matrix form, as

$$\begin{aligned}\begin{pmatrix} c_0 & 0 \\ 1 & 0 \end{pmatrix} &= \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \\ \begin{pmatrix} c_i & 0 \\ 1 & 0 \end{pmatrix} &= \begin{pmatrix} p_i & s_i \\ 0 & 1 \end{pmatrix} \begin{pmatrix} c_{i-1} & 0 \\ 1 & 0 \end{pmatrix}\end{aligned}$$

with addition being  $\vee$  (or) and multiplication being  $\wedge$  (and). Thus, if

$$M_0 = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}, M_i = \begin{pmatrix} p_i & s_i \\ 0 & 1 \end{pmatrix}, \text{ for } i = 1, \dots, n,$$

we can compute the carry bits by computing the prefixes  $M_i, \dots, M_0$ , for  $i = 1, \dots, n$ ; any parallel prefix circuit can be used to build a carry-lookahead adder.

*Linear recurrences:* Consider an order one linear recurrence, of the form

$$x_i = a_i x_{i-1} + b_i.$$

The can be written as

$$\begin{pmatrix} x_i & 0 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} a_i & b_i \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_{i-1} & 0 \\ 1 & 0 \end{pmatrix},$$

with the usual interpretation of addition and multiplication. This has the same formal structure as the previous recursion.

The same approach extends to higher-order linear recurrences: A recurrence of the form  $x_i = \sum_{j=1}^k a_{ij} x_{i-j} + b_i$  can be solved in parallel by computing the parallel prefix of the product of  $(k+1) \times (k+1)$  matrices.

*Function composition:* Given  $n$  functions  $f_1, \dots, f_n$  consider the problem of computing  $f_1, f_2 \circ f_1, \dots, f_n \circ \dots \circ f_1$ . Since function composition is associative, then a parallel prefix algorithm can be used. This is a generalization of the usual formulation for prefix: Define

$$\begin{aligned}f_1 : x &\rightarrow a_1, \\f_i : x &\rightarrow x \otimes a_i, \text{ for } i = 2, \dots, n\end{aligned}$$

then  $f_i \circ \dots \circ f_1 = a_1 \otimes \dots \otimes a_i$ .

In this formulation, one has a set  $\mathcal{F}$  of functions that is closed under composition. This general formulation leads to a practical algorithm if functions in  $\mathcal{F}$  have a representation such that function composition is easily computed in that representation. The linear recurrence example is such a case: The affine functions  $x \rightarrow ax + b$  is represented by the matrix

$$\begin{pmatrix} a & b \\ 0 & 1 \end{pmatrix}$$

and function composition corresponds, in this representation, to matrix product.

Another example is provided by deterministic finite state transducers: The sequence of outputs generated by the automaton can be computed in logarithmic time from the sequence of inputs.

Finally, consider a sequence of accesses to a memory location that are either loads, stores, or fetch&adds. Then the values returned by the loads and fetch&adds can be computed in logarithmic time [8]. (*Fetch&add(address, increment)* is defined as the atomic execution of the following code: {temp = &address; &address += increment; return temp}; the operation increments a shared variable and returns its old value.)

## Linked List Reduce and Scan

The previous algorithms are not easily applied in the case where elements are in a linked list, as the rank of the elements is not known (indeed, computing the ranks is a parallel prefix computation!): It is not obvious which elements should be involved in the computation at each stage.

One approach is the *recursive-doubling* algorithm shown in [Algorithm 5](#), The algorithm has  $\lceil \lg n \rceil$  stages and uses  $n$  processes, one per node in the linked list; at stage  $i$ , each process computes the product of the (up

---

### Algorithm 5 Recursive doubling algorithm

---

```
for (i = 1; i ≤ ⌈ log n ⌉; i++)
 forall v in linked list
 if v.next ≠ NULL {
 v.x = (v.next → x) ⊗ v.x;
 v.next = v.next → next;
 }
```

---

to  $2^i$  values at consecutive nodes in the list ending with the process' node, and links that node to the node that is  $2^i$  positions down the list. This algorithm performs  $W = n[\log n] - 2^{[\log n]} + 1$  operations and requires

$D = [\log n]$  steps. The  $[\log n]$  depth is optimal, but this algorithm is not *work-efficient*, as it uses  $\Theta(n \log n)$  operations, compared to  $\Theta(n)$  for the sequential algorithm.

---

**Algorithm 6** Generic linked list reduce algorithm
 

---

```
Reduce(list L) {
 if ((L.head == &v)&&(v.next == NULL)) return v.x; // list has only one element
 else {
 pick an independent subset V of active nodes from L;
 forall v ∈ V {
 // delete predecessor of v from the list
 ptr = v.next;
 (v.x = ptr → x) ⊗ v.x;
 v.next = ptr → next;
 }
 }
 barrier;
 Reduce(L);
}
```

---

**Algorithm 7** Generic linked list scan algorithm
 

---

```
Scan(list L) {
 if ((L.head == &v)&&(v.next == NULL)) return v.x; // list has only one element
 else {
 pick an independent subset V of active nodes from L;
 forall v ∈ V {
 // delete predecessor of v from the list
 ptr = v.next;
 v.x = (ptr → x) ⊗ v.x;
 v.next = ptr → next;
 }
 }
 barrier;
 Scan(L);
 barrier;
 forall v ∈ V {
 // reinsert predecessor of v in list
 if (ptr.next ≠ NULL) ptr.x = ptr.next → x ⊗ ptr.x;
 v.next = ptr;
 }
 barrier;
}
```

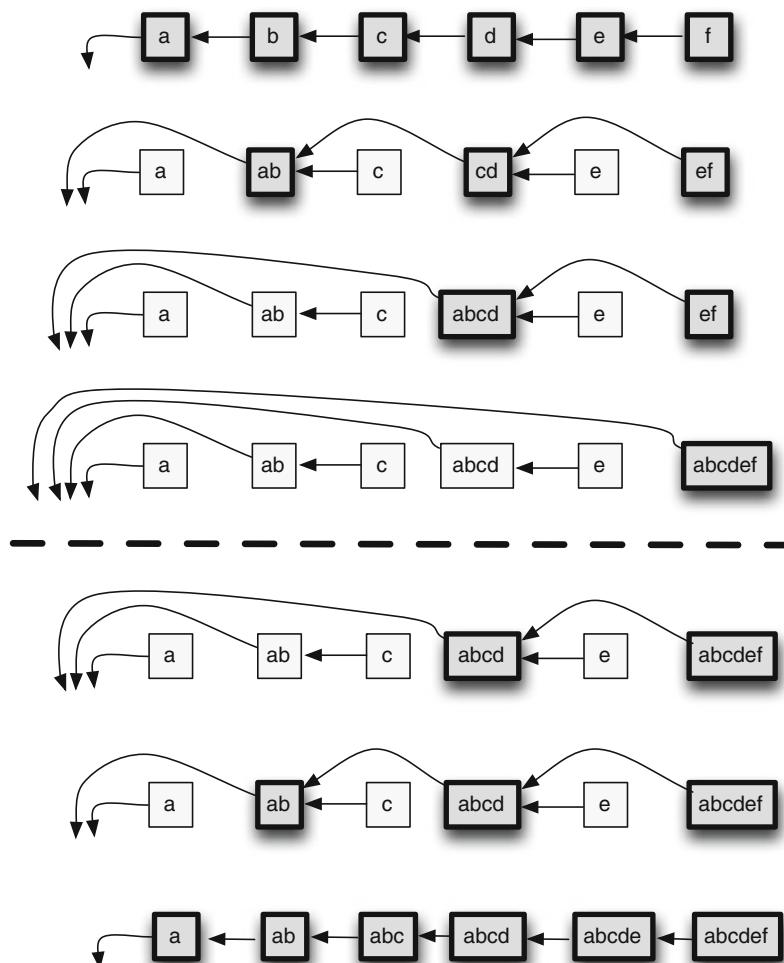
---

A set of nodes of the linked list is *independent* if each node in the set has a predecessor outside the set. A linked list reduction algorithm that performs no superfluous products has the generic form shown in [Algorithm 6](#). At each step of this parallel algorithm one replaces nonoverlapping pairs of adjacent nodes with one node that contains the product of these two nodes, until we are left with a single node. Given such reduction algorithm, we can design a parallel prefix algorithm, by “climbing back” the reduction tree, as shown in [Algorithm 7](#). The reduction is illustrated, for a possible schedule, on the top of [Fig. 6](#), while the unwinding of the recursion tree that computes the remaining prefix values is shown on the bottom of [Fig. 6](#).

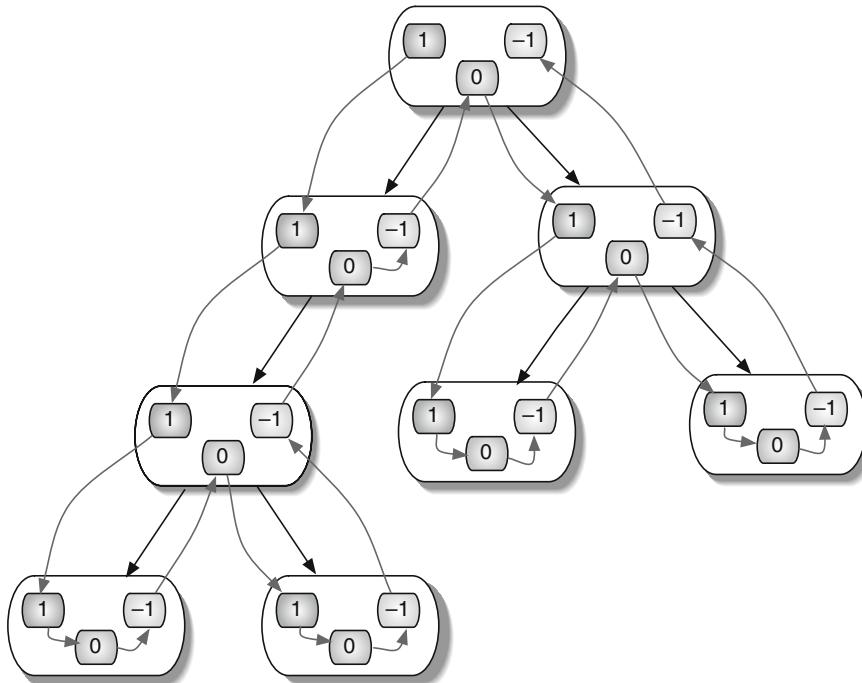
The problem, then, is to find a large independent subset and schedule the processes to handle the nodes in the subset at each round.

One possible approach is to use randomization for symmetry breaking: At each round, active processes choose randomly, with probability 1/2, whether to participate in this round; if a process decided to participate and its successor also decided to participate, it drops from the current round. The remaining processes form an independent set. On average, 1/4 of the processes that are active at the beginning of a round participate in the round. With overwhelming probability, the algorithm will require a logarithmic number of rounds.

Deterministic “coin-tossing” algorithms have been proposed by various authors. Cole and Vishkin are



**Reduce and Scan. Fig. 6** Linked list parallel prefix



### Reduce and Scan. Fig. 7 Tree depth computation

showing in [4] how to solve deterministically the linked list parallel prefix problem in time  $O(\log n \log^* n)$  using  $n/\log n \log^* n$  processes.

## ► Connection Machine

## ► MapReduce

► NESL

## ► Ultracomputer, NYU

## Applications

Linked list prefix computations can be used to solve a variety of graph problems, such as computing spanning forests, connected components, biconnected components, and minimum spanning trees [9].

A simple example is shown below.

*Tree depth computation:* Given a tree, one wishes to compute for each node the distance of that node from the root. One builds an Euler tour of the tree, as shown in Fig. 7. Each tree node is replaced by three nodes in the tour, with value 1 (moving down), 0 (moving right), and -1 (moving up). One can easily see that a prefix sum on the tour will compute the depth of each node. A variant of this algorithm (with -1 replaced by 0) will number the nodes in preorder [12].

### **Related Entries**

- ▶ Array Languages
  - ▶ Collective Communication

## Bibliography

1. Bilardi G, Preparata FP (1989) Size-time complexity of Boolean networks for prefix computations. J ACM 36:362–382
  2. Blelloch GE (1989) Scans as primitive parallel operations. IEEE Trans Comp 38:1526–1538
  3. Chatterjee B, Blelloch GE, Zagha M (1990) Scan primitives for vector computers. In: Proceedings of supercomputing conference. IEEE Computer Society Press, New York
  4. Cole R, Vishkin U (1986) Deterministic coin tossing with applications to optimal parallel list ranking. Inform Control 70:32–53
  5. Greenberg AG, Lubachevsky BD, Mitrani I (1996) Superfast parallel discrete event simulations. ACM Trans Model Comput Simul 6:107–136
  6. Iverson KE (1962) A Programming language. Wiley, New York
  7. Kruskal CP, Rudolph L, Snir M (1986) Efficient synchronization on multiprocessors with shared memory. ACM TOPLAS 10:579–601
  8. Kruskal CP, Rudolph L, Snir M (1990) Efficient parallel algorithms for graph problems. Algorithmica 5:43–64
  9. Ladner RE, Fischer MJ (1980) Parallel prefix computation. J Assoc Comput Mach 27:832–838

10. Message Passing Interface Forum (1994) MPI: a message-passing interface standard. *Int J Supercomput Appl High Perform Comput* 8:165–416
11. Snir M (1986) Depth-size trade-offs for parallel prefix computation. *J Algorithms* 7:185–201
12. Tarjan RE, Vishkin U (1985) An efficient parallel biconnectivity algorithm. *Siam J Comput* 14:862–874

## Relaxed Memory Consistency Models

► [Memory Models](#)

## Reliable Networks

► [Networks, Fault-Tolerant](#)

## Rendezvous

► [Synchronization](#)

## Reordering

MICHAEL HEATH

University of Illinois at Urbana-Champaign, Urbana, IL, USA

### Definition

*Reordering for parallelism* is a technique for enabling or enhancing concurrent processing of a list of items by determining an ordering that removes or reduces serial dependencies among the items.

### Discussion

Ordered lists are ubiquitous and seem entirely natural in many aspects of daily life. But the specific order in which items are listed may be irrelevant, or even

counterproductive, to the purpose of the list. In making a shopping list, for example, we usually list items in the order we happen to think of them, but rarely will such an order coincide with the most efficient path for locating the items in a store. Even for a structure that may have no natural linear ordering, such as a graph, we nevertheless typically number its nodes in some order, which may or may not facilitate efficient processing. In general, finding an optimal ordering is often a difficult combinatorial problem, as famously exemplified by the traveling salesperson problem.

Similarly, in programming when we write a *for* loop, we specify an ordering that may or may not have any significance for the successive tasks to be done inside the loop, yet the semantics of most programming languages usually require the compiler to schedule the tasks in the specified sequential order, even if the tasks are totally independent and could validly be done in any order. This constraint often goes unnoticed in serial computation, but it can seriously inhibit parallel processing of sequential loops. If each successive task depends on the result of the immediately preceding one, then the given sequential order must be honored in order for the program to execute correctly. But if the tasks have no such interdependence, for example, in a loop initializing all the elements of an array to zero, then the element assignments could be done in any order, or simultaneously in parallel.

Unfortunately, conventional programming languages offer no mechanism for indicating when tasks are independent, which has motivated the development of compiler techniques for discovering and exploiting loop-based parallelism, as well as compiler directives, such as those provided by OpenMP, that enable programmers to specify parallel loops. It has also motivated the development of higher-level languages in which structures such as arrays are first-class objects that can be referenced as a whole without having to enumerate array elements sequentially. Although smart compilers, compiler directives, and higher-level objects can help identify and preserve potential parallelism, it may still be difficult to determine how best to exploit the parallelism, especially if the ordering affects the total amount of work required. In such cases, the necessary reordering may be considerably deeper than it is reasonable to expect any compiler or automated system to identify or exploit.

These issues are well illustrated by computational linear algebra. When we write a vector in coordinate notation, we implicitly specify an ordering of the coordinate basis for the vector space that is generally arbitrary. Similarly, when we write a system of  $n$  linear equations in  $n$  unknowns,  $Ax = b$ , where  $A$  is an  $n \times n$  matrix,  $b$  is a given  $n$ -vector, and  $x$  is an  $n$ -vector to be determined, the ordering of the rows of  $A$  is arbitrary, as each equation in the system must be satisfied individually, irrespective of the order in which they happen to be listed. Consequently, the rows of the matrix  $A$  can be permuted arbitrarily without affecting the solution  $x$ . The column ordering of  $A$  is also arbitrary in the sense that it corresponds to the arbitrary ordering chosen for the entries of  $x$ , so that permuting the columns of  $A$  affects the solution only in that it correspondingly permutes the entries of  $x$ . To summarize, the solution to the system  $Ax = b$  is given by  $x = Qy$ , where  $y$  is the solution to the permuted system  $PAQy = Pb$ , and  $P$  and  $Q$  are any permutation matrices, so in this sense the original and permuted systems are mathematically equivalent, although the computational resources required to solve them may be quite different. This freedom in choosing the ordering can be exploited to enhance numerical stability (e.g., partial or complete pivoting) or computational efficiency, or a combination of these.

The order in which the entries of a matrix are accessed can have a dramatic effect on the performance of matrix algorithms, in part because of the memory hierarchy (registers, multiple levels of cache, main memory, etc.) typical of modern microprocessors. A dense matrix is usually stored in a two-dimensional array, whose layout in memory (typically row-wise or column-wise) strongly affects the cache behavior of programs that access the matrix entries. Many matrix algorithms, such as matrix multiplication and matrix factorization, systematically access the entries of the matrix using nested loops whose indices can validly be arranged in any order, so the order can be chosen to make optimal use of data residing in the most rapidly accessible level of the memory hierarchy. In addition, the order in which matrix rows or columns (or both, in the case of a two-dimensional decomposition) are assigned to processors may strongly affect parallel efficiency. Assignment by contiguous blocks, for example, tends to reduce communication but inhibits

concurrency and can yield a poor load balance, whereas assigning rows or columns to processors in a cyclic manner (like dealing cards) has the opposite effects. Optimizing the tradeoff between these extremes may suggest a hybrid block-cyclic assignment, with smaller blocks of tunable size assigned cyclically.

These issues are further complicated for matrices that are *sparse*, meaning that the entries are mostly zeros, so that it is advantageous to employ data structures that store only the nonzero entries (along with information indicating their locations within the matrix) and algorithms that operate on only nonzero entries. For a sparse matrix, the ordering of the rows and columns strongly affects the total amount of work and storage required to compute a factorization of the matrix. Sparsity can also introduce additional parallelism into the factorization process, beyond the substantial parallelism already available in the dense case.

To illustrate the effects of ordering for sparse systems, we focus on systems for which the sparse matrix  $A$  is symmetric and positive definite, so that we will not have the additional complication of ordering to preserve numerical stability as well. In this case, for a direct solution we compute the *Cholesky factorization*  $A = LL^T$ , where  $L$  is lower triangular, so that the solution  $x$  to the system  $Ax = b$  can be computed by solving the lower triangular system  $Ly = b$  for  $y$  by forward substitution and then the upper triangular system  $L^Tx = y$  for  $x$  by back substitution.

To see how the ordering of the sparse matrix affects the work, storage, and parallelism, we need to take a closer look at the Cholesky algorithm, shown in Fig. 1, in which only the lower triangle of the input matrix  $A$  is accessed, and is overwritten by the Cholesky factor  $L$ . The outer loop processes successive columns of the matrix. The first inner loop, labeled *cdiv(k)*, scales column  $k$  by dividing it by the square root of its diagonal entry, while the second inner loop, labeled *cmod(j, k)*, modifies each subsequent column  $j$  by subtracting from it a scalar multiple,  $a_{jk}$ , of column  $k$ .

The first important observation to make about the Cholesky algorithm is that a given *cmod(j, k)* operation may introduce new nonzero entries, called *fill*, into column  $j$  in locations that were previously zero. Such fill entries require additional storage and work to process, so we would like to limit the amount of fill, which is dramatically affected by the ordering of the matrix, as

```

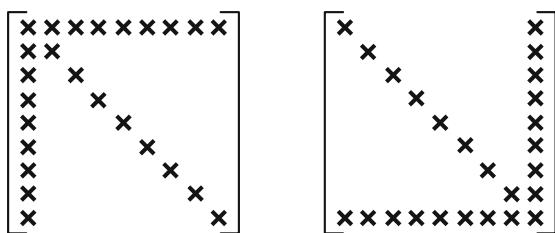
for k = 1 to n
 $a_{kk} = \sqrt{a_{kk}}$
 for i = k + 1 to n
 $a_{ik} = a_{ik}/a_{kk}$
 end
 for j = k + 1 to n
 for i = j to n
 $a_{ij} = a_{ij} - a_{ik}a_{jk}$
 end
 end
end

```

$\{cdiv(k)\}$

$\{cmod(j,k)\}$

**Reordering.** Fig. 1 Cholesky factorization algorithm



**Reordering.** Fig. 2 “Arrow” matrix example illustrating dependence of fill on ordering. Nonzero entries indicated by  $\times$

illustrated for an extreme case by the “arrow” matrix whose nonzero pattern is shown for a small example in Fig. 2 with two different orderings. The ordering on the left results in a dense Cholesky factor (complete fill), whereas the ordering on the right results in no new nonzero entries (no fill). In general, we would like to choose an ordering for the matrix  $A$  that minimizes fill in the Cholesky factor  $L$ , but this problem is known to be NP-complete, so instead we use heuristic ordering strategies such as *minimum degree* or *nested dissection*, which effectively limit fill at reasonable cost.

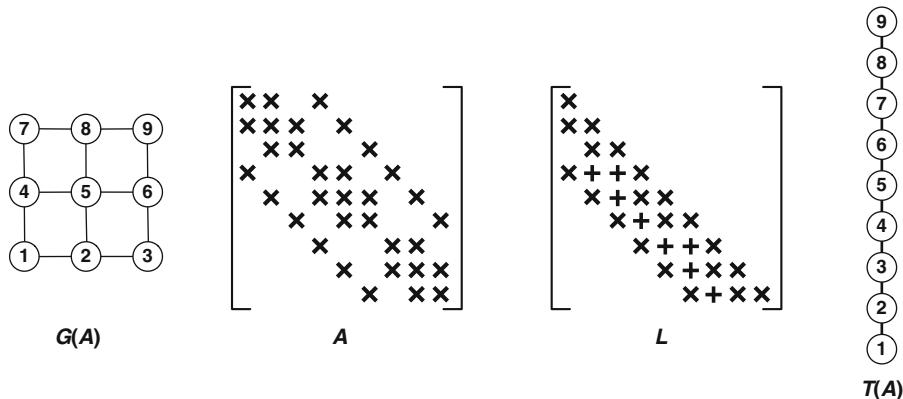
A second important observation is that the *cdiv* operation that completes the computation of a given column of  $L$  cannot take place until the modifications (*cmods*) by all previous columns have been done. Thus, the successive *cdiv* operations appear to require serial execution, and this is indeed the case for a dense matrix. But a given  $cmod(j, k)$  operation has no effect, and hence need not be done, if  $a_{jk} = 0$ , so a sparse matrix may therefore enable additional parallelism that is not available in the dense case. Once again, however, this potential benefit depends critically on the ordering chosen for the rows and columns of the sparse matrix.

Even if the *cdiv* operations must be performed serially, however, multiple *cmod* operations can still be performed in parallel, and this is the principal source of parallelism for a dense matrix.

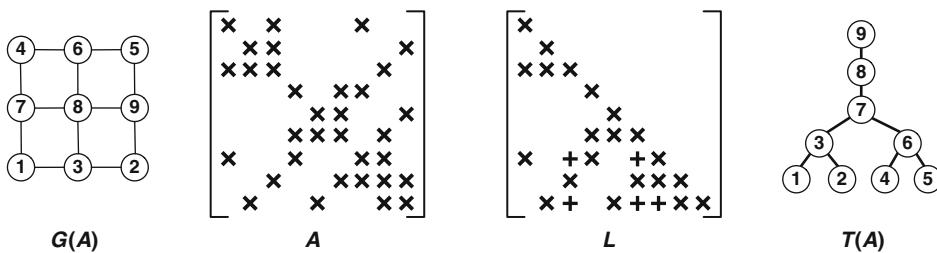
The interdependence among the columns of the Cholesky factor  $L$  is precisely characterized by the *elimination tree*, which has one node per column of  $A$ , with the parent of node  $j$  being the row index of the first subdiagonal nonzero in column  $j$  of  $L$ . In particular, each column of  $L$  depends only on its descendants in the elimination tree. For a dense matrix, this potential source of additional parallelism is of no help, as the elimination tree is simply a linear chain. But for a sparse matrix, more advantageous tree structures are possible, depending on the ordering chosen.

A small example is shown in Fig. 3, in which a two-dimensional mesh  $G(A)$ , whose edges correspond to nonzero entries of matrix  $A$ , is ordered row-wise, resulting in a banded matrix  $A$  and Cholesky factor  $L$ . With this ordering, the elimination tree  $T(A)$  is a linear chain, implying sequential execution of the corresponding *cdiv* operations, as each column of  $L$  depends on all preceding columns.

An alternative ordering is shown in Fig. 4, in which the same two-dimensional mesh is ordered by *nested dissection*, where the graph is recursively split into pieces with the nodes in the *separators* at each level numbered last. In this example, numbering the “middle” nodes last, that is, 7, 8, and 9, leaves two disjoint pieces that are in turn split by their “middle” nodes, numbered 3 and 6, leaving four isolated nodes, numbered 1, 2, 4, and 5. The absence of edges between the separated pieces of the graph induces blocks of zeros in the matrix  $A$  that are preserved in the Cholesky factor  $L$ , which accordingly suffers fewer fill entries than with the banded ordering, and hence requires less storage and work to compute. Equally important, the elimination tree now has a hierarchical structure, with multiple leaf nodes corresponding to *cdiv* operations that can be executed simultaneously in parallel at each level of the tree up to the final separator. The lessons from this small example apply much more broadly: with an appropriate ordering, sparsity can enable additional parallelism that is unavailable for dense matrices. A divide-and-conquer approach, exemplified by nested dissection, is an excellent way to identify and exploit such parallelism, and it typically also reduces the overall



**Reordering.** Fig. 3 Two-dimensional mesh  $G(A)$  ordered row-wise, with corresponding matrix  $A$ , Cholesky factor  $L$ , and elimination tree  $T(A)$ . Nonzero entries indicated by  $\times$  and fill entries by  $+$



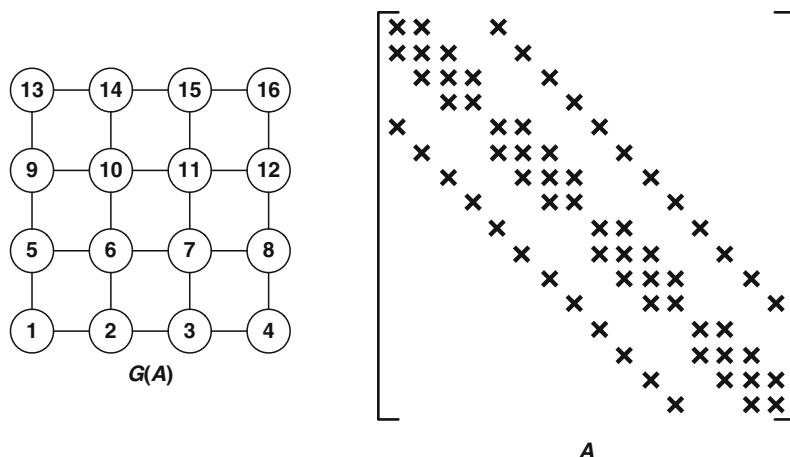
**Reordering.** Fig. 4 Two-dimensional mesh  $G(A)$  ordered by nested dissection, with corresponding matrix  $A$ , Cholesky factor  $L$ , and elimination tree  $T(A)$ . Nonzero entries indicated by  $\times$  and fill entries by  $+$

work and storage required for sparse factorization by reducing fill.

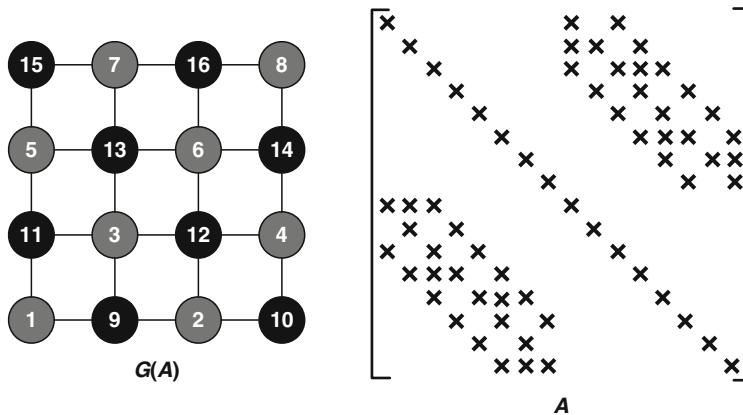
In addition to direct methods based on matrix factorization, reordering also plays a major role in parallel implementation of iterative methods for solving sparse linear systems. For example, reordering by nested dissection can be used to reduce communication in parallel sparse matrix-vector multiplication, which is the primary computational kernel in Krylov subspace iterative methods such as conjugate gradients. Another example is in parallel implementation of the Gauss-Seidel or Successive Overrelaxation (SOR) iterative methods. These methods repeatedly sweep through the successive rows of the matrix, updating each corresponding component of the solution vector  $x$  by solving for it based on the most recent values of all the other components. This dependence of each component update on the immediately preceding one seems to require strictly serial processing, as illustrated for a small example in Fig. 5, where a two-dimensional mesh is numbered in

a natural row-wise ordering, with the corresponding matrix shown, and indeed only one row can be processed at a time.

By reordering the mesh and corresponding matrix using a *red-black* ordering, in which the nodes of the mesh alternate colors in a checkerboard pattern as shown in Fig. 6, with the red nodes numbered before the black ones, solution components having the same color no longer depend directly on each other, as can be seen both in the graph and by the form of the reordered matrix. Thus, all of the components corresponding to red nodes can be updated simultaneously in parallel, and then all of the black nodes can similarly be done in parallel, so the algorithm proceeds in alternating sweeps between the two colors with ample parallelism. For an arbitrary sparse matrix, the same idea still works, but more colors may be required to color its graph, and hence the resulting parallel implementation will have as many successive parallel phases as the number of colors. One cautionary note, however, is that the convergence



**Reordering.** Fig. 5 Two-dimensional mesh  $G(A)$  ordered row-wise, with corresponding matrix  $A$ . Nonzero entries indicated by  $\times$



**Reordering.** Fig. 6 Two-dimensional mesh  $G(A)$  with red-black ordering and corresponding reordered matrix  $A$ . Nonzero entries indicated by  $\times$

rate of the underlying iterative method may depend on the particular ordering, so the gain in performance per iteration may be offset somewhat if an increased number of iterations is required to meet the same convergence tolerance.

In this brief article we have barely scratched the surface of the many ways in which reordering can be used to enable or enhance parallelism. We focused mainly on computational linear algebra, but additional examples abound in many other areas, such as the use of multicolor reordering to enhance parallelism in solving ordinary differential equations using waveform relaxation methods or in solving partial differential equations using domain decomposition methods. Yet another example is in the parallel implementation of

the Fast Fourier Transform (FFT), where reordering can be used to optimize the tradeoff between latency and bandwidth for a given parallel architecture.

## Related Entries

- [Dense Linear System Solvers](#)
- [Graph Partitioning](#)
- [Layout, Array](#)
- [Loop Nest Parallelization](#)
- [Linear Algebra, Numerical](#)
- [Loops, Parallel](#)
- [Scheduling Algorithms](#)
- [Sparse Direct Methods](#)
- [Task Graph Scheduling](#)

## Bibliographic Notes and Further Reading

For a broad overview of parallelism in computational linear algebra, see [2–4]. For a general discussion of sparse factorization methods, see [1, 5]. For specific discussion of reordering for parallel sparse elimination, see [6–8].

## Bibliography

1. Davis TA (2006) Direct methods for sparse linear systems. SIAM, Philadelphia, PA
2. Demmel JW, Heath MT, van der Vorst HA (1993) Parallel numerical linear algebra. *Acta Numerica* 2:111–197
3. Dongarra JJ, Duff IS, Sorenson DC, van der Vorst HA (1998) Numerical linear algebra for high-performance computers. SIAM, Philadelphia
4. Gallivan KA, Heath MT, Ng E, Ortega JM, Peyton BW, Plemmons RJ, Romine CH, Sameh AH, Voigt RG (1990) Parallel algorithms for matrix computations. SIAM, Philadelphia
5. George A, Liu JW-H (1981) Computer solution of large sparse positive definite systems. Prentice Hall, Englewood Cliffs
6. Liu JW-H (1986) Computational models and task scheduling for parallel sparse Cholesky factorization. *Parallel Comput* 3:327–342
7. Liu JW-H (1989) Reordering sparse matrices for parallel elimination. *Parallel Comput* 11:73–91
8. Liu JW-H (1990) The role of elimination trees in sparse factorization. *SIAM J Matrix Anal Appl* 11:134–172

## Resource Affinity Scheduling

### ► [Affinity Scheduling](#)

## Resource Management for Parallel Computers

### ► [Operating System Strategies](#)

## Rewriting Logic

### ► [Maude](#)

## Ring

### ► [Networks, Direct](#)

## Roadrunner Project, Los Alamos

DON GRICE<sup>1</sup>, ANDREW B. WHITE, JR.<sup>2</sup>

<sup>1</sup>IBM Corporation, Poughkeepsie, NY, USA

<sup>2</sup>Los Alamos National Laboratory, Los Alamos, NM, USA

## Synonyms

[Petaflop barrier](#)

## Definition

The Los Alamos Roadrunner Project was a joint Los Alamos and IBM project that developed the first supercomputer to break the Petaflop barrier on a Linpack run for the Top500 list.

## Discussion

### Introduction

In 2002, Los Alamos and IBM began to explore the potential of a new class of power-efficient high-performance computing systems based on a radical view of the future of supercomputing. The resulting LANL Roadrunner project had three primary roles: (1) an advanced architecture which presaged the primary components of next generation supercomputing systems; (2) a uniquely capable software and hardware environment for investigation and solution of science and engineering problems of importance to the US Department of Energy (DOE). Section 5.1 describes some of the key LANL applications that were studied; and (3) achievement of the first sustained petaflops.

A significant point in the thinking surrounding petascale and now exascale computing, is that the challenge is not just “solving the same problem faster” but rather “solving a better problem” by harnessing the leap in computing performance; that is, the key to better predictive simulations in national nuclear security, in climate and energy simulations, and in basic science is increasing the fidelity of our computational science by using better models, by increasing model resolution, and imbedding uncertainty quantification within the solution itself.

The computing capability of the supercomputer on the top of the Top500 list continues to grow at a staggering rate. In 2008, Roadrunner broke the sustained petaflops boundary on the list for the first time. This represented a performance improvement of over a

factor of 1,000 in just 11 years. As a result of the availability of such large machines, and the accompanying price and performance improvements for smaller machines, HPC solutions are now moving into markets that were previously considered beyond the reach of computing technology.

More comprehensive drug simulations, car crash simulations, and climate models are advances one might have easily predicted, but the impacts of HPC on movie making, games, and even remote surgery are things that might not have been so easy to see on the horizon. Today even financial institutions are using real-time modeling to guide investment choices. The growth of HPC opportunities in markets that did not involve traditional HPC users is what is driving the business excitement around the improvements in computing capability.

However, these opportunities do not come without their own technical challenges. The total amount of power and cooling required to drive some large systems, the programming complexity as hundreds of thousands of threads need to be harnessed into a single solution, and overall system reliability, availability, and serviceability (RAS) are the biggest hurdles. It was important to address the fundamental technology challenges facing further advances in application performance.

Designed and developed for the US Department of Energy (DOE) and Los Alamos National Laboratory

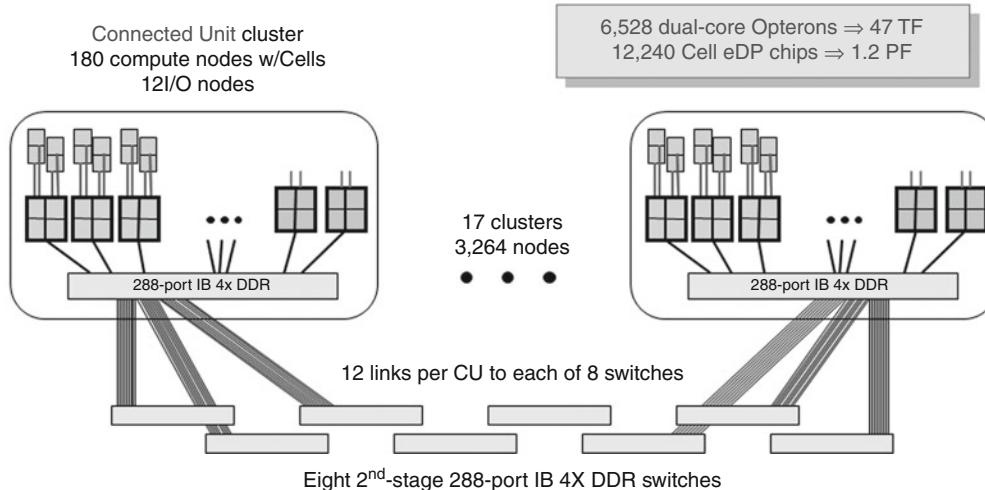
(LANL), the DOE/LANL Roadrunner project was a joint venture between LANL and IBM to address computing challenges with a range of technical solutions that are opening up new opportunities for business, scientific, and social advances. The Roadrunner project is named after the state bird of New Mexico where the Roadrunner system has been installed at LANL.

To address the power and performance issues, a fundamental design decision was made to use a hybrid programming approach that combines standard x86-architecture processors and Cell BE-based accelerators. The Cell BE accelerators provide a much denser and power-efficient peak performance than the standard processors, but introduce a different dimension in programming complexity because of the hybrid architecture and the necessity to explicitly manage memory on the Cell processor. When the project first started, this approach was considered radical, but now these attributes are widely accepted as necessary components of next generation systems leading to exascale. The Roadrunner program was targeted at getting a head start on the software issues involved in programming these architectures.

### Roadrunner Hardware Architecture

As shown in Fig. 1 and described in the Sect. “Management for Scalability,” the Roadrunner system

Roadrunner is a hybrid petascale system delivered in 2008



Roadrunner Project, Los Alamos. Fig. 1 Overall roadrunner structure

is a cluster of over 3,000 *triblade* nodes (described in detail below). The nodes are organized in smaller clusters called *connected units* (CUs). Each CU is composed of 180 computational triblade nodes, 12 I/O nodes, and a service node. Each CU has its own 288-port Voltaire Infiniband (IB) switch. [2] These CUs are then connected to each other with a standard second level switch topology where each of the first level switches has connections to each of the second level switches. There are 17 of these CUs in the overall system.

The triblade node architecture and implementation described in the Sect. “Triblade Architecture” were chosen in order to use existing hardware-building blocks and to capitalize on the RAS characteristics of the IBM Blade Center [8] infrastructure. Ease of construction and service were critical requirements because of the short schedule and the large number of parts in the final system. Since the compute nodes make up the bulk of the system hardware, they were the focus of the hardware and software design efforts.

## Management for Scalability

The management configuration of the Roadrunner system was designed for scalability from the beginning. The primary concern was the ability to use and control individual CUs without impacting the operation of the other CUs. That was the fundamental reason for having a first level IB switch dedicated to each CU. One can reboot the nodes in a CU, and even replace nodes in a CU, without impacting the operation of the other CUs, or applications running in those CUs. It is possible to isolate the first level switches from the second level switches with software commands so that other CUs do not route data to a CU that is under debug or repair.

The I/O nodes are dedicated to use by nodes within a CU which allows file system traffic to stay within a CU and not utilize any second level switch bandwidth. This also allows for file system connectivity and performance measurements to be done in isolation, including rebooting the CU, at the CU level. Actually, there are two sets of six I/O nodes for each CU which each serve half of the compute nodes, thus providing further performance isolation and greater performance consistency for applications.

## Triblade Architecture

As described in the Sect. “Heterogeneous Cores and Hybrid Architecture,” a fundamental innovation in the

Roadrunner design was the use of heterogeneous core types to maximize the energy efficiency of the system. Having cores designed for specific purposes allows for energy efficiency by eliminating circuits in the cores that will not be used. The three core types in the Roadrunner system are all programmable and not hardwired algorithmic cores, such as a cryptography engine, but the circuits are optimized for the type of code that will run on each core type.

There are two computational chips in the system: the PowerXCell 8i 3.2GHz chip which is packaged on the QS22 [5] blade and the AMD Opteron dual core 1.8GHz chip that is packaged on the LS21 [4] blade.

The PowerXCell 8i has eight specialized SPEs (Synergistic Processing Elements) that run the bulk of the application floating point operations but do not need the circuits nor the state to run the OS. It also has a PowerPC core that is neither as space nor energy efficient as the SPEs but it does run the OS on the QS22 and controls the program flow for the SPEs.

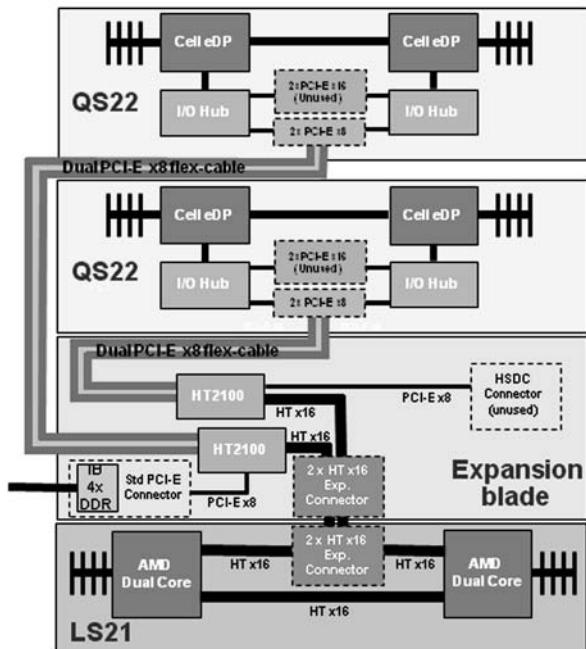
The AMD chip runs the MPI stack and controls the communications between the triblades in the system. It also controls the overall application flow and does application work for sections of the code that are not floating point intensive. The LS21 blade has its own address space, separate from the QS22, which introduces another level of hybridization into the node structure.

The triblades, as shown in Fig. 2, are the compute nodes that contain these computational chips and the heart of the system. They consist of three computational blades: one LS21 and two QS22s and a fourth expansion blade that handles the connectivity among the three computational blades as well as the connection to the CU IB switch.

The LS21 is a dual socket, dual core AMD based blade, so there are four AMD cores per triblade. The QS22 is a dual socket PowerXCell 8i (Cell BE chip) blade, so there are four Cell BE chips in the triblade to pair with the four AMD cores. The software model most commonly used is one MPI (message passing interface) task per AMD core-Cell BE chip pair. There is 16 GB of main storage on the LS21 and a total of 16 GB of main storage on the pair of QS22s. Having matched storage of 4 GB on each AMD core and each Cell BE chip simplifies designing and coding the applications that share data structures on each processing unit in the pair, since

there can be duplicate copies kept on each side and only updates to the structures need to be communicated

A Roadrunner Triblade node integrates Cell and Opteron blades:



Roadrunner Project, Los Alamos. Fig. 2 Triblade architecture

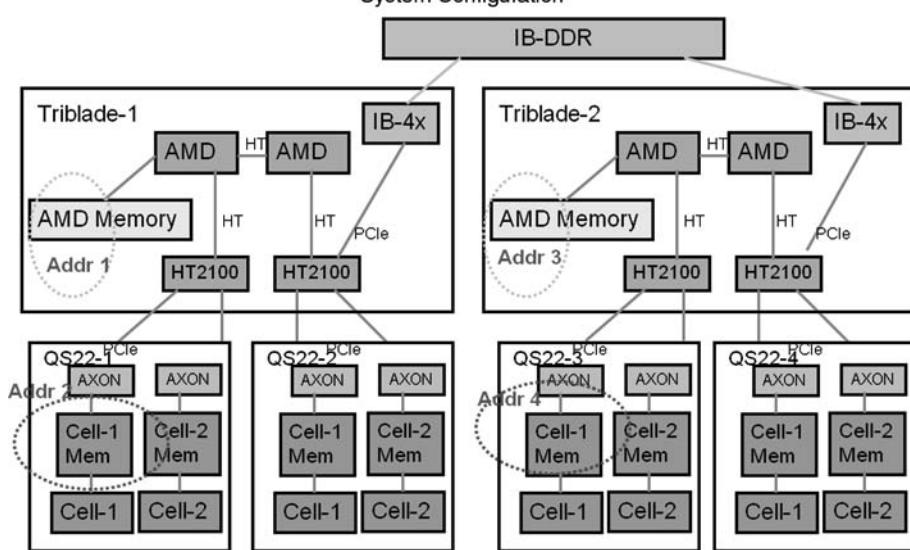
between the process running on the AMD core and the process running on the partner Cell BE chip.

Since the triblade structure presents dual address spaces, (one on the LS21 and one on the QS22) to each of the MPI tasks, as shown in Fig. 3, having bandwidth and latency efficient methods for transferring data between these address spaces is critical to the scaling success of the applications.

A significant architectural feature of the triblade is the set of four dedicated peripheral component interconnect express (PCIe)-x8 links that connect the four AMD cores on the LS21 to their partner Cell BE chips on the QS22s. All four links can transmit data in both directions simultaneously for moving Cell BE data structures up to the Opteron processors and Opteron structures down to the Cell BE chips.

A common application goal, not surprisingly, was to maximize the percentage of the work done by the Cell BE synergistic processing elements (SPEs) [6]. When the SPEs need to communicate boundary information with their logical neighbor SPEs in the problem space, that are located in physically different triblades, they need to send the data up to their partner AMD core, which in turn will send it, usually using MPI, to the AMD core that is the partner of the actual target SPE. The receiving AMD will then send the data down to the actual target SPEs. Having the four independent

#### System Configuration



Roadrunner Project, Los Alamos. Fig. 3 System configuration

PCIe links allows all four Cell BE chips to send the data to and from their partner AMD cores at the same time.

Figure 3 shows a logical picture of two triblades connected together with their IB link to show the different address spaces that exist and need to be managed by the application data flow. What is shown is one of the four pairs of address spaces on each of the triblades. Each AMD core and each Cell BE chip has its own logical address space. Since each AMD core is paired with its own Cell BE chip, there are four pairs of logical address spaces on a tribblade. Address space 1 is the AMD address space for Core 0 in the AMD chip on the left of tribblade 1. That core is partnered with Cell BE-1 on QS22-1 which owns address space 2. There are corresponding address spaces 3 and 4 on tribblade 2. There is a set of library functions that can be called from the application on the AMD or the Cell BE to transfer data between address spaces 1 and 2, and 3 and 4. OpenMPI is used to transfer data between the AMD address spaces 1 and 3 using typical MPI and IB methods. The difference between the standard MPI model and the tribblade MPI model is that in the tribblade model, an MPI task controls both a standard Opteron core and the heterogeneous Cell BE chip. This is similar in some ways to having multiple threads running under a single MPI task except that the address spaces need to be explicitly controlled and the cores are heterogeneous.

The ability to control the data flow both between the hybrid address spaces and within the Cell BE address space allows the application to optimize the utilization of the memory bandwidths in the system. As chip densities increase, but circuit and pin frequencies do not, utilizing pin bandwidths (both memory bandwidth and communication bandwidth) efficiently will become increasingly important. The Cell BE chip local stores, which act like software controlled cache structures, allow for some very efficient computational models. As an example, the DGEMM (double-precision general matrix multiply) routine that ran in the hybrid Linpack application [7], was over 99% efficient, and was the heart of the computational efficiency that led to the sustained petaflops achievement. The same software cache model applies to the separate address spaces that exist in the AMD and Cell BE chips within the MPI task.

## Technology Drivers

The ongoing reduction in the rate of processor frequency improvement has led to three fundamental issues in development to achieve dramatically better application performance: (1) Software complexity which is caused by the growing number of cores needed for ever greater performance, (2) Energy requirements driven by the increased number of cores and chips needed for performance improvements, and (3) RAS requirements driven by the ever increasing number of circuits and components in the larger systems. All three of these issues were important features of the Roadrunner project overall design and implementation.

## Software Complexity

For the last several decades, lithographic technology improvements guided by Dennard's CMOS Scaling Theory [14] in accordance with Moore's law [10] have not only provided circuit density improvements, but the shrinkage in the vertical dimension also provided an accompanying frequency increase. This meant that for applications to scale in performance, one did not need to modify the software to utilize additional cores or threads, since system performance improved simply because of the frequency scaling.

The elimination of future frequency scaling puts the burden of application performance improvement on the application software and programming environment, and requires a doubling of the number of cores applied to a problem in order to achieve a doubling of the peak performance available to the application. Since sustained application performance for strong scaling, and to a lesser extent weak scaling, is likely to be less than linear in the number of cores available, more than double the number of cores will usually be required to achieve double the sustained application performance.

At a nominal 1 GHz, effectively a million functional units are required for petascale sustained performance, and the number of cores will continue to grow as greater sustained performance is desired. Finding ways to cope with this fundamental issue, in a way that is tenable for both algorithm development and code maintenance, while still achieving reasonable fractions of nominal peak performance for the sustained performance on real (as opposed to test or benchmark) applications, was a major goal of the project. Since application scaling going to petascale and beyond was such a major focus,

considerable time and energy were spent on making sure that real applications would scale on the machine.

While high-level language constructs and advances, in and of themselves were not a major focus of the project, rethinking the applications themselves was. The LANL application and performance modeling teams, as well as the IBM application and performance modeling teams, analyzed how the construction of the base algorithm flow and the interaction with the projected hardware would work.

The IBM Linpack team modified Linpack to operate in the hybrid environment and modeling was done to optimize the structure of the code around the anticipated performance characteristics of the system. The workhorse of the sustained petaflops performance was the DGEMM algorithm on the Cell BE chip since it provided over 99% of the computation required by the overall solution of the Linpack problem. The AMD core provided the overall control flow for the application, performed the OpenMPI communication, and did some of the algorithmic steps that could be overlapped with the Cell BE DGEMM calculations. The final structure of the solutions was both hybrid (between the x86 and Cell BE) units and heterogeneous on the Cell BE itself [7].

Linpack was only one of the codes that were studied in great detail, and had the machine only been useful for a Linpack run, and not for critical science and DOE codes, it would not have been built. A full year was spent analyzing and testing pieces of several applications prior to making the decision to go forward with the actual machine manufacture and construction.

Los Alamos National Laboratory revamped several applications to the Roadrunner architecture. These applications represent a wide variety of workload types. Among the applications ported are [11]:

- VPIC – full relativistic, charge-conserving, 3D explicit particle-in-cell code.
- SPaSM – Scalable Parallel Short-range Molecular Dynamics code, originally developed for the Thinking Machines Connection Machine model CM-5.
- Milagro – Parallel, multidimensional, object-oriented code for thermal x-ray transport via implicit Monte Carlo methods on a variety of meshes.
- Sweep3D – Simplified 1-group 3D Cartesian discrete ordinates kernel representative of a neutron transport code.

The different applications employed multiple methods to exploit the capabilities of Roadrunner. Milagro used a host-centric accelerator offload model where the majority of the application was unchanged, (except for the control code which was modified due to the new hybrid structure of the triblade,) and selected computationally intensive portions were modified to move their most performance critical sections to the accelerator (Cell BE). VPIC employed an accelerator-centric approach where the majority of the program logic was migrated to the accelerator and the host systems and the InfiniBand interconnect served primarily to provide accelerator to accelerator message communications.

Early performance results on the applications showed speed-ups over an unaccelerated cluster in the range of 4–9 times depending on how much work could be moved to the Cell. Between 2 KLOC (thousand lines of code) and 33 KLOC needed to be modified to achieve these speed-ups. Some of the applications had obvious compute kernels that could be transferred to the Cell BE and those applications required fewer changes to the code than the cases where the compute parts were more distributed or where the control code itself had to be restructured, as was the case with Milagro. But, even in the case of Milagro, it was a small percentage of the code that was changed.

Understanding of how to construct applications and lay out data structures to achieve reasonable sustained scaled performance will be important to future machine designs as well as provide input and examples for the translation of high-level language constructs into operational code. One example of a hardware feature in the base architecture of the QS22 is the ability for software to control the Local Store memory traffic [5, 6]. Since memory bandwidth will be an important commodity to optimize around in the future processor designs, understanding how best to do data pipelining from an application point of view is important base data to have. Hardware controlled flow of memory pre-fetching (cache) is a good feature for processors to have in general, but optimal efficiency will require explicit application tuning of bandwidth utilization. Automating the software control based on algorithms, hints, or language constructs is something that can be added later to take some of the burden off of the software developers but for now, understanding how best to use features like these is more important.

## Energy

As was stated earlier, energy consumption and cooling of IT facilities are an increasing critical problem. The business-as-usual technology projections in 2010 indicate that by 2020 large-scale systems will be more expensive to operate than to acquire [12]. Due to the increase in power requirements as ultra-scale computing systems grow in computing capability, it is often the case that new facilities need to be built to house the anticipated supercomputing machines.

Another innovation in the Roadrunner project was minimizing the overall power required by using heterogeneous core types that optimize the power utilization, not by lowering the thread performance (frequency) but by eliminating unnecessary circuits in the SPEs themselves. As was discussed previously, this presents a different set of issues to the programmer. The Roadrunner system, using heterogeneous core types, uses fewer overall threads for the same performance than a homogeneous core machine using the same amount of power, but the core types need separate compiled code, and more explicit thought about where each piece of the algorithm should run.

## Heterogeneous Cores and Hybrid Architecture

Using heterogeneous core types on the PowerXCell8i was a way of dealing with the computational density and power efficiency. Using a hybrid node architecture that included a combination of the QS22s for computational efficiency and the LS21 for the general purpose computations and communications control was a way of overcoming the limitations of the PowerPC core on the Cell BE chip. One can easily imagine future supercomputers based on SoC designs that do not require the additional level of heterogeneity utilized in Roadrunner.

The PowerXCell8i has two core types, a general purpose embedded PowerPC core (the power processing element, PPE) which handles all of the operating system and hybrid communication tasks and eight specialized compute cores (the synergistic processing elements, SPEs). The SPEs take up significantly less chip area than the PowerPC core since they only need the facilities and instruction set to handle the intense computational tasks. This adds to the software complexity of the application algorithms because tasks must be assigned appropriately to the cores, and additional memory space management is required, but the results are well worth

it [6]. Software tools such as OpenMP and OpenCL are emerging which will help hide the heterogeneity from the application user, but the complexity will still exist in the system itself.

The Roadrunner solution was not only the first to break the sustained petaflops Linpack barrier and therefore first on the Top500 list in June 2008 but it was third on the June 2008 Green500 as well. The first and second places on that Green500 list went to QS22 only based systems, so the energy efficiency of using the heterogeneous core approach is obvious. It is a trend that will continue into the future as the industry works to figure out the best utilization of the increasing number of circuits available on chips while also making the computational efficiency of the circuits at the application level improve.

## Reliability, Availability, and Serviceability

As machines grow in size and complexity, and applications scale to use increasing amounts of hardware for long run times, the reliability and availability characteristics become increasingly important. For an application to make forward progress, it needs to either checkpoint its progress often enough or have a mechanism for migrating processes running on failing parts of the system onto other working hardware. The Roadrunner system chose to use the checkpoint/restart approach since it seemed the most generally practical for a machine of its size and for the types of applications that it would be running. High availability (HA) through failover system design, at the node or even cluster level, was not a cost-effective approach for the technologies that are employed, and checkpointing could be tolerated.

Of course, there are certainly parts of the system, like the memory DIMMs (dual in-line memory modules), that have error-correcting facilities as well as memory sparing capabilities, and these increase the inherent overall node availability. But making the nodes themselves redundant or failover capable outside of the checkpoint/restart process was not an objective.

Three technology developments whose impact on service were very critical to the overall success, and in particular the ability to build the entire system in a short time, are IBM BladeCenter technology, the Roadrunner triblade mechanical design, and the integrated optical cable that was used for the IB subsystem [9].

The IBM BladeCenter was designed from the outset to maximize serviceability and ease of upgradability. The modular nature of the BladeCenter chassis with the ability to just slide new blades into an existing and pretested infrastructure that has the system management and power management built in, and I/O connections that can stay in place as the blades are slid in and out, provides a very robust base for building large-scale systems. This was proven in practice when the petaflop barrier was broken only four days after the triblades for the final CU were delivered to the lab from manufacturing. They were plugged in, verified to be good, integrated in with the other CUs, and then the entire system turned over to the applications group in less than 48 h.

## Conclusion

Roadrunner has achieved each and every one of its original goals, and more. First and foremost, fully capable simulations from fluid mechanics to radiation transport, from molecular to cosmological scale, are running routinely and efficiently on this system. The development of these new capabilities provides a roadmap for developing exascale applications over the next decade.

While there will always be challenges to getting to the next orders of magnitude in computing performance, the Roadrunner project collaboration between LANL and IBM was the first to recognize that a new paradigm was necessary for high-performance computing and to address the energy efficiency and computational performance of a new generation of these systems.

There was concentrated effort on the software algorithm structures needed to implement heterogeneous and hybrid computing models at this scale. It has shown that Open Source components can be used to advantage when the technical challenges attract interested and motivated participants.

The use of a modular building block approach to hardware design, including the BladeCenter management and control infrastructure, provided a very stable base for doing the software experimentation. With some of the breakthroughs that this program produced, the HPC community is now well positioned to solve, or help solve, the scaling issues that will produce major breakthroughs in business, science, energy, and social arenas.

The HPC community needs to be thinking “Ultra Scale” computing in order to harness the emerging technological advances and head into the exascale arena with the momentum that IBM and LANL built crossing the petaflops boundary.

## Bibliography

1. Los Alamos National Laboratory, LANL Roadrunner. <http://www.lanl.gov/roadrunner/>
2. Voltaire Inc. Infiniband Products. <http://www.voltaire.com/Products/Infiniband>
3. xCAT: Extreme Cloud Administration Tool 2.0. <http://xcat.sourceforge.net/>
4. IBM Corporation, IBM BladeCenter LS21. <http://www-03.ibm.com/systems/bladecenter/hardware/servers/ls21/index.html>
5. Vogt J-S, Land R, Boettiger H, Krnjajic Z, Baier H (2009) IBM BladeCenter QS22: design, performance and utilization in hybrid computing systems. IBM J Res Dev 53(5) Paper 3:1–14
6. Gschwind M (2009) Integrated execution: a programming model for accelerators. IBM J Res Dev 53(5) Paper 4:1–14
7. Kistler M, Gunnels J, Brokenshire D, Benton B (2009) Programming the Linpack benchmark for Roadrunner. IBM J Res Dev 53(5) Paper 9:1–11
8. IBM Corporation, IBM BladeCenter. <http://www-03.ibm.com/systems/bladecenter/>
9. EMCORE Corporation, Infiniband Optics Cable. [http://www.emcore.com/fiber\\_optics/emcoreconnects](http://www.emcore.com/fiber_optics/emcoreconnects)
10. Moore G (1965) Cramming more components onto integrated circuits. Electronics 38(8):114–117
11. Los Alamos National Laboratory, LANL Applications. <http://lanl.gov/roadrunner/rseminars.shtml>
12. Humphreys J, Scaramella J IDC The Impact of Power and Cooling on Data Center Infrastructure, Document #201722, May 2006. [http://www-03.ibm.com/systems/resources/systems\\_z\\_pdf\\_IDC\\_ImpactofPowerandCooling.pdf](http://www-03.ibm.com/systems/resources/systems_z_pdf_IDC_ImpactofPowerandCooling.pdf)
13. Green500 List. <http://www.green500.org/>
14. Dennard R, Gaenslen F, Rideout V, Bassous E, LeBlanc A (1974) Design of ion-implanted MOSFETs with very small physical dimensions. IEEE J Solid State Circuits SC-9(5):256–268
15. IBM Corporation, IBM Deep Computing. <http://www-03.ibm.com/systems/deepcomputing/>

## Router Architecture

### ► Switch Architecture

## Router-Based Networks

- ▶ Networks, Direct
- ▶ Networks, Multistage

## Routing (Including Deadlock Avoidance)

PEDRO LÓPEZ

Universidad Politécnica de Valencia, Valencia, Spain

### Synonyms

Forwarding

### Definition

The routing method used in an interconnection network defines the path followed by packets or messages to communicate two nodes, the source or origin node, which generates the packet, and the destination node, which consumes the packet. This method is often referred to as the routing algorithm.

### Discussion

#### Introduction

Processing nodes in a parallel computer communicate by means of an interconnection network. The interconnection network is composed of switches interconnected by point to point links. Topology is the representation of how these switches are connected. Direct or indirect regular topologies are often used. In direct topologies every network switch has an associated processing node. In indirect topologies, only some switches have processing nodes attached to them. The  $k$ -ary  $n$ -cube (which includes tori and meshes) and  $k$ -ary  $n$ -tree (an implementation of fat-trees) are the most frequently used direct and indirect topologies, respectively. In the absence of a direct connection among all network nodes, a routing algorithm is required to select the path packets must follow to communicate every pair of nodes.

#### Taxonomy of Routing Algorithms

Routing algorithms can be classified according to several criteria. The first one considers the place where

decisions are made. In source routing, the full path is computed at the source node, before injecting the packet into the network. The path is stored as a list of switch output ports in the packet header, and intermediate switches are configured according to this information. On the contrary, in distributed routing, each switch computes the next link that will be used by the packet while the packet travels across the network. By repeating this process at each switch, the packet reaches its destination. The packet header only contains the destination node identifier. The main advantage of source routing is that switches are simpler. They only have to select the output port for a packet according to the information stored in its header. However, since the packet header itself must be transmitted through the network, it consumes network bandwidth. On the other hand, distributed routing has been used in most hardware routers for efficiency reasons, since packet headers are more compact. Also, it allows more flexibility, as it may dynamically change the path followed by packets according to network conditions or in case of faults.

Routing algorithms can be also classified as deterministic or adaptive. In deterministic routing, the path followed by packets exclusively depends on their source and destination nodes. In other words, for a given source-destination pair, they always choose the same path.

On the contrary, in adaptive routing, the path followed by packets also depends on network status, such as node or link occupancy or other network load information. Adaptive routing increases routing flexibility, which allows for a better traffic balance on the network, thus improving network performance.

Deterministic routing algorithms are simpler to implement than adaptive ones. On the other hand, adaptive routing introduces the problem of out of order delivery of packets, as two consecutive packets sent from the same source to the same destination may follow different paths. An easy way of increasing the number of routing options is virtual channel multiplexing [2]. In this case, each physical link is decomposed into several virtual channels, each one with its own buffer, which can be separately reserved. Physical link is shared among all the virtual channels using time division multiplexing. Adaptive routing algorithms may provide all the feasible routing options

to forward the packet towards its destination (fully adaptive routing) or only a subset (partially adaptive routing).

If adaptive routing is used, the selection function chooses one of the feasible routing options according to some criteria. For instance, in case of distributed adaptive routing, the selection function may select the link with the lowest buffer occupation or with the lowest number of busy virtual channels.

A routing algorithm is minimal if the paths it provides are included in the set of shortest or minimal paths from source to destination. In other words, with minimal routing, every link crossed by a packet reduces its distance to destination. On the contrary, a non-minimal routing algorithm allows packets to use paths longer than the shortest ones. Although these routing algorithms are useful to circumvent congested or faulty areas, they use more resources than strictly needed, which may negatively impact on performance. Additionally, paths having an unbounded number of allowed non-minimal hops might result in packets never reaching their destination, situation that is referred to as livelock.

Routing algorithms can be implemented in several ways. Some routers use routing tables with a number of entries equal to the number of destinations. With source routing, these tables are associated with each source node, and each entry contains the whole path towards the corresponding destination. With distributed routing, these tables (known as forwarding tables) are associated to each network switch, and contain the next link a packet destined to the corresponding entry must follow. With a single entry per destination, only deterministic routing is allowed. The main advantage of table-based routing is that any topology and any routing algorithm can be used in the network. However, routing based on tables suffers from a lack of scalability. The size of the table grows linearly with network size ( $O(N)$  storage requirements), and, most important, the time required to access the table also depends on network size.

An alternative to tables is to place a specialized hardware at each node that implements a logic circuit and computes the output port to be used as a function of the current and destination nodes and the status of the output ports. The implementation is very efficient in terms of both area and speed, but the algorithm is specific to

the topology and to the routing strategy used on that topology. This is the approach used for the fixed regular topologies and routing algorithms used in large parallel computers.

There are hybrid implementations, such as the Flexible Interval Routing (FIR) [11] approach, which defines the routing algorithm by programming a set of registers associated to each output port. The strategy is able to implement the most commonly-used deterministic and adaptive routing algorithms in the most widely used regular topologies, with  $O(\log(N))$  storage requirements.

## Deadlock Handling

A deadlock in an interconnection network occurs when some packets cannot advance toward their destination because all of them are waiting on one another to release resources (buffers or links). In other words, the involved packets request and hold resources in a cyclic fashion. There are three strategies for deadlock handling: deadlock prevention, deadlock avoidance, and deadlock recovery.

Deadlock prevention incurs in a significant overhead and is only used in circuit switching. It consists of reserving all the required resources before starting transmission. In deadlock avoidance, resources are requested as packets advance but routing is restricted in such a way that there are no cyclic dependencies between channels. Another approach consists of allowing the existence of cyclic dependencies between channels while providing some escape paths to avoid deadlock, therefore increasing routing flexibility. Deadlock recovery strategies allow the use of unrestricted fully adaptive routing, potentially outperforming deadlock avoidance techniques. However, these strategies require a deadlock detection mechanism [13] and a deadlock recovery mechanism [1, 12] that is able to recover from deadlocks. If a deadlock is detected, the recovery mechanism is triggered, resolving the deadlock. Progressive recovery techniques deallocate buffer resources from other “normal” packets and reassign them to deadlocked packets for quick delivery. Disha [1] and software-based recovery [12] are examples of this approach. Regressive techniques deallocate resources from deadlocked packets by killing and later re-injecting them at the original source router (i.e., abort-and-retry).

In deadlock-avoidance, the routing algorithm restricts the paths allowed by packets to only those ones that keep the global network state deadlock-free. Dally [2] proposed the necessary and sufficient condition for a deterministic routing algorithm to be deadlock-free. Based on this condition, he defined a channel dependency graph (CDG) and established a total order among channels. Routing is restricted to visit channels in order, to eliminate cycles in the CDG. The CDG associated to a routing algorithm on a given topology is a directed graph, where the vertices are the network links and the edges join those links with direct dependencies between them, i.e., those adjacent links that could be consecutively used by the routing algorithm. The routing algorithm is deadlock free if and only if the CDG is acyclic. To design deadlock-free routing algorithms based on this idea while keeping network connectivity, it may be required that physical channels are split into several virtual channels, and the ordering is performed among the set of virtual channels. Although this idea was initially proposed for deterministic routing, it has been also applied to adaptive routing. For instance, in the turn model [10], a packet produces a turn when it changes direction, usually changing from one dimension to another in a mesh. Turns are combined into cycles. Prohibiting just enough turns to break all the cycles prevents deadlock.

Although imposing an acyclic CDG avoids deadlock, it results in poor network performance, as routing flexibility is strongly constrained. Duato [5] demonstrates that a routing algorithm can have cycles in the CDG while remaining deadlock-free. This allows the design of deadlock-free routing algorithms without significant constraints imposed on routing flexibility. The key idea that supports deadlock freedom despite the existence of cyclic dependencies is the concept of escape path. Packets can be routed without restrictions, provided that there exists a subset of network links – the escape paths – without cyclic dependencies, which allow every packet to escape from a potential cycle. An important characteristic of the escape paths is that it must be possible to deliver packets from any point in the network to their corresponding destination using the escape paths alone. However, this does not imply that a packet that used an escape path at some router must continue using escape paths until delivered. Indeed, packets are free to leave the escape paths at any point in

the network and continue using full routing flexibility until delivered.

These ideas can be stated more formally as follows. A routing subfunction R1 associated to the routing function R supplies a subset of the links provided by R (the set of escape channels), therefore restricting its routing flexibility, while still being able to deliver packets from any valid location in the network to any destination. The concept of indirect dependency between two escape channels is defined as follows. A packet may reserve a set of adjacent channels  $c_i, c_{i+1}, \dots, c_{k-1}, c_k$ , where only  $c_i$  and  $c_k$  are provided by R1, but  $c_{i+1}, \dots, c_{k-1}$  are not. The extended CDG for R1 has as vertices the escape channels, and as edges both the direct and indirect dependences between escape channels.

The theory states that a routing function R is deadlock-free if there exists a routing subfunction R1 without cycles in its extended CDG. A simple way to design adaptive routing algorithms based on the theory is to depart from a well-known deterministic deadlock-free routing function, adding virtual channels in a regular way. The new additional channels can be used for fully adaptive routing. As mentioned above, when a packet uses an escape channel at a given node, it can freely use any of the available channels supplied by the routing function at the next node.

Duato also developed necessary and sufficient conditions for deadlock-free adaptive routing under different switching techniques, and for different sets of input data for the routing function. All of those theoretical extensions are based on the principle described above.

### Routing in $k$ -ary $n$ -cubes

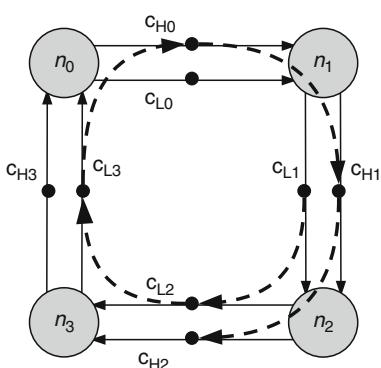
A  $k$ -ary  $n$ -cube is a  $n$ -dimensional network, with  $k$  nodes in each dimension, connected in a ring-fashion. Every node is connected to  $2n$  neighbors. The number of nodes is  $N = k^n$ . This topology is often referred to as a  $n$ -dimensional torus, and it has been used in some of the largest parallel computers, such as the Cray T3E or the IBM BlueGene (both of them use a 3D-torus). The  $n$ -dimensional mesh is a particular case where the nodes of each dimension are connected as a linear array (i.e., there are not wraparound connections). The symmetry and regularity of the  $k$ -ary  $n$ -cube simplify network implementation and packet routing as the movement of a packet along a given network dimension does not

modify the number of remaining hops in any other dimension toward its destination.

In a mesh, dimension-order routing (DOR) avoids deadlocks (and livelocks) by allowing only the use of minimal paths that cross the network dimensions in some total order. That is, the routing algorithm does not allow the use of links of a given dimension until no other links are needed by the packet in all of the preceding dimensions to reach its destination. In a 2D-mesh, this is easy to achieve by crossing dimensions in XY order.

This simple routing algorithm is not deadlock-free in a torus. The wraparound links create a cycle among the nodes of each dimension. This is easily solved by multiplexing each physical channel into two virtual channels (H -high- and L -low-), and restricting routing in such a way that H channels can only be used if the coordinate of the destination node in the corresponding dimension is higher than the current one and vice versa. Consider a 4-node unidirectional ring (a 4-ary 1-cube) with nodes  $n_i$ ,  $i = 0, 1, 2, 3$  and two channels connecting each pair of adjacent nodes. Let  $c_{Hi}$  and  $c_{Li}$ ,  $i = 0, 1, 2, 3$  be output channels of node  $n_i$ . Figure 1 shows the CDG corresponding to the routing algorithm presented above. As there are not cycles in the channel dependency graph, it is deadlock free.

An adaptive routing algorithm for tori can be easily defined by merely adding a new virtual channel to each physical channel (the A -adaptive- channel). This new channel can be freely used, even to cross network dimensions in any order, thus providing fully adaptive routing on the  $k$ -ary  $n$ -cube. In this routing algorithm,



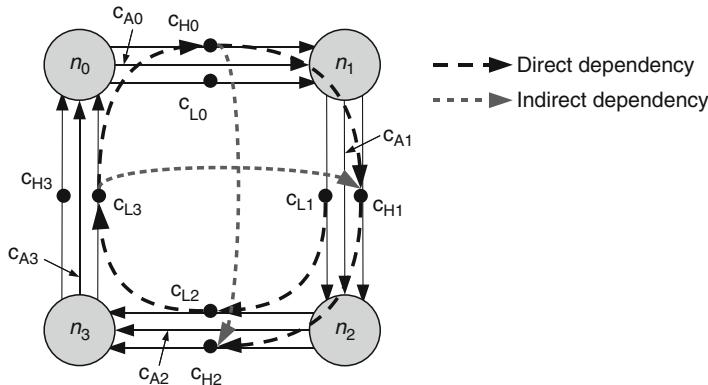
**Routing (Including Deadlock Avoidance).** Fig. 1 Channel dependency graph for a 4-ary 1-cube

the H and L channels belong to the set provided by the routing subfunction. Figure 2 shows the extended CDG for this routing algorithm in a 4-node unidirectional ring, which is acyclic, and thus the routing algorithm is deadlock-free. This routing algorithm will provide several routing options, depending on the relative locations of the node that currently stores the packet and the destination node. For instance, in a 2D-torus, if both X and Y channels remain to be crossed, two adaptive virtual channels in both dimensions and one escape channel (the H or L in the X dimension) will be provided. The selection function is in charge of making the final choice, selecting one of the links also considering network status. A simple selection function may be based on assigning static priorities to virtual channels, returning the first free channel according to this order. Priorities could be assigned to give more preference to adaptive channels. More sophisticated selection functions could be also used, such as selecting a channel of the least multiplexed physical channel to minimize the negative effects of virtual channel multiplexing.

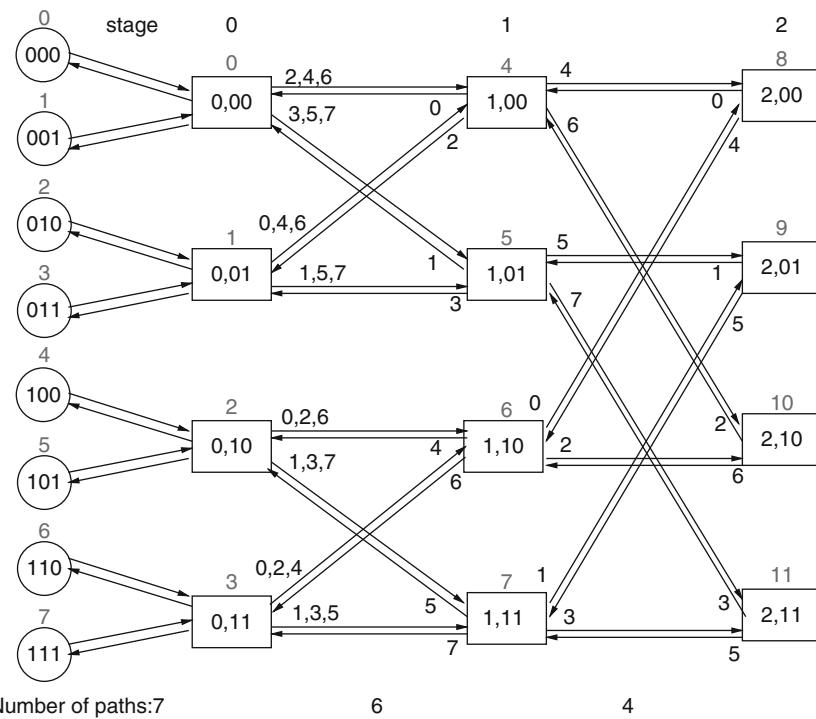
### Routing in $k$ -ary $n$ -trees

Clusters of PCs are being considered as a cost-effective alternative to small and medium scale parallel computing systems. These machines use any of the commercial high-performance switch-based point-to-point interconnects. Multistage interconnection networks (MINs) are the most usual choice. In particular, fat-trees have risen in popularity in the past few years (for instance, Myrinet, InfiniBand, Quadrics).

A fat-tree is based on a complete tree that gets thicker near the root. Processing nodes are located at the leaves. However, assuming constant link bandwidth, the number of ports of the switches increases as we approach the root, which makes the physical implementation unfeasible. For this reason, some alternative implementations have been proposed in order to use switches with fixed arity. In  $k$ -ary  $n$ -trees, bandwidth is increased as we approach the root by replicating switches. A  $k$ -ary  $n$ -tree (see Fig. 3) has  $n$  stages, with  $k$  (arity) links connecting each switch to the previous or to the next stage. A  $k$ -ary  $n$ -tree is able to connect  $N = k^n$  processing nodes using  $nk^{n-1}$  switches.



Routing (Including Deadlock Avoidance). Fig. 2 Extended channel dependency graph for a 4-ary 1-cube



Routing (Including Deadlock Avoidance). Fig. 3 Load-balanced deterministic routing in a 2-ary  $n$ -tree

In a  $k$ -ary  $n$ -tree, minimal routing from a source to a destination can be accomplished by sending packets upwards to one of the nearest common ancestors of the source and destination nodes and then, from there, downwards to destination. When crossing stages in the upward direction, several paths are possible, thus providing adaptive routing. In fact, each switch can select

any of its up output ports. Once a nearest common ancestor has been reached, the packet is turned around and sent downwards to its destination and just a single path is available.

A deterministic routing algorithm for  $k$ -ary  $n$ -trees can be obtained by reducing the multiple ascending paths in a fat-tree to a single one for each source-

destination pair. The path reduction should be done trying to balance network link utilization. That is, all the links of a given stage should be used by a similar number of paths. A simple idea is to shuffle, at each switch, consecutive destinations in the ascending phase [8]. In other words, consecutive destinations in a given switch are distributed among the different ascending links, reaching different switches in the next stage. [Figure 3](#) shows the destination node distribution in the ascending and descending links of a 2-ary 3-tree following this idea. In the figure, each ascending link has been labeled with the destinations it can forward to. As can be seen, packets destined to the same node reach the same switch at the last stage, independently of their source node. Each switch of the last stage receives packets addressed only to two destinations, and packets destined to each one are forwarded through a different descending link (as there are two descending links per switch). Therefore, this mechanism efficiently distributes the traffic destined to different nodes, and balances load across the links.

most popular topology agnostic routing scheme used in cluster networks. Routing is based on an assignment of direction labels (up or down) to the links in the network by building a breadth first spanning tree (BFS). Cyclic channel dependencies are avoided by prohibiting messages to traverse a link in the up direction after having traversed one in the down direction. Although simple, this approach generates a network unbalance because a high percentage of traffic is forced to cross the root node. Derived from  $\text{up}^*/\text{down}^*$ , some improvements were proposed, such as assigning directions to links by computing a depth first spanning tree (DFS), the Flexible Routing scheme, which introduced unidirectional routing restrictions to break cycles in each direction at different positions, or the Segment-based Routing algorithm, which splits a topology into subnets, and subnets into segments. This allows placing bidirectional turn restrictions locally within a segment, which results in a larger degree of freedom.

Most of these routing algorithms do not guarantee that all packets will be routed through a minimal path. This causes an increment in the packet latency and an inefficient use of network resources, affecting the overall network performance. The Minimal Adaptive routing algorithm, which is based on Duato's theory, adds a new virtual channel which provides minimal routing, keeping the paths provided by  $\text{up}^*/\text{down}^*$  as escape paths. The In-Transit Buffers mechanism [7] also provides minimal routing for all the packets. To avoid deadlocks, packets are ejected from the network, temporarily stored in the network interface card at some intermediate hosts and later re-injected into the network. However, this mechanism requires some support at NICs and at least one host to be attached to every switch in the network, which cannot always be guaranteed. Other approaches make use of virtual channels to route all the packets through minimal paths while still guaranteeing deadlock freedom. For instance, Layered Shortest Path (LASH) guarantees deadlock freedom by dividing the physical network into a set of virtual networks using separate virtual channels. Minimal paths between every source-destination pair of hosts are spread onto these layers, such that each layer becomes deadlock free. In Transition Oriented Routing (TOR), unlike the LASH routing,  $\text{up}^*/\text{down}^*$  is used as a baseline routing algorithm in order to decide when to change to a new virtual net

## Routing in Irregular Networks. Agnostic Routing

As stated above, direct topologies and multistage networks are often used when performance is the primary concern. However, in the presence of some switch or link failures, a regular network will become an irregular one. In fact, most of the commercially available interconnects for clusters support irregular topologies. An alternative for tolerating faults in regular networks without requiring additional logic is the use of generic or topology agnostic routing algorithms. These routing algorithms can be applied to any topology and provide a valid and deadlock free path for every source-destination pair of nodes in the network (if they are physically connected). Therefore, in the presence of any possible combination of faults, a topology agnostic routing algorithm provides a valid solution for routing.

An extensive number of topology agnostic routing strategies have been proposed. They differ in their goals and approaches (e.g., obtaining minimal paths, fast computation time) and the resources they need (typically virtual channels). The oldest topology agnostic routing algorithm is  $\text{up}^*/\text{down}^*$  [14], being the

work (when a forbidden transition down→up appears). To remove cyclic channel dependencies, virtual networks are crossed in increasing order, thus guaranteeing deadlock freedom. However, due to the large number of routing restrictions imposed by up\*/down\*, providing minimal paths among every pair of hosts may require a high number of virtual channels.

## Related Entries

- ▶ [Clusters](#)
- ▶ [Deadlocks](#)
- ▶ [Flow Control](#)
- ▶ [Infiniband](#)
- ▶ [Interconnection Networks](#)
- ▶ [Myrinet](#)
- ▶ [Network of Workstations](#)
- ▶ [Networks, Direct](#)
- ▶ [Networks, Fault-Tolerant](#)
- ▶ [Networks, Multistage](#)
- ▶ [Switch Architecture](#)
- ▶ [Switching Techniques](#)

## Bibliographic Notes and Further Reading

Dally's [4] and Duato's [6] books are excellent textbooks on Interconnection Networks and contain abundant material about routing. The sources of the theories about deadlock-free routing can be found in [2] and [5]. Although virtual channels were introduced in [2], they were formally proposed in [3] as "virtual channel flow control." The Turn-model, which allows adaptive routing in tori without virtual channels was proposed in [10]. The selection function has some impact on network performance. Several selection functions have been proposed. For instance, [9] proposed a set of possible selection functions in the context of fat-trees. Deadlock-recovery strategies are based on the assumption that deadlocks are rare. This assumption was evaluated in [15], where the frequency of deadlocks was measured. Disha and software-based recovery, two progressive deadlock-recovery mechanisms, were proposed in [1] and [12], respectively. To make deadlock-recovery feasible, an efficient deadlock detection mechanism that matches the low deadlock occurrence must be used, such as the FC3D deadlock-detection mechanism proposed in [13]. FIR [11] is an alternative implementation to algorithmic and

table-based routing that could be used in commercial switches. The load-balanced deterministic routing algorithm described for fat-trees can be found in [8]. Up\*/down\* routing was initially proposed for Autonet [14]. The ITB mechanism [7] was proposed to improve performance of source routing in the context of Myrinet networks.

## Bibliography

1. Anjan KV, Pinkston TM (1995) DISHA: a deadlock recovery scheme for fully adaptive routing. In: Proceedings of the 9th international parallel processing symposium, Santa Barbara, CA, pp 537–543
2. Dally WJ (1992) Virtual-channel flow control. IEEE Trans Parallel Distributed Syst 3(2):194–205
3. Duato J (1993) A new theory of deadlock-free adaptive routing in wormhole networks. IEEE Trans Parallel Distributed Syst 4(12):1320–1331
4. Dally WJ, Seitz CL (1987) Deadlock-free message routing in multiprocessor interconnection Networks. IEEE Trans Comput C-36(5):547–553
5. Dally WJ, Towles B (2004) Principles and practices of interconnection networks. Morgan Kaufmann, San Francisco, CA
6. Duato J, Yalamanchili S, Ni LM (2003) Interconnection networks: an engineering approach. Morgan Kaufmann, San Francisco, CA
7. Flich J, López P, Malumbres MP, Duato J (2002) Boosting the performance of myrinet networks. IEEE Trans Parallel Distributed Syst 13(11):1166–1182
8. Gómez C, Gilabert F, Gómez ME, López P, Duato J (2007) Deterministic vs. adaptive routing in fat-trees. In: Proceedings of the 2007 international parallel and distributed processing symposium, IEEE Computer Society Press, Long Beach, CA
9. Gilabert F, Gómez ME, López P, Duato J (2006) On the influence of the selection function on the performance of fat-trees. Lecture Notes in Computer Science 4128, Springer, Berlin
10. Glass CJ, Ni LM (1992) The turn model for adaptive routing. In: Proceedings of the 19th international symposium on computer architecture, ACM, New York, pp 278–287
11. Gómez ME, López P, Duato J (2006) FIR: an efficient routing strategy for tori and meshes. J Parallel Distributed Comput Elsevier 66(7):907–921
12. Martínez JM, López P, Duato J (2001) A cost-effective approach to deadlock handling in wormhole networks. IEEE Trans Parallel Distributed Syst 12(7):716–729
13. Martínez JM, López P, Duato J (2003) FC3D: flow control based distributed deadlock detection mechanism for true fully adaptive routing in wormhole networks. IEEE Trans Parallel Distributed Syst 14(8):765–779
14. Schroeder MD et al (1990) Autonet: a high-speed, self-configuring local area network using point-to-point links. Technical report SRC research report 59, DEC, April 1990

15. Warnakulasuriya S, Pinkston TM (1997) Characterization of deadlocks in interconnection Networks. In: Proceedings of the 11th international parallel processing symposium, IEEE Computer Society, Washington, DC, pp 80–86

## R-Stream Compiler

BENOIT MEISTER, NICOLAS VASILACHE, DAVID WOHLFORD, MUTHU MANIKANDAN BASKARAN, ALLEN LEUNG, RICHARD LETHIN  
Reservoir Labs, Inc., New York, NY, USA

### Definition

R-Stream (R-Stream is a registered trademark of Reservoir Labs, Inc.) is a source-to-source, auto-parallelizing compiler developed by Reservoir Labs, Inc. R-Stream compiles programs in the domains of high-performance scientific (HPC) and high-performance embedded computing (HPEC), where loop nests, dense matrices, and arrays are the common idioms. It generates *mapped* programs, i.e., optimized programs for parallel execution on the target architecture. R-Stream targets modern, heterogeneous, multi-core architectures, including multiprocessors with caches, systems with accelerators, and distributed memory architectures that require explicit memory management and data movement. The compiler accepts the C language as input. Depending on the target platform, the compiled output program can be in C with the appropriate target APIs or annotations for parallel execution, other data parallel languages such as CUDA for GPUs, or dataflow assembly for FPGA targets.

### History

DARPA funded R-Stream development in the Polymorphous Computer Architecture (PCA) research program, starting in 2002. The PCA program developed new programmable computer architectures for approaching the energy efficiency of application-specific fixed function processors, particularly those for advanced signal processing in radars. These new PCA chips included RAW [36], TRIPS [34], Smart Memories [26], and Monarch [32]; these chips innovated architecture features for energy efficiency, such

as manycores, heterogeneity, mixed execution models, and explicit memory communications management. DARPA wanted R-Stream for programming the architectural space spanned by these chips from a single high-level and target-independent source. Since the PCA architectures uniformly presented a very high ratio of on-chip computational rates to off-chip bandwidth, R-Stream research prioritized transformations that increased arithmetic intensity (the ratio of computation to communication) in concert with parallelization, and on creating mappings that executed the program in a streaming manner – hence the name of the compiler, R-Stream (the “R” stands for Reservoir).

As the R-Stream compiler team refined the mapping algorithms, the conception of what the high-level, machine independent programming language should be also evolved. Streaming as an *execution model* had been well established for many decades, as the dominant execution paradigm for programming digital signal processors (DSPs): software-pipelining of computation with direct memory access (DMA). The PCA research program investigated whether streaming should be considered as the *programming model*. Initially, two streaming-based research programming languages, Stanford Brook [9] and MIT StreamIt [21], were supported in the PCA program. These languages provide means for expressing computations as aggregate operations (Brook calls them *kernels*; StreamIt calls them *filters*) on linear streams of data. However, as more understanding of the compilation process was acquired, the R-Stream compiler team started to move away from streaming as being the right expression form for algorithms for several reasons:

1. The expressiveness of the streaming languages was limited. They were biased toward one-dimensional streams, but radar signal processing involves multi dimensional data “cubes” processed along various dimensions [33].
2. Transformations for compiling streaming languages are isomorphic to common loop transformations – e.g., parallelization, tiling, loop fusion, index set splitting, array expansion/contraction, and so forth.
3. While streaming notations were found to sometimes be a good shorthand for expression of a particular mapping of an algorithm to a particular architecture, that particular mapping would not run well

on another architecture without significant transformation. The notation and idioms of the streaming languages actually frustrate translation, because a compiler would have to undo them to extract semantics for remapping.

Consequently, the R-Stream team moved away from streaming as a programming paradigm to instead supporting input programs (mainly loops) written in the C language: C can more succinctly express the multi-dimensional radar algorithms (compared to the streaming languages); the basic theory of loop optimization for C was well understood; and C can be written in a textbook manner with the semantics plain. However, the compilation challenges coming from the architectural features in the PCA chips were open problems. To attack these problems, the R-Stream team adopted and extended the powerful polyhedral framework for loop optimization.

PCA features are now in commercial computing chips. Programming such chips is complex, requiring more than just parallelization. Thus, the R-Stream compiler is being used by programmers and researchers to map C to chips such as Cell, GPGPUs, Tilera, and many-core SMP. It is also being extended in research on compilation to ExaScale supercomputers. R-Stream is available commercially and also licensed in source form to research collaborators.

## Design Principles

R-Stream performs high-level automatic parallelization. The high-level mapping tasks that R-Stream performs include parallelism extraction, locality improvement, processor assignment, managing the data layout, and generating explicit data movements. R-Stream takes as input sequential programs written in C, with the kernels to be optimized marked with a simple pragma, but with all mapping decisions for the target automated.

Low-level or machine-level optimizations, such as instruction selection, instruction scheduling, and register allocation, are left to an external *low-level compiler* that takes the mapped source produced by R-Stream.

R-Stream currently performs mostly static mapping for all its target architectures. Resource and scheduling decisions are computed statically at compile time, with very little dynamic decisions done at runtime. For all architectures, the mapping is also “bare-metal,” with

only a small dedicated runtime layer that is specialized to each architecture.

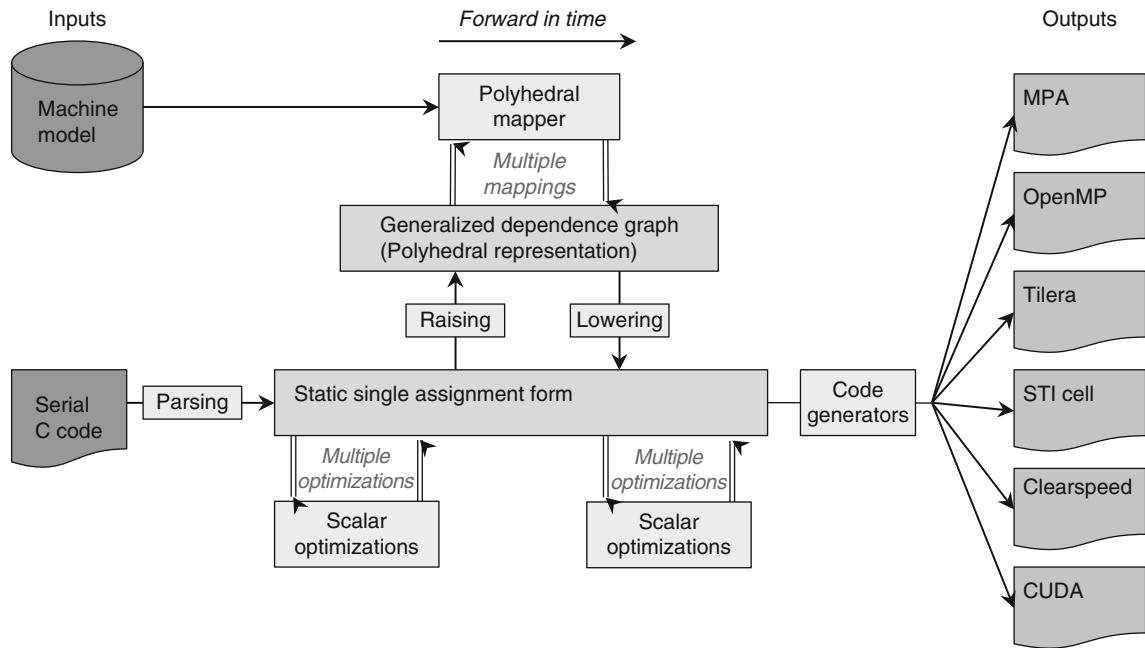
## Using R-Stream

Programmers write succinct loop nests in sequential ANSI C. The loop nest can be imperfect but must fit within an *extended static control program* form. That is, the loops have affine parametric extents and static integer strides. Multidimensional array references must be affine functions of loop index variables and parameters. Certain dynamic control features are allowed, such as conditionals that are affine functions of index variables and parameters. Programs that are not directly within the affine static control program form can be handled by wrapping the code in an abstraction layer and providing a compliant image function in C that models the effects of the wrapped function.

The programmer indicates the region to be mapped with a simple single pragma. Again, R-Stream does not require the programmer to indicate *how* this mapping should proceed: not through any particular idiomatic correspondence of the loop nest to particular hardware features, nor through detailed specifications encoded in pragma. R-Stream *automatically* determines the mapping, based on the target machine and emits transformed code. For example, a loop nest expressed in ANSI C can be rendered into optimized CUDA [1].

## Architecture

R-Stream is built on a set of independent and reusable optimizations on two intermediate representations (IRs). Unlike many source-to-source transformation tools, R-Stream shuns ad hoc pattern-matching-based transformations in favor of mathematically sound transformations, in particular dependence- and semantics-based transformations in the polyhedral model. The two intermediate representations are designed to support this view. The first intermediate representation, called Sprig, is used in the scalar optimizer. It is based on static single assignment (SSA) form [14], similar to Click’s graph-based IR [13], augmented with operators from value dependence graph [41] for expressing state dependencies. The second intermediate representation is the *generalized dependence graph* (GDG), the representation used in the polyhedral mapper. Conversions from the SSA representation to the GDG representation are performed by the *raising* phase,



**R-Stream Compiler.** Fig. 1 R-Stream's architecture

and the converse by the *lowering* phase. These components are shown as different parts of the R-Stream compiler flow in Fig. 1.

### Scalar Optimizer

The main task of the scalar optimizer is to simplify and normalize the input source so that subsequent analysis of the program in the raising phase can proceed. The scalar optimizer also runs after mapping and lowering, to remove redundancies and simplify expressions that result from the polyhedral scanning phases. The scalar optimizer provides traditional optimizations.

### Syntax Recovery

The output of R-Stream has to be processed by low-level compilers corresponding to the target architectures. These low-level compilers often have trouble compiling automatically generated programs, even if the code is semantically legal, because they pattern-match optimizations to the code idioms that humans typically write. For example, many low-level compilers cannot perform loop optimizations on loops that are expressed using gotos and labels. Thus, to ensure that the output

code can be effectively optimized by the low-level compiler, R-Stream's code generation back-end performs syntax recovery to convert a program in the scalar representation into idioms that look human-like. The engineering challenge is that R-Stream is designed to represent and transform programs in semantics-based IRs, far from the original syntax and language. Source reconstruction is more complex than simple unparsing. R-Stream uses an algorithm using early coalescing of  $\phi$ -functions to exit SSA, extensions to Cifuentes [10, 11] to detect high-level statement forms, and then localized pattern matching to generate syntax. Sprig also represents types in terms of the input source language, to allow for faithful reproduction of those types at output.

### Machine Model

R-Stream supports Cell, Tilera, GPUs, ClearSpeed, symmetric multiprocessors, and FPGAs as targets. These are rendered and detailed in machine models (MM), expressed in an XML-based description language. The target description is an entity graph connecting physical components that implicitly encodes the system capabilities and execution models that can be exploited by the mapper.

The nodes in the entity graph are processors, memories, data links, and command links. Processor entities are computation engines, including scalar processors, SIMD processors, and multiprocessors. These are structured as a multi dimensional grid geometry. Memory entities are data-caches, instruction-caches, combined data and I-caches, and main and scratchpad memories. There are two types of edges in the graph. Data-link edges stand for explicit communication protocols that require software control, such as DMA. Command-link edges stand for instructions that one processor can send to another entity, such as thread spawning or DMA initiation.

The final ingredient in an MM is the morph. A morph is a parallel computer, or a part of one, on which the mapping algorithms are focused. The term morph derives from the polymorphous aspect of the DARPA PCA program for efficient chips that could be reconfigured; a morph would describe one among many configurations of a polymorphous architecture. R-Stream also uses the term “morph” to describe the single configuration for a non-reconfigurable portion of a target.

Each morph in a MM contains a *host processor* and a set of *processing elements* (PEs), plus its entity graph. The host processor and the PEs are allowed to be identical, e.g., in a SMP environment. Associated with the hosts and PEs is its *topology* and a list of submorphs.

A machine model is subdivided into a set of morphs. This mechanism allows a mapping problem for a complex machine to be decomposed into a set of smaller problems, each concentrating on one morph of the machine. In hierarchical mapping, the set of morphs form a tree, with sub-morphs representing submapping problems after the high-level mapping problems have been completed.

## The Generalized Dependence Graph

The GDG is the IR used by the mapper. All phases of the mapper take a set of GDGs as an input and produce a set of modified GDGs as output. Thus, mapping proceeds strictly in this IR.

Initially, the input to the mapper is a single GDG that represents the part of the program that the programmer has indicated should be mapped. Mapping phases annotate and rewrite this graph, essentially choosing a

projection of the GDG into the target machine space (across processors) and time (schedule), and then find a good description of that projection in terms of the target machine execution model. That is, the mapper finds a schedule of execution of computation, memory use, and communication.

As the mapping process proceeds, the GDG may be split into multiple GDGs, for submapping problems to portions of a hierarchical or heterogeneous machine.

The GDG (defined below) uses polyhedra to model the computations, dependences, and memory references of the part of the program being mapped. The polyhedral description provides *compactness* and *precision*, for a broad set of useful loop nest structures, and enables tractible analytical formulations of concepts like “dataflow analysis” and “coarse-grained parallelization.”

A GDG is a multigraph, where the nodes represent statements (called *polyops*), and the edges represent the dependences between the statements. The basic information attached to each polyop node includes:

- Its iteration domain, represented as a polyhedral set
- A list of array references, each represented as an affine function from the iteration space, to the data space of the corresponding array
- A predicate function indicating under what conditions the statement should be executed
- Data-dependent conditionals are supported through the predicates.

Attached to each edge of the GDG is its dependence polyhedron, a polyhedral set that encodes the dependence between the iterations of the edge’s nodes.

The polyhedral sets used in the GDG are unions of intersections of integer lattices with parametric polyhedra, called Z-Domains. Z-Domains are the essence of the mathematical language of polyhedral compilation. The use of unions allows for describing a broad range of “shapes” of programs, the integer lattices provide precision through complex mappings. Using parametric polyhedra also provides a richness of description; it enables parametrically specified families of programs to be optimized at once and it enables describing loop nests and array references that depend on iteration variables from enclosing loop nests. R-Stream includes a ground-up implementation of a library for manipulating Z-Domains that provides performance

and functionality that we could not obtain had we used Loehner’s Polylib [25].

## Mapping Flow

The basic mapping flow in R-Stream is through the following mapper phases:

- Perform *raising* to convert mappable regions into the polyhedral form.
- Perform dependence analysis and/or dependence removal optimizations such as array expansion.
- Perform affine scheduling to extract parallelism.
- Perform task formation (a generalization of tiling) and processor placement. Also perform thread partitioning to split the mapped region among the host processor and the co-processing elements.
- Perform memory promotion to improve locality. If necessary, also perform data layout transformations and insert communications.
- Perform *lowering* to convert the mapped program back into the scalar IR, Sprig.

These phases are mixed and matched and can be recursively called for different target architectures. For example, for SMP targets, it is not necessary to promote memory and insert explicit communications, since such machines have caches. (However, in some cases-explicit copies also help on cache-based machines by compacting the footprint of the data accessed by a task; we call such efforts “virtual scratchpad.”) For a target machine with a host processor and multiple GPUs, R-Stream will first map coarsely across GPUs (inserting thread controls and explicit communications) and then recursively perform mapping for parts of the code that will be executed within the GPUs.

## Raising

The raising phase is responsible for converting loop nests inside mappable regions into the polyhedral form. R-Stream’s raising process is as follows:

1. Mappable regions are identified by the programmer via pragmas. Mappable regions can also be expanded by user-directed inlining pragmas.
2. Perform loop detection and if/then/else region detection.
3. Perform if-conversion [3] to convert data dependent predicates into predicated statements.

4. Perform induction variable detection using the algorithm of Pop [30]. The algorithm has been extended to detect inductions based on address arithmetic. Affine loop bounds and affine address expressions can be converted into the iteration domains and access functions in the GDG.
5. Within each basic block, partition the operators into maximal subsets of operators connected via value flow such that the partitions can be sequentially ordered. These maximal subsets are made into individual polyops.
6. Finally, convert the loop iteration domains and access functions into constraints form in the GDG.

## Dependence Analysis and Array Expansion

R-Stream provides a more advanced covering dependence analysis from Vasilache [39] to remove transitively implied dependencies, which improves upon the algorithms from Feautrier [18].

R-Stream provides a novel *corrective array expansion* based on Vasilache’s violated dependence analysis [37, 38]. Corrective array expansion essentially fuses array expansion phases into scheduling phases.

A “violated” dependence is a relationship between a source and a target instruction of the program that exhibit a memory-based dependence in the original program that is not fulfilled under a chosen schedule. A violation arises when the target statement is scheduled before the source statement in the transformed program.

Such a violation can be corrected by either recomputing a new schedule or by changing the memory locations read and written. This allows for better schedules with as small a memory footprint as possible by performing the following steps:

1. Provide a subset of the original dependencies in the program to the affine scheduler phase so that it determines an aggressive schedule with maximum parallelism.
2. When the schedule turns out to be incorrect with regards to the whole set of dependencies, perform a correction of the resulting schedule by means of renaming and array expansion.
3. Perform lazy corrections targeting only the very causes of the semantics violations in the aggressively scheduled program.

Unlike previous array expansion algorithms in the literature by Feautrier [16], Barthou et al. [5], and Offner et al. [29], which tend to expand all arrays fully and render the program in dynamic single assignment form, the above algorithm only expands arrays by need – i.e., only if the parallelism exposed by the scheduling algorithm requires the array to be expanded.

## Affine Scheduling

Under the exact affine representation of dependences in the GDG, it was known that useful scheduling properties of programs can be optimized such as maximal fine-grained parallelism using Feautrier’s algorithm [20], maximal coarse-grained parallelism using Lim and Lam’s algorithm [24], or maximal parallelism given a (maximal) fusion/distribution structure using Bondhugula’s et al.’s algorithm [8]. R-Stream improved on these algorithms by introducing the concept of affine fusion and this enables a single seamless formulation of the joint optimization of cost functions representing trade-offs between amount of parallelism and amount of locality at various depths in the loop nest hierarchy. Scheduling algorithms in R-Stream search an optimization space that is either constructed on a depth-by-depth basis as solutions are found, or based on the convex space of all legal multi dimensional schedules as had been illustrated by Vasilache [39]. R-Stream allows direct optimization of the tradeoff function using Integer Linear Programming (ILP) solvers as well as iterative exploration of the search space. Redundant solutions in the search space are implicitly pruned out by the combined tradeoff function as they exhibit the same overall cost. In practice, a solution to the optimization problem represents a whole class of equivalent scheduling functions with the same cost.

Building upon the multi dimensional affine fusion formulation, R-Stream also introduced the concept of joint optimization of parallelism, locality, and amount of contiguous memory accesses. This additional metric is targeted at memory hierarchies where accessing a contiguous set of memory references is crucial to obtain high performance. Such hardware features include hardware and software prefetchers, and explicit vector load and store into and from registers, and coalescing hardware in GPUs. R-Stream defines the cost of contiguous memory accesses up to an affine data layout transformation.

To support hierarchical/heterogeneous architectures, the scheduling algorithm can be used hierarchically. Different levels of the hardware hierarchy are optimized using very different cost functions. At the outermost levels of the hierarchy, weight is put on obtaining coarse-grained parallelism with minimal communications that keep as many compute cores busy. At the innermost levels, after communications have been introduced, focus is set to fine-grained parallelism, contiguity and alignment to enable hardware features such as vectorization. Tiling also separates mapping concerns between different levels of the hierarchy.

## Task Formation and Placement

The streaming execution model that R-Stream targets is as follows:

- A bulk set of input data  $D_i$  is loaded into a memory  $M$  that is close to processing element  $P$ .
- A set of operations  $T$ , called “task,” works on  $D_i$ .
- A live-out data set  $D_o$  produced by  $T$  is unloaded from  $M$  to make room for the next task’s input data.

The role of task formation is to partition the program’s operations into such tasks in a way that minimizes the program execution time. Since computing is much faster than transferring data, one of the goals of task formation is to obtain a high computation-per-communication ratio for the tasks. The memory  $M$  may be a local or scratchpad memory, which holds a fixed number of bytes. This imposes a capacity constraint: the working data set of a task should not overflow  $M$ ’s capacity. Finally, the considered processing element  $P$  could be composite, in the sense that it may be itself formed of several processing elements. Task formation also ensures that enough parallelism is provided to such processing elements.

R-Stream’s task formation algorithm improves on prior algorithms for tiling by Ancourt et al. [4], Xue [42], and Ahmed et al. [2]. The algorithm is enabled by using Ehrhart polynomial techniques developed by Clauss et al. [12], Verdoolage et al. [40], and Meister et al. [28]. The Ehrhart polynomials are used to quickly count the volume (number of integer points) in Z-Domains that model the data footprint implied by prospective iteration tilings. The R-Stream task formation algorithm directly tiles imperfect loop nests independently. The prior work requires perfect loop nests or

expanded imperfect loop nests into a perfect form that forced all sections of code, even vastly unrelated ones into the same tiling structure.

The R-Stream task formation algorithm uses *evaluators* and *manipulators*, which can be associated with task groups, loops or the whole function. Evaluators are a function of the elements of the entity they are associated with. They can be turned into a constraint (implicitly: evaluator must be nonnegative) or an objective function. Manipulators enforce constraints on the entity they are associated with. For instance, the tile size of a loop can be made to be a multiple of a certain size, or a power of two, and a set of loops can be set to have the same tile size. Evaluators and manipulators can be combined (affine combination, sequence, conditional sequence) to produce a custom tiling problem.

The task formation algorithm addresses whether the task can be executed in parallel or sequentially; it uses heuristics that walk the set of operations of the GDG and groups the operations based on fusion decisions made by the scheduler, data locality considerations, and the possibility of executing the operations on the same processor.

The task formation algorithm also can detect a class of reductions and mark such loops. It then uses associativity of the scanned operations to relax polyhedral dependence constraints when computing the loop's tiling constraints.

## Memory Promotion

R-Stream performs memory promotion for architectures with fast but limited scratch-pad memories on distributed memory architectures. When data are migrated from one memory to another, the compiler also performs a reorganization of the data layout to improve storage utilization, or locality of reference, or to enable other optimizations. Such data layout reorganization often comes “for free,” especially if it can be overlapped with computation via hardware support, such as DMA. This technique is a generalization of Schreiber et al. [35].

## Communication Generation and DMA Optimizations

When the data set is promoted from a source memory to a target memory, R-Stream generates data transfer from the source array to the target array and back. Those transfers are represented as a element-wise copy loop

nest with additional information that identifies which innermost loops transfer the data required for one task. When there is explicit bulk communication hardware between the source and target memories, R-Stream concatenates element-wise transfer operations into strided transfer (e.g., DMA) commands.

## Target-Specific Optimizations: GPU

While R-Stream can target many different architectures, the GPU target is an illustrative case. GPUs are many-core architectures that have multiple levels of parallelism with outer coarse-grained MIMD parallelism and inner finer-grained SIMD (Single Instruction Multiple Threads) parallelism. Furthermore, they have complex memory hierarchy with memories of varied latencies and varied characteristics. To achieve high performance on such architectures, it is very important to exploit all levels of parallelism and utilize appropriate memories in an efficient manner. There are three important aspects to be addressed by an automatically parallelizing high-level compiler for GPUs, namely,

- To identify and utilize outer coarse-grained and inner fine-grained parallelism
- To make GPU DRAM (“global memory”) accesses to be contiguous and aligned in the parallel loop
- To make proper utilization of the on-chip memories, namely “shared memory” and “registers”

R-Stream exploits the affine scheduling algorithm mentioned earlier, to have, whenever possible, coarse-grained parallelism at outer level and fine-grained parallelism at inner level with contiguously aligned memory accesses. The affine scheduler tries to find a right mix of parallelism, data locality, and data contiguity. As a result, not all global memory array accesses can be made contiguous by the schedule. In such cases, and also in cases where global memory accesses are used multiple times, R-Stream generates data transfers from global DRAM to shared memory (which resides on chip and is shared by all threads in a multiprocessor) as a element-wise copy loop nest such that the global memory accesses are contiguous in the transfer. The global memory accesses are then appropriately replaced by the shared memory accesses. As a result, R-Stream achieves data contiguity in global memory and also improves data reuse across threads.

```
#pragma rstream map
void gaussseidel2D_9points(real_t (*A) [N], int pT, int pN) {
 int i, j, t;
 for (t=0; t<pT; t++) {
 for (i=1; i<pN-1; i++) {
 for (j=1; j<pN-1; j++) {
 A[i][j] = C0*A[i][j] +
 C1*(A[i-1][j-1] + A[i-1][j] + A[i-1][j+1] +
 A[i][j-1] + A[i][j+1] +
 A[i+1][j-1] + A[i+1][j] + A[i+1][j+1]);
 }
 }
 }
}
```

**R-Stream Compiler. Fig. 2** R-Stream's input: Gauss-Seidel nine-point stencil

```
#pragma unroll
for (il = (_maxs_32(_maxs_32(-480 * j + -64 * k + -16 * (int)blockIdx.x + -
 (int)threadIdx.y + 7 >> 3, - (int)threadIdx.y + 7 >> 3), __maxs_32(-64 *
i + -1440 * j + -32 * k + -48 * (int)blockIdx.x + - (int)threadIdx.y + 31 +
7 >> 3, 480 * j + 64 * k + 16 * (int)blockIdx.x + - (int)threadIdx.y +
-2076 + 7 >> 3)); il <= _t3; il++) {
 int _t6; int j1;
 _t6 = (_mins_32(_mins_32(480 * j + 64 * k + -8 * il + 16 * (int)blockIdx.x +
 - (int)threadIdx.x + - (int)threadIdx.y + 66 >> 4, -64 * i + -960 * j + 32
 * k + -32 * (int)blockIdx.x + - (int)threadIdx.x + 2078 >> 4), __mins_32(
 (int)threadIdx.x + 2047 >> 4, _mins_32(480 * j + 64 * k + 8 * il + 16 *
 (int)blockIdx.x + - (int)threadIdx.x + (int)threadIdx.y + 2 >> 4, 480 * j +
 64 * k + 16 * (int)blockIdx.x + - (int)threadIdx.x + 33 >> 4)));
#pragma unroll
for (j1 = (_maxs_32(_maxs_32(480 * j + 64 * k + -8 * il + 16 * (int)blockIdx.x +
 - (int)threadIdx.x + - (int)threadIdx.y + -31 + 15 >> 4, __maxs_32(-64 *
i + -960 * j + 32 * k + -32 * (int)blockIdx.x + - (int)threadIdx.x + -30 +
15 >> 4, 64 * i + 1920 * j + 96 * k + 64 * (int)blockIdx.x + -
(int)threadIdx.x + -2107 + 15 >> 4)), __maxs_32(- (int)threadIdx.x + 15 >>
4, __maxs_32(480 * j + 64 * k + 8 * il + 16 * (int)blockIdx.x + -
(int)threadIdx.x + (int)threadIdx.y + -63 + 15 >> 4, 480 * j + 64 * k + 16
* (int)blockIdx.x + - (int)threadIdx.x + -46 + 15 >> 4))): j1 <= _t6; j1++)
{
 A_1_l[8 * il + (int)threadIdx.y][46 + (-480 * j + -64 * k) + (16 * j1 + (-16 *
 (int)blockIdx.x + (int)threadIdx.x))] = A_1_l[-31 + (64 * i + 1440 * j) + (32
 * k + 8 * il + (48 * (int)blockIdx.x + (int)threadIdx.y))][16 * j1 +
 (int)threadIdx.x];
}
syncthreads();
```

**R-Stream Compiler. Fig. 3** Excerpt of R-Stream's CUDA output: Gauss-Seidel nine-point stencil

## Example

[Figure 2](#) shows input code to R-Stream. The input code is ANSI C, for a simple nine point Gauss Seidel stencil operated on a 2D array. The input style that can be raised is succinct and the user does not give directives describing the mapping, only the indication that the compiler should map the function. The compiler determines the mapping structure automatically. The total output code for this important kernel must be hundreds of lines long to achieve performance in CUDA and on GPUs; a small excerpt of the output from R-Stream in CUDA from this example is in [Fig. 3](#). One can see that the compiler has formed parallelism that is implicit in the CUDA thread and block structure. The chosen schedule balances parallelism, locality, and in particular contiguity, to enable coalescing of loads. Also illustrated is the creation of local copies of the array, the generation of explicit copies into the CUDA scratchpad “shared memories.”

## Related Entries

► [Parallelization, Automatic](#)

## Bibliographic Notes and Further Reading

The final report on R-Stream research for DARPA is Lethin et al. [22]. More information on R-Stream is available from various subsequent workshop papers by Leung et al. [23], Meister et al. [27], and Bastoul et al. [7]. Several US and International patent applications by this encyclopedia entry’s authors are pending and will publish with even more detail on the optimizations in R-Stream.

Feautrier is credited with defining the polyhedral model in the late 1980s and early 1990s [17, 19, 20]. Darte et al. provided a detailed comparison of polyhedral techniques to classical techniques, in terms of the benefits of the exact dependence relations, in 2000 [15]. Much theoretical work accumulated on the polyhedral model in the decades after its invention, but it took the work of Quillere, Rajopadhye, and Wilde to unlock the technique for application by showing effective code generation algorithms [31]; these were later improved by Bastoul [6] and Vasilache [39] (the latter thesis also providing a good bibliography on the polyhedral model).

## Bibliography

1. nVidia CUDA Compute Unified Device Architecture Programming Guide (Version 2.0), June 2008
2. Ahmed N, Mateev N, Pingali K (2000) Tiling imperfectly-nested loop nests. In: Supercomputing ’00: proceedings of the 2000 ACM/IEEE conference on supercomputing (CDROM), IEEE Computer Society, Washington, DC, pp 60–90
3. Allen JR, Kennedy K, Porterfield C, Warren J (1983) Conversion of control dependence to data dependence. In: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on principles of programming languages, New York, pp 177–189
4. Ancourt C, Irigoin F (1991) Scanning polyhedra with DO loops. In: Proceedings of the 3rd ACM SIGPLAN symposium on principles and practice of parallel programming, Williamsburg, VA, pp 39–50, Apr 1991
5. Barthou D, Cohen A, Collard JF (1998) Maximal static expansion. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on principles of programming languages, New York, pp 98–106
6. Bastoul C (2003) Efficient code generation for automatic parallelization and optimization. In: Proceedings of the international symposium on parallel and distributed computing, Ljubljana, pp 23–30, Oct 2003
7. Bastoul C, Vasilache N, Leung A, Meister B, Wohlford D, Lethin R (2009) Extended static control programs as a programming model for accelerators: a case study: targetting Clearspeed CSX700 with the R-Stream compiler. In: First workshop on Programming Models for Emerging Architectures (PMEA)
8. Bondhugula U, Hartono A, Ramanujan J, Sadayappan P (2008) A practical automatic polyhedral parallelizer and locality optimizer. In: ACM SIGPLAN Programming Languages Design and Implementation (PLDI ’08), Tucson, Arizona, June 2008
9. Buck I (2003) Brook v0.2 specification. Technical report, Stanford University, Oct 2003
10. Cifuentes C (1993) A structuring algorithm for decompilation. In: Proceedings of the XIX Conferencia Latinoamericana de Informatica, Buenos Aires, Argentina, pp 267–276
11. Cifuentes C (1994) Structuring Decompiled graphs. Technical Report FIT-TR-1994-05, Department of Computer Science, University of Tasmania, Australia, 19. Also in Proceedings of the 6th international conference on compiler construction, 1996, pp 91–105
12. Clauss P, Loehner V (1996) Parametric analysis of polyhedral iteration spaces. In: IEEE international conference on application specific array processors, ASAP’96. IEEE Computer Society, Los Alamitos, Calif, Aug 1996
13. Click C, Paleczny M (1995) A simple graph-based intermediate representation. ACM SIGPLAN Notices, San Francisco, CA
14. Cytron R, Ferrante J, Rosen BK, Zadeck FK (1991) Efficiently computing static single assignment form and the control dependence graph. ACM Trans Program Lang Syst 13(4):451–490
15. Darte A, Schreiber R, Villard G (2005) Lattice-based memory allocation. IEEE Trans Comput 54(10):1242–1257
16. Feautrier P (1988) Array expansion. In: Proceedings of the 2nd international conference on supercomputing, St. Malo, France

17. Feautrier P (1988) Parametric integer programming. *RAIRO-Recherche Opérationnelle*, 22(3):243–268
18. Feautrier P (1991) Dataflow analysis of array and scalar references. *Int J Parallel Prog* 20(1):23–52
19. Feautrier P (1992) Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *Int J Parallel Prog* 21(5):313–348
20. Feautrier P (1992) Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Int J Parallel Prog* 21(6):389–420
21. StreamIt Group (2003) StreamIt language specification, version 2.0. Technical report, Massachusetts Institute of Technology, Oct 2003
22. Lethin R, Leung A, Meister B, Szilagyi P, Vasilache N, Wohlford D (2008) Final report on the R-Stream 3.0 compiler DARPA/AFRL Contract # F03602-03-C-0033, DTIC AFRL-RI-RS-TR-2008-160. Technical report, Reservoir Labs, Inc., May 2008
23. Leung A, Meister B, Vasilache N, Baskaran M, Wohlford D, Bastoul C, Lethin R (2010) A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In: Third Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU-3, Mar 2010
24. Lim AW, Lam MS (1997) Maximizing parallelism and minimizing synchronization with affine transforms. In: Proceedings of the 24th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, Paris, France, pp 201–214
25. Loechner V (1999) Polylib: a library for manipulating parametrized polyhedra. Technical report, University of Louis Pasteur, Strasbourg, France, Mar 1999
26. Mai K, Paaske T, Jayasena N, Ho R, Dally W, Horowitz M (2000) Smart memories: a modular reconfigurable architecture. In: Proceedings of the international symposium on Computer architecture, pp 161–171, June 2000
27. Meister B, Leung A, Vasilache N, Wohlford D, Bastoul C, Lethin R (2009) Productivity via automatic code generation for PGAS platforms with the R-Stream compiler. In: Workshop on asynchrony in the PGAS programming model, June 2009
28. Meister B, Verdoolege S (2008) Polynomial approximations in the polytope model: bringing the power of quasi-polynomials to the masses. In: ODES-6: 6th workshop on optimizations for DSP and embedded systems, Apr 2008
29. Offner C, Knobe K (2003) Weak dynamic single assignment form. Technical Report HPL-2003-169, HP Labs
30. Pop S, Cohen A, Silber G (2005) Induction variable analysis with delayed abstractions. In: Proceedings of the 2005 international conference on high performance embedded architectures and compilers, Barcelona, Spain
31. Quilleré F, Rajopadhye S, Wilde D (2000) Generation of efficient nested loops from polyhedra. *Int J Parallel Prog* 28(5): 469–498
32. Rettberg RD, Crowther WR, Carvey PP, Tomlinson RS (1990) The Monarch parallel processor hardware design. *Computer* 23:18–30
33. Richards MA (2005) Fundamentals of radar signal processing. McGraw-Hill, New York
34. Sankaralingam K, Nagarajan R, Gratz P, Desikan R, Gulati D, Hanson H, Kim C, Liu H, Ranganathan N, Sethumadhavan S, Sharif S, Shivakumar P, Yoder W, McDonald R, Keckler SW, Burger DC (2006) The distributed microarchitecture of the TRIPS prototype processor. In: 39th international symposium on microarchitecture (MICRO), Los Alamitos, Calif, Dec 2006
35. Schreiber R, Cronquist DC (2004) Near-optimal allocation of local memory arrays. Technical Report HPL-2004-24, Hewlett-Packard Laboratories, Feb 2004
36. Taylor MB, Kim J, Miller J, Wentzlaff D, Ghodrat F, Greenwald B, Hoffmann H, Johnson P, Lee JW, Lee W, Ma A, Saraf A, Seneski M, Shnidman N, Strumpen V, Frank M, Amarasinghe S, Agarwal A (2002) The raw microprocessor: a computational fabric for software circuits and general purpose programs. *Micro*, Mar 2002
37. Vasilache N, Bastoul C, Cohen A, Girbal S (2006) Violated dependence analysis. In: Proceedings of the 20th international conference on supercomputing (ICS'06), Cairns, Queensland, Australia. ACM, New York, NY, USA, pp 335–344
38. Vasilache N, Cohen A, Pouchet LN (2007) Automatic correction of loop transformations. In: 16th international conference on parallel architecture and compilation techniques (PACT'07), IEEE Computer Society Press, Brasov, Romania, pp 292–304, Sept 2007
39. Vasilache NT (2007) Scalable program optimization techniques in the polyhedral model. PhD thesis, Université Paris Sud XI, Orsay, Sept 2007
40. Verdoolege S, Seghir R, Beyls K, Loechner V, Bruynooghe M (2004) Analytical computation of Ehrhart polynomials: enabling more compiler analyses and optimizations. In: Proceedings of the 2004 international conference on compilers, architecture, and synthesis for embedded systems, ACM Press, New York, pp 248–258
41. Weise D, Crew R, Ernst M, Steensgaard B (1994) Value dependence graph: representation without taxation. In: ACM symposium on principles of programming languages, New York, pp 297–310
42. Xue J (1997) On tiling as a loop transformation. *Parallel Process Lett* 7(4):409–424

## Run Time Parallelization

JOEL H. SALTZ<sup>1</sup>, RAJA DAS<sup>2</sup>

<sup>1</sup>Emory University, Atlanta, GA, USA

<sup>2</sup>IBM Corporation, Armonk, NY, USA

### Synonyms

Parallelization

Irregular array accesses arise in many scientific applications including sparse matrix solvers, unstructured mesh partial differential equation (PDE) solvers,

and particle methods. Traditional compilation techniques require that indices to data arrays be symbolically analyzable at compile time. A common characteristic of irregular applications is the use of indirect indexing to represent relationships among array elements. This means that data arrays are indexed through values of other arrays, called *indirection arrays*. Figure 1 depicts a simple example of a loop with indirection arrays. The use of indirection arrays prevents compilers from identifying array data access patterns. Inability to characterize array access patterns symbolically can prevent compilers from generating efficient code for irregular applications.

The inspector/executor strategy involves using compilers to generate code to examine and analyze data references *during program execution*. The results of this execution-time analysis may be used (1) to determine which off-processor data needs to be fetched and where the data will be stored once it is received and (2) to reorder and coordinate execution of loop iterations in problems with irregular loop carried dependencies, as seen in the example in Fig. 2. The initial examination and analysis phase is called the *inspector*, while the phase that uses results of analysis to optimize program execution is called the *executor*.

Irregular applications can be divided into two subclasses: static and adaptive. Static irregular applications are those in which each object in the system interacts with a predetermined fixed set of objects. The indirection arrays, which capture the object interactions, do not change during the course of the computation (e.g., unstructured mesh PDE solver). In adaptive irregular applications, each object interacts with an evolving list of objects (e.g., molecular dynamics codes). This causes

```
for i = 1 to n do
 x(ia(i)) += y(ib(i))
end do
```

**Run Time Parallelization. Fig. 1** Example of a loop involving indirection arrays

```
for i = 1 to n do
 y(i) += y(ia(i)) + y(ib(i)) + y(ic(i))
end do
```

**Run Time Parallelization. Fig. 2** Example of loop-carried dependencies determined by a subscript array

the indirection array, which is the object interaction list, to slowly change over the life of the computation. Figure 3 shows an example of an adaptive code. In this figure, ia, ib, and ic are the indirection arrays. Arrays x and y are the data arrays containing properties associated with the objects of the system.

The inspector/executor strategy can be used to parallelize static irregular applications targeted for distributed parallel machines and can achieve high performance. Additional optimizations are required to achieve high performance when adaptive irregular applications are parallelized using inspector/executor approach. The example adaptive code in Fig. 3 shows that the indirection array ic does not change its value for every iteration of the outer loop. It typically has to be regenerated every few iterations of the outer loop, providing the opportunity for optimization.

Inspector/executor strategies have also been adopted to optimize I/O performance when applications need to carry out large numbers of small non-contiguous I/O requests. The inspector/executor strategy was initially articulated in the late 1980s and early 1990s, as described in Mirchandaney [20], Saltz [25], Koelbel [13], Krothapalli [14], and Walker [32]. This article addresses use of inspector/executor methods to optimize retrieval and management of off-processor data and in parallelization of loops with loop-carried dependencies created by indirection arrays. Use of inspector/executor methods and related compiler frameworks to optimize I/O has also been described.

## Inspector/Executor Methods and Distributed Memory

The inspector/executor strategy works as follows: During program execution, the inspector examines data references made by a processor and determines (1) which off-processor data elements need to be obtained and (2) where the data will be stored once it is received. The executor uses the information from the inspector to gather off-processor data, carry out the actual computations, and then scatter results back to their home processors after the computational phase is completed. A central strategy of inspector/executor optimization is to identify and exploit situations that permit reuse of information obtained by inspectors. As discussed below, it is often possible to relocate preprocessing outside a set of loops or to carry out interprocedural analysis and

```

for t = 1, step //outer step loop
for i = 1, n //inner loop
 x(ia(i)) = x(ia(i)) + y(ib(i))
endfor //end inner loop
if (required) then
 regenerate ic(:)
endif
for i = 1, n //inner loop
 x(ic(i)) = x(ic(i)) + y(ic(i))
endfor //end inner loop
endfor //end outer step loop

```

**Run Time Parallelization.** Fig. 3 Example loop from adaptive irregular applications

|                                                  |                                                                          |                                                                                 |
|--------------------------------------------------|--------------------------------------------------------------------------|---------------------------------------------------------------------------------|
| for i = 1, n do<br>x(ia(ib(i))) = ....<br>end do | for i = 1, n do<br>if (ic(i)) then<br>x(ia(i)) = ...<br>end if<br>end do | for I =1, n do<br>do j = ia(i), ia(i + 1)<br>x(ia(j)) = ...<br>end do<br>end do |
|--------------------------------------------------|--------------------------------------------------------------------------|---------------------------------------------------------------------------------|

**Run Time Parallelization.** Fig. 4 Examples of loops with complex access functions

transformation so that results from an inspector can be reused many times.

A variety of program analysis methods and transformations have been developed to generate programs that can make use of inspector/executor strategies. In loops with simple irregular access patterns such as those in Fig. 1, a single inspector/executor pair can be generated by the straightforward method described in [31]. Many application codes contain access patterns with more complex access functions such as those depicted in Fig. 4. Subscripted subscripts and subscripted guards can make indexing of one distributed array dependent on values in another. To handle this kind of situation, an inspector must itself be split into an inspector-executor pair as described in [10]. A dataflow framework was developed to help place executor communication calls, determine when it is safe to combine communications statements, move them into less frequently executed code regions, or avoid them altogether in favor of reusing data that is already buffered locally.

## Communication Schedules

A communication schedule is used to fetch off-processor elements into a local buffer before the

computation phase and to scatter computed data back to home processors. Communication schedules determine the volume of communication and number of communication startups. A variety of approaches to schedule generation have been taken. In the CHAOS system [12], a hash table data structure is used to support a two-phase schedule generation process. The index analysis phase examines data access patterns to determine which references are off-processor, remove duplicate off-processor references, assign local buffers for off-processor references, and translate global indices to local indices. The results of index analyses are managed in a hash table data structure. The schedule generation phase then produces communication schedules based on hash table information. The CHAOS hash table contains a bit array used to identify which indirection arrays entered each element into the hash table. This approach facilitates efficient management and coordination of analysis information derived from programs with multiple indirection arrays and provides support for building incremental and merged communication schedules.

In some cases it can be advantageous to employ communication schedules that carry out bounding box

or entire-array bulk read or write operations rather than using schedules that only communicate accessed array elements. These approaches differ in costs associated with both the inspector and the executor phases. The bounding box approach requires only that the inspector compute a bounding box containing all the necessary elements and no inspector is required to retrieve the entire array. A compiler and runtime system using performance models to choose between element-wise gather, bounding box bulk read, and entire-array retrieval were implemented by the Titanium group discussed in [28].

A variety of techniques have been incorporated into CHAOS and into efficiently supported applications such as particle codes, which manifest access patterns that change from iteration to iteration. The crucial observation is that in many such codes, the communication-intensive loops are actually carrying out a generalized reduction in which it is not necessary to control the order of elements stored. Hwang [12] demonstrates that order-independence can be used to dramatically reduce schedule generation overhead.

## Runtime Parallelization

A variety of runtime parallelization techniques have been developed for indirectly indexed loops with data dependencies among iterations, as shown in Figs. 2 and 5. Most of this work has targeted shared memory architectures due to the fine grained concurrency that typically results when such codes are parallelized. One of the earliest techniques for runtime parallelization of loops involves the use of a key field associated with each indirectly indexed array element to order accesses. The algorithm repeatedly sweeps over all loop iterations in alternating analysis and computation phases. An iteration is allowed to proceed only if all accesses to array elements  $y(ia(i))$  and  $y(ib(i))$  by iterations  $j < i$  have completed. These sweeps have the effect of partitioning the computation into wavefronts. Key fields are used to ensure that dependences are taken into account. This is discussed further in [19]. This strategy is extended by allowing concurrent reads to the same entry, as Midkiff explains, and through development of a systematic dependence analysis and program transformation framework for singly nested loops. Intra-procedural and interprocedural analysis techniques are presented in an article by Lin [15] to analyze the common and

```
for i = 1 to n do
 y(ia(i)) = ...
 ...
 ... = y(ib(i))
end do
```

**Run Time Parallelization. Fig. 5** Subscript

array-determined loop carried dependences with output dependence

important cases of irregular single-indexed accesses and simple indirect array accesses.

An inspector/executor approach, applicable to loops without output dependencies, can be carried out through generation of an inspector that identifies *wavefronts* of concurrently executable loop iterations followed by an executor that transforms the original loop  $L$  into two loops,  $L_1$  and  $L_2$ . The new outer loop  $L_1$  is sequential and the new inner loop  $L_2$  involves all loop indexes assigned to each wavefront, as described in Saltz [25]. This is analogous to *loop skewing*, as described by Wolfe [33], except that the identification of parallelism is carried out during program execution.

A doacross loop construct can also be used as an executor. The wavefronts generated by the inspector are used sort indices assigned to each processor into ascending wavefront order. Full/empty bit or busy-waiting synchronization enforces true dependencies; computation does not proceed until the array element required for the computation is available. A doacross executor construct was introduced in [25], and this concept has been greatly refined through a variety of schemes that support operation level synchronization in the work of Chen and Singh [6, 26]. A very fine-grained data flow inspector/executor scheme that also employed operation level synchronization was proposed and tested on the CM-5, as described in Chong's work [7].

Many situations occur in which runtime information might demonstrate that: (1) all loop iterations are independent and could execute in parallel or (2) that cross-iteration dependences are reductions. A kind of inspector/executor technique has been developed that involves (1) carrying out compiler transformations that assume parallelism or reduction parallelism, (2) running the generated speculative loop, and then (3)

invoking tests to assess correctness of the speculative parallel execution, as discussed in Rauchwerger [23]. The compilation and runtime support framework generate tests and recovery code that need to be invoked if the tests demonstrate that the speculation was unsuccessful.

## Structural Abstraction

Over the course of a decade Baden explored three successive runtime systems to treat data-dependent communication patterns and workload distributions: LPAR, LPAR-X, and KeLP [1, 2]. The libraries were used in various applications including: structured adaptive mesh refinement for first principles simulation of real materials, genetic algorithms, cluster identification for spin models in statistical mechanics, turbulent flow, and cell microphysiology. This effort contributed the notion of structural abstraction which supports user-level geometric meta-data that admit data-dependent decompositions used in irregular applications. These also enable communication to be expressed in a high level geometric form which may be manipulated at runtime to optimize communication. KeLP was later extended to handle more general notions of sparse sets of data that logically communicate within rectangular subspaces of  $Z^d$ , even if the data does not exist as rectangular sets of data, e.g., particles, as noted by Baden. A multi-tier variant of KeLP, KeLP2, subsequently appeared to treat hierarchical parallelism in the then-emerging SMP cluster technologies, with support to mask communication delays [1].

## Loop Transformation

The compiler parallelizes the loop shown in Fig. 6 by generating the corresponding inspector/executor pair. Assume that all arrays are aligned and distributed in blocks among the processors, and the iterations of the  $i$ -loop are likewise block-partitioned. The resulting computation mapping is equivalent to that produced by the “owner computes” rule, a heuristic that maps the computation of an assignment statement to the processor that owns the left-hand-side reference. Data array  $y$  is indexed using array  $ia$ , causing a single level of indirection.

The compiler generates a single executable: a copy of this executable runs on each processor. The program running on each processor determines what processor

```
for i = 1 to n do
 x(i) = y(ia(i)) + z(i)
end do
```

**Run Time Parallelization.** Fig. 6 Simple irregular loop

it is running on, and uses that information to figure out where its data and iteration fit in the global computation. Let  $my\_elements$  represent the number of iterations assigned to a processor and also the number of elements of data arrays  $x$ ,  $y$ ,  $z$  and indirection array  $ia$  assigned to the processor. The code generated by the compiler and executed by each processor is shown in Fig. 7.

## Compiler Implementations

Inspector/executor schemes have been implemented in a variety of compilers. Initial inspector/executor implementations in the early 1990s were carried out in the ARF [24], [31] and Kali [13] compilers as discussed by Saltz and Koelbel, respectively. Inspector/executor schemes were subsequently implemented in Fortran D; Fortran 90D [21, 30]; the Polaris compiler discussed by Lin [15]; the SUPERB [5] compiler; the Vienna Fortran compiler [29]; the Titanium Java compiler [28]; compilation systems designed to support efficient implementation of OpenMP on platforms that support MPI, discussed by Basumallik [3]; to platforms that support Global Arrays, discussed by Liu [16]; and in HPF compilers described in examples by Benkner and Merlin [4, 18]. Many of these compiler frameworks support user-specified data and iteration partitioning methods. A framework that supports compiler based analysis about runtime data and iteration reordering transformations is described in [27].

Speculative parallelization techniques have been proposed to effectively parallelize loops with complex data access patterns. Various software [8, 11, 23] and hardware [9, 22] techniques have been proposed. The hardware techniques involve increased hardware complexity but can catch cross-iteration data dependencies immediately with minimal overheads. On the other hand, full software solutions do not require new hardware but in certain cases can lead to significant performance penalties. The first known work on speculative parallelization techniques to parallelize loops

```

//inspector code
for i = 1, my_elements
 index_y(i) = ia(i)
endfor
sched_ia = generate_schedule(index_y)//call the inspector schedule
 generation codes
call gather(y(begin_buffer), y, sched_ia)
//executor code
for i = 1, my_elements
 x(i)= y(index_y(i)) + z(i)
endfor

```

**Run Time Parallelization.** Fig. 7 Transformed irregular loop

with complex data access pattern was proposed by Rauchwerger et al. [23]. Instead of breaking loops into inspector/executor computations, they do a speculative execution of the loop as a doall with a runtime check (the LRPD test) to assess whether there are any cross-iteration dependencies. The scheme supports back-tracking and serial re-execution of the loop if the runtime check fails. The biggest advantage of this technique is that the access pattern of the data arrays does not have to be analyzed separately and the runtime tests are performed during the actual computation of the loop. They used this technique on loops, which cannot be parallelized statically, from the PERFECT Benchmark and showed that in many cases, it leads to code that performs better than the inspector/executor strategy. The scheme can be used for automatic parallelization of complex loops, but its main limitation is that dependency violations are detected after the computation of the loop, and a severe penalty has to be paid in terms of rolling back the computation and executing it serially. Gupta et al. [11] proposed a number of new runtime tests that improve the techniques presented in [23]; thus significantly reducing the penalties associated with misspeculation. Dang et al. [9] proposed a new technique that transforms a partially parallel loop into a sequence of fully parallel loops. It is a recursive technique where in each step all remaining iterations of the irregular loop are executed in parallel. After the execution, the LRPD test is done to detect dependencies. The iterations that were correct are committed, and the recursive process starts with the remaining iterations. The limitation of the technique is that the loop iterations have to be statically block-scheduled in increasing order

of iterations between the processors, possibly causing work imbalance.

## Bibliography

- Baden SB, Fink SJ (2000) A programming methodology for dual-tier multicompilers. *IEEE Trans Softw Eng* 26(3):212–226
- Baden SB, Kohn SR (1995) Portable parallel programming of numerical problems under the LPAR system. *J Parallel Distrib Comput* 27(1):38–55
- Basumallik A, Eigenmann R (2006) Optimizing irregular shared-memory applications for distributed-memory systems. In: Proceedings of the eleventh ACM SIGPLAN symposium on principles and practice of parallel programming (PPoPP'06), New York, 29–31 Mar 2006. ACM, New York, pp 119–128
- Benkner S, Mehrotra P, Van Rosendale J, Zima H (1998) High-level management of communication schedules in HPF-like languages. In: Proceedings of the 12th international conference on supercomputing (ICS'98), Melbourne. ACM, New York, pp 109–116
- Brezany P, Gerndt M, Sipkova V, Zima HP (1992) SUPERB support for irregular scientific computations. In: Proceedings of the scalable high performance computing conference (SHPCC-92), Williamsburg, 26–29 Apr 1992, pp 314–321
- Chen DK, Torrellas J, Yew PC (1994) An efficient algorithm for the run-time parallelization of DOACROSS loops. In: Proceedings of the 1994 ACM/IEEE conference on supercomputing, Washington, DC, pp 518–527
- Chong FT, Sharma SD, Brewer EA, Saltz JH (1995) Multiprocessor runtime support for fine-grained, irregular DAGs. *Parallel Process Lett* 5:671–683
- Cintra M, Martinez J, Torrellas J (2000) Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In: Proceedings of 27th annual international symposium on computer architecture, Vancouver, pp 13–24
- Dang F, Yu H, Rauchwerger L (2002) The R-LRPD test: speculative parallelization of partially parallel loops. In: Proceedings of the international parallel and distributed processing symposium (IPDPS02), Ft. Lauderdale

10. Das R, Saltz J, von Hanxleden R (1994) Slicing analysis and indirect accesses to distributed arrays. Lecture notes in computer science, vol 768. Springer, Berlin/Heidelberg
11. Gupta M, Nim R (1998) Techniques for speculative run-time parallelization of loops. In: Proceedings of the 1998 ACM/IEEE conference on supercomputing (SC'98), pp 1–12
12. Hwang YS, Moon B, Sharma SD, Ponnusamy R, Das R, Saltz JH (1995) Runtime and language support for compiling adaptive irregular programs. *Softw Pract Exp* 25(6):597–621
13. Koelbel C, Mehrotra P, Van Rosendale J (1990) Supporting shared data structures on distributed memory machines. In: Symposium on principles and practice of parallel programming. ACM, New York, pp 177–186
14. Krothapalli VP, Sadayappan P (1990) Dynamic scheduling of DOACROSS loops for multiprocessors. In: International conference on databases, parallel architectures and their applications (PARBASE-90), Miami, pp 66–75
15. Lin Y, Padua DA (2000) Compiler analysis of irregular memory accesses. In: Proceedings of the ACM SIGPLAN 2000 conference on programming language design and implementation. Vancouver, pp 157–168
16. Liu Z, Huang L, Chapman BM, Weng TH (2004) Efficient implementation of OpenMP for clusters with implicit data distribution. WOMPAT, Houston, pp 121–136
17. Lusk EL, Overbeek RA (1987) A minimalist approach to portable, parallel programming. In: Jamieson L, Gannon D, Douglass R (eds) The characteristics of parallel algorithms. MIT Press, Cambridge, MA, pp 351–362
18. Merlin JH, Baden SB, Fink S, Chapman BM (1999) Multiple data parallelism with HPF and KeLP. *Future Gener Comput Syst* 15(3):393–405
19. Midkiff SP, Padua DA (1987) Compiler algorithms for synchronization. *IEEE Trans Comput* 36(12):1485–1495
20. Mirchandaney R, Saltz JH, Smith RM, Nicol DM, Crowley K (1988) Principles of runtime support for parallel processors. In: Proceedings of the second international conference on supercomputing (ICS'88), St. Malo, pp 140–152
21. Ponnusamy R, Hwang YS, Das R, Saltz JH, Choudhary A, Fox G (1995) Supporting irregular distributions using data-parallel languages. *Parallel Distrib Technol Syst Appl* 3(1):12–24
22. Prvulovic M, Garzaran MJ, Rauchwerger L, Torrellas J (2001) Removing architectural bottlenecks to the scalability of speculative parallelization. In: Proceedings, 28th annual international symposium on Computer architecture, pp 204–215
23. Rauchwerger L, Padua D (1995) The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In: Proceedings of the ACM SIGPLAN 1995 conference on programming language design and implementation (PLDI'95), La Jolla, 18–21 June 1995. ACM, New York, pp 218–232
24. Saltz JH, Berryman H, Wu J (1991) Multiprocessors and run-time compilation. *Concurr Pract Exp* 3(6):573–592
25. Saltz JH, Mirchandaney R, Crowley K (1991) Run-time parallelization and scheduling of loops. *IEEE Trans Comput* 40(5):603–612
26. Singh DE, Martín MJ, Rivera FF (2003) Increasing the parallelism of irregular loops with dependences. In: Euro-Par, Klagenfurt, pp 287–296
27. Strout M, Carter L, Ferrante J (2003) Compile-time composition of run-time data and iteration reordering. *Program Lang Des Implement* 38(5):91–102
28. Su J, Yelick K (2005) Automatic support for irregular computations in a high-level language. In: Proceedings, 19th IEEE International Parallel and distributed processing symposium, Atlanta, pp 53b, 04–08 Apr 2005
29. Ujaldon M, Zapata EL, Chapman BM, Zima HP (1997) Vienna-Fortran/HPF extensions for sparse and irregular problems and their compilation. *IEEE Trans Parallel Distrib Syst* 8(10):1068–1083
30. von Hanxleden R, Kennedy K, Koelbel C, Das R, Saltz J (1993) Compiler analysis for irregular problems in Fortran D. In: Proceedings of the fifth international workshop on languages and compilers for parallel computing. Springer, London, pp 97–111
31. Wu J, Das R, Saltz J, Berryman H, Hiranandani S (1995) Distributed memory compiler design for sparse problems. *IEEE Trans Comput* 44(6):737–753
32. Walker D (1989) The implementation of a three-dimensional PIC Code on a hypercube concurrent processor. In: Conference on hypercubes, concurrent computers, and application, Pasadena
33. Wolfe M (1989) More iteration space tiling. In: Proceedings of the 1989 ACM/IEEE conference on supercomputing (Supercomputing'89), pp 655–664

## Runtime System

### ► Operating System Strategies

# S

## Scalability

- ▶ Metrics
- ▶ Single System Image

## Scalable Coherent Interface (SCI)

- ▶ SCI (Scalable Coherent Interface)

## ScalAPACK

JACK DONGARRA, PIOTR LUSZCZEK  
University of Tennessee, Knoxville, TN, USA

### Definition

ScalAPACK is a library of high-performance linear algebra routines for distributed-memory message-passing MIMD computers and networks of workstations supporting PVM [1] and/or MPI [2, 3]. It is a continuation of the LAPACK [4] project, which designed and produced analogous software for workstations, vector supercomputers, and shared-memory parallel computers.

### Discussion

Both LAPACK and ScalAPACK libraries contain routines for solving systems of linear equations, least squares problems, and eigenvalue problems. The goals of both projects are efficiency (to run as fast as possible), scalability (as the problem size and number of processors grow), reliability (including error bounds), portability (across all important parallel machines), flexibility (so users can construct new routines from well-designed parts), and ease of use (by making the interface to LAPACK and ScalAPACK look as similar as possible). Many of these goals, particularly

portability, are aided by developing and promoting standards, especially for low-level communication and computation routines. These goals have been successfully attained, limiting most machine dependencies to two standard libraries called the BLAS, or Basic Linear Algebra Subprograms [5–8], and BLACS, or Basic Linear Algebra Communication Subprograms [9, 10]. LAPACK will run on any machine where the BLAS are available, and ScalAPACK will run on any machine where both the BLAS and the BLACS are available.

The library is written in Fortran 77 (with the exception of a few symmetric eigenproblem auxiliary routines written in C to exploit IEEE arithmetic) in a Single Program Multiple Data (SPMD) style using explicit message passing for interprocessor communication. The name ScalAPACK is an acronym for Scalable Linear Algebra PACKage, or Scalable LAPACK.

ScalAPACK can solve systems of linear equations, linear least squares problems, eigenvalue problems, and singular value problems. ScalAPACK can also handle many associated computations such as matrix factorizations or estimating condition numbers.

Like LAPACK, the ScalAPACK routines are based on block-partitioned algorithms in order to minimize the frequency of data movement between different levels of the memory hierarchy. The fundamental building blocks of the ScalAPACK library are distributed-memory versions of the Level 1, Level 2, and Level 3 BLAS, called the Parallel BLAS or PBLAS [11, 12], and a set of Basic Linear Algebra Communication Subprograms (BLACS) [9, 10] for communication tasks that arise frequently in parallel linear algebra computations. In the ScalAPACK routines, the majority of interprocessor communication occurs within the PBLAS. So the source code of the top software layer of ScalAPACK looks similar to that of LAPACK.

ScalAPACK contains *driver routines* for solving standard types of problems, *computational routines* to perform a distinct computational task, and *auxiliary routines* to perform a certain subtask or common

low-level computation. Each driver routine typically calls a sequence of computational routines. Taken as a whole, the computational routines can perform a wider range of tasks than are covered by the driver routines. Many of the auxiliary routines may be of use to numerical analysts or software developers, so the Fortran source for these routines have been documented with the same level of detail used for the ScaLAPACK computational routines and driver routines.

Dense and band matrices are provided for, but not general sparse matrices. Similar functionality is provided for real and complex matrices. However, not all the facilities of LAPACK are covered by ScaLAPACK yet.

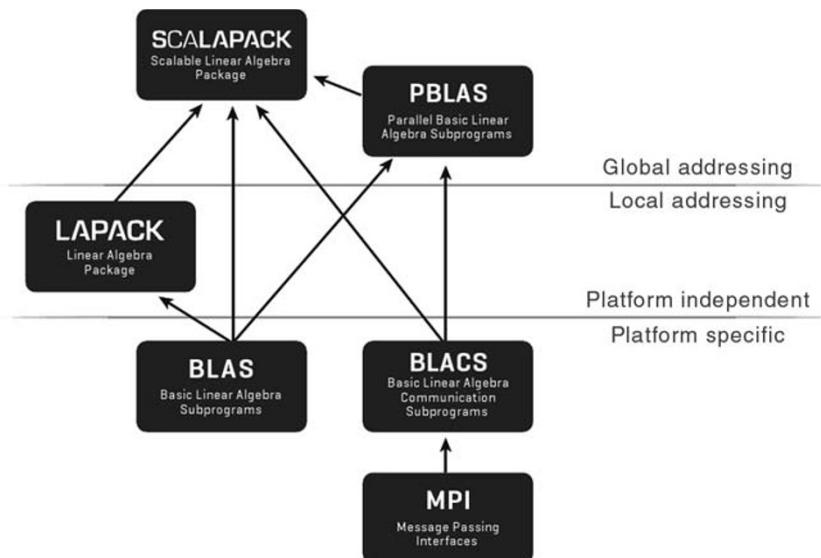
ScaLAPACK is designed to give high efficiency on MIMD distributed-memory concurrent supercomputers, such as the older ones like Intel Paragon, IBM SP series, and the Cray T3 series, as well as the newer ones, such IBM Blue Gene series and Cray XT series of supercomputers. In addition, the software is designed so that it can be used with clusters of workstations through a networked environment and with a heterogeneous computing environment via PVM or MPI. Indeed, ScaLAPACK can run on any machine that supports either PVM or MPI.

The ScaLAPACK strategy for combining efficiency with portability is to construct the software so that as much as possible of the computation is performed by

calls to the Parallel Basic Linear Algebra Subprograms (PBLAS). The PBLAS [11, 12] perform global computation by relying on the Basic Linear Algebra Subprograms (BLAS) [5–8] for local computation and the Basic Linear Algebra Communication Subprograms (BLACS) [9, 10] for communication.

The efficiency of ScaLAPACK software depends on the use of block-partitioned algorithms and on efficient implementations of the BLAS and the BLACS being provided by computer vendors (and others) for their machines. Thus, the BLAS and the BLACS form a low-level interface between ScaLAPACK software and different machine architectures. Above this level, all of the ScaLAPACK software is portable.

The BLAS, PBLAS, and the BLACS are not, strictly speaking, part of ScaLAPACK. C code for the PBLAS is included in the ScaLAPACK distribution. Since the performance of the package depends upon the BLAS and the BLACS being implemented efficiently, they have not been included with the ScaLAPACK distribution. A machine-specific implementation of the BLAS and the BLACS should be used. If a machine-optimized version of the BLAS is not available, a Fortran 77 reference implementation of the BLAS is available from Netlib [13]. This code constitutes the “model implementation” [14, 15]. The model implementation of the BLAS is not expected to perform as well as a specially tuned implementation on most high-performance computers – on



**ScaLAPACK. Fig. 1** ScaLAPACK’s software hierarchy

some machines it may give much worse performance – but it allows users to run ScaLAPACK codes on machines that do not offer any other implementation of the BLAS.

If a vendor-optimized version of the BLACS is not available for a specific architecture, efficiently ported versions of the BLACS are available on Netlib. Currently, the BLACS have been efficiently ported on machine-specific message-passing libraries such as the IBM (MPL) and Intel (NX) message-passing libraries, as well as more generic interfaces such as PVM and MPI. The BLACS overhead has been shown to be negligible [10]. Refer to the URL for the blacs directory on Netlib for more details: <http://www.netlib.org/blacs/index.html>

Figure 1 describes the ScaLAPACK software hierarchy. The components below the line, labeled *local addressing*, are called on a single processor, with arguments stored on single processors only. The components above the line, labeled *global addressing*, are synchronous parallel routines, whose arguments include matrices and vectors distributed across multiple processors.

LAPACK and ScaLAPACK are freely available software packages provided on the World Wide Web on Netlib. They can be, and have been, included in commercial packages. The authors ask only that proper credit be given to them which is very much like the modified BSD license.

## Related Entries

- [LAPACK](#)
- [Linear Algebra, Numerical](#)

## Bibliography

1. Geist A, Beguelin A, Dongarra J, Jiang W, Manchek R, Sunderam V (1994) Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge, MA, 1994
2. MPI Forum, MPI: A message passing interface standard, International Journal of Supercomputer Applications and High Performance Computing, 8 (1994), pp 3–4. Special issue on MPI. Also available electronically, the URL is [ftp://www.netlib.org/mpi/mpi-report.ps](http://www.netlib.org/mpi/mpi-report.ps)
3. Snir M, Otto SW, Huss-Lederman S, Walker DW, Dongarra JJ (1996) MPI: The Complete Reference, MIT Press, Cambridge, MA

4. Anderson E, Bai Z, Bischof C, Blackford LS, Demmel JW, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D, LAPACK Users' Guide, SIAM, 1992
5. Hanson R, Krogh F, Lawson CA (1973) A proposal for standard linear algebra subprograms, ACM SIGNUM Newsl, 8
6. Lawson CL, Hanson RJ, Kincaid D, Krogh FT (1979) Basic Linear Algebra Subprograms for Fortran Usage, ACM Trans Math Soft 5:308–323
7. Dongarra JJ, Du Croz J, Hammarling Richard S, Hanson J (March 1988) An Extended Set of FORTRAN Basic Linear Algebra Subroutines, ACM Trans Math Soft 14(1):1–17
8. Dongarra JJ, Du Croz J, Duff IS, Hammarling S (March 1990) A Set of Level 3 Basic Linear Algebra Subprograms, ACM Trans Math Soft 16(1):1–17
9. Dongarra J, van de Geijn R (1991) Two dimensional basic linear algebra communication subprograms, Computer Science Dept. Technical Report CS-91-138, University of Tennessee, Knoxville, TN (Also LAPACK Working Note #37)
10. Dongarra J, Whaley RC (1995) A user's guide to the BLACS v1.1, Computer Science Dept. Technical Report CS-95-281, University of Tennessee, Knoxville, TN (Also LAPACK Working Note #94)
11. Choi J, Dongarra J, Ostrouchov S, Petitet A, Walker D, Whaley RC (May 1995) A proposal for a set of parallel basic linear algebra subprograms, Computer Science Dept. Technical Report CS-95-292, University of Tennessee, Knoxville, TN (Also LAPACK Working Note #100)
12. Petitet A (1996) Algorithmic Redistribution Methods for Block Cyclic Decompositions, PhD thesis, University of Tennessee, Knoxville, TN
13. Dongarra JJ, Grosse E (July 1987) Distribution of Mathematical Software via Electronic Mail, Communications of the ACM 30(5): 403–407
14. Dongarra JJ, du Croz J, Duff IS, Hammarling S (1990) Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms, ACM Trans Math Soft 16:18–28
15. Dongarra JJ, DU Croz J, Hammarling S, Hanson RJ (1998) Algorithm 656: An extended set of FORTRAN Basic Linear Algebra Subroutines, ACM Trans Math Soft 14:18–32

## Scalasca

FELIX WOLF  
Aachen University, Aachen, Germany

## Synonyms

The predecessor of Scalasca, from which Scalasca evolved, is known by the name of KOJAK.

## Definition

Scalasca is an open-source software tool that supports the performance optimization of parallel programs by

measuring and analyzing their runtime behavior. The analysis identifies potential performance bottlenecks – in particular those concerning communication and synchronization – and offers guidance in exploring their causes. Scalasca targets mainly scientific and engineering applications based on the programming interfaces MPI and OpenMP, including hybrid applications based on a combination of the two. The tool has been specifically designed for use on large-scale systems including IBM Blue Gene and Cray XT, but is also well suited for small- and medium-scale HPC platforms.

To address these challenges, Scalasca has been designed as a diagnostic tool to support application optimization on highly scalable systems. Although also covering single-node performance via hardware-counter measurements, Scalasca mainly targets communication and synchronization issues, whose understanding is critical for scaling applications to performance levels in the petaflops range. A distinctive feature of Scalasca is its ability to identify wait states that occur, for example, as a result of unevenly distributed workloads.

## Discussion

### Introduction

Driven by growing application requirements and accelerated by current trends in microprocessor design, the number of processor cores on modern supercomputers is expanding from generation to generation. As a consequence, supercomputing applications are required to harness much higher degrees of parallelism in order to satisfy their enormous demand for computing power. However, with today's leadership systems featuring more than a hundred thousand cores, writing efficient codes that exploit all the available parallelism becomes increasingly difficult. Performance optimization is therefore expected to become an even more essential software-process activity, critical for the success of many simulation projects. The situation is exacerbated by the fact that the growing number of cores imposes scalability demands not only on applications but also on the software tools needed for their development.

Making applications run efficiently on larger scales is often thwarted by excessive communication and synchronization overheads. Especially during simulations of irregular and dynamic domains, these overheads are often enlarged by wait states that appear in the wake of load or communication imbalance when processes fail to reach synchronization points simultaneously. Even small delays of single processes may spread wait states across the entire machine, and their accumulated duration can constitute a substantial fraction of the overall resource consumption. In particular, when trying to scale communication-intensive applications to large processor counts, such wait states can result in substantial performance degradation.

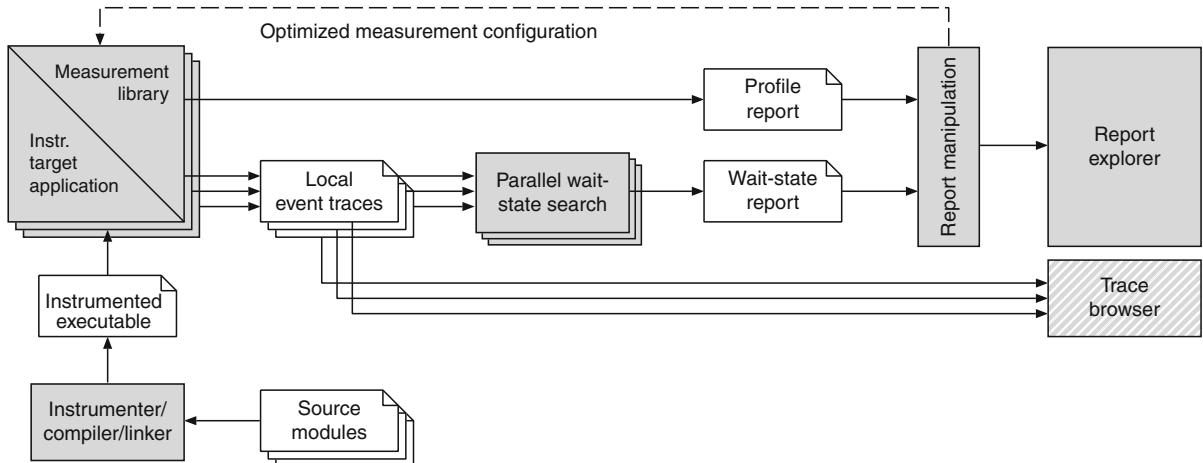
### Functionality

To evaluate the behavior of parallel programs, Scalasca takes performance measurements at runtime to be analyzed *postmortem* (i.e., after program termination). The user of Scalasca can choose between two different analysis modes:

- Performance overview on the call-path level via profiling (called runtime summarization in Scalasca terminology)
- In-depth study of application behavior via event tracing

In profiling mode, Scalasca generates aggregate performance metrics for individual function call paths, which are useful for identifying the most resource-intensive parts of the program and assessing process-local performance via hardware-counter analysis. In tracing mode, Scalasca goes one step further and records individual performance-relevant events, allowing the automatic identification of call paths that exhibit wait states. This core feature is the reason why Scalasca is classified as an automatic tool. As an alternative, the resulting traces can be visualized in a traditional time-line browser such as ▶VAMPIR to study the detailed interactions among different processes or threads. While providing more behavioral detail, traces also consume significantly more storage space and therefore have to be generated with care.

Figure 1 shows the basic analysis workflow supported by Scalasca. Before any performance data can be collected, the target application must be instrumented, that is, probes must be inserted into the code which carry out the measurements. This can happen at different levels, including source code, object code, or library.



**Scalasca.** Fig. 1 Schematic overview of the performance data flow in Scalasca. Gray rectangles denote programs and white rectangles with the upper right corner turned down denote files. Stacked symbols denote multiple instances of programs or files, in most cases running or being processed in parallel. Hatched boxes represent optional third-party components

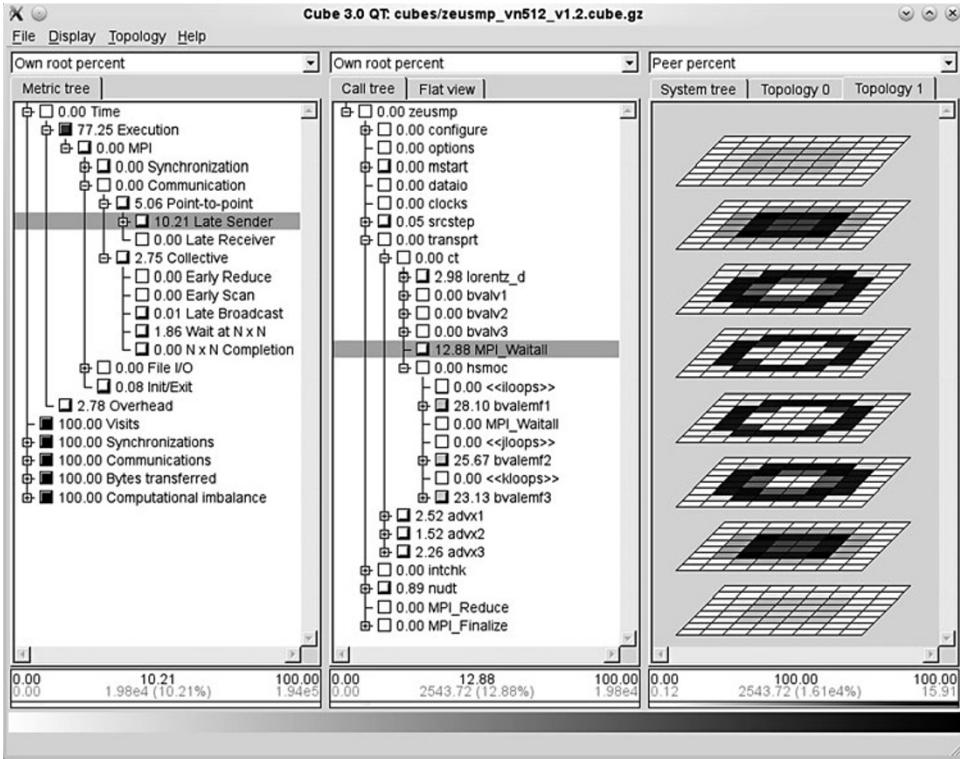
Before running the instrumented executable on the parallel machine, the user can choose between generating a profile or an event trace. When tracing is enabled, each process generates a trace file containing records for its process-local events. To prevent traces from becoming too large or inaccurate as a result of measurement intrusion, it is generally recommended to optimize the instrumentation based on a previously generated profile report. After program termination, Scalasca loads the trace files into main memory and analyzes them in parallel using as many cores as have been used for the target application itself. During the analysis, Scalasca searches for wait states, classifies detected instances by category, and quantifies their significance. The result is a wait-state report similar in structure to the profile report but enriched with higher-level communication and synchronization inefficiency metrics. Both profile and wait-state reports contain performance metrics for every combination of function call path and process/thread and can be interactively examined in the provided analysis report explorer (Fig. 2) along the dimensions of performance metric, call tree, and system. In addition, reports can be combined or manipulated to allow comparisons or aggregations, or to focus the analysis on specific extracts of a report. For example, the difference between two reports can be calculated to assess the effectiveness of an optimization or a new report can be generated after eliminating uninteresting phases (e.g., initialization).

## Instrumentation

Preparation of a target application executable for measurement and analysis requires it to be *instrumented* to notify the measurement library, which is linked to the executable, of performance-relevant execution events whenever they occur at runtime. On all systems, a mix of manual and automatic instrumentation mechanisms is offered. Instrumentation configuration and processing of source files are achieved by prefixing selected compilation commands and the final link command with the Scalasca instrumenter, without requiring other changes to optimization levels or the build process, as in the following example for the file `foo.c`:

```
> scalasca -instrument mpicc -c foo.c
```

Scalasca follows a *direct instrumentation* approach. In contrast to interrupt-based sampling, which takes periodic measurements whenever a timer expires, Scalasca takes measurements when the control flow reaches certain points in the code. These points mark performance-relevant events, such as entering/leaving a function or sending/receiving a message. Although instrumentation points have to be chosen with care to minimize intrusion, direct instrumentation offers advantages for the global analysis of communication and synchronization operations. In addition to pure direct instrumentation, future versions of Scalasca will combine direct instrumentation with sampling in



**Scalasca.** Fig. 2 Interactive performance-report exploration for the Zeus MP/2 code [8] with Scalasca. The *left pane* lists different performance metrics arranged in a specialization hierarchy. The selected metric is a frequently occurring wait state called *Late Sender*, during which a process waits for a message that has not yet been sent (Fig. 3a). The number and the color of the icon to the *left* of the label indicate the percentage of execution time lost due to this wait state, in this case 10.21%. In the *middle pane*, the user can see which call paths are most affected. For example, 12.88% of the 10.21% is caused by the call path `zeusmp () → transprt () → ct () → MPI_Waitall ()`. The *right pane* shows the distribution of waiting times for the selected combination of wait state and call path across the virtual process topology. The display indicates that most wait states occur at the outer rim of a spherical region in the center of the three-dimensional Cartesian topology

profiling mode to better control runtime dilation, while still supporting global communication analyses [15].

## Measurement

Measurements are collected and analyzed under the control of a workflow manager that determines how the application should be run, and then configures measurement and analysis accordingly. When tracing is requested, it automatically configures and executes the parallel trace analyzer with the same number of processes as used for measurement. The following examples demonstrate how to request measurements from MPI application bar to be executed with 65,536 ranks, once

in profiling and once in tracing mode (distinguished by the use of the “-t” option).

```
> scalasca -analyze mpiexec \
 -np 65536 bar <arglist>
> scalasca -analyze -t mpiexec \
 -np 65536 bar <arglist>
```

## Call-Path Profiling

Scalasca can efficiently calculate many execution performance metrics by accumulating statistics during measurement, avoiding the cost of storing them with events for later analysis. For example, elapsed times and hardware-counter metrics for source regions

(e.g., routines or loops) can be immediately determined and the differences accumulated. Whereas trace storage requirements increase in proportion to the number of events (dependent on the measurement duration), summarized statistics for a call-path profile per thread have a fixed storage requirement (dependent on the number of threads and executed call paths).

In addition to call-path visit counts, execution times, and optional hardware counter metrics, Scalasca profiles include various MPI statistics, such as the numbers of synchronization, communication and file I/O operations along with the associated number of bytes transferred. Each metric is broken down into collective versus point-to-point/individual, sends/writes versus receives/reads, and so on. Call-path execution times separate MPI message-passing and OpenMP multithreading costs from purely local computation, and break them down further into initialization/finalization, synchronization, communication, file I/O and thread management overheads (as appropriate). For measurements using OpenMP, additional thread idle time and limited parallelism metrics are derived, assuming a dedicated core for each thread.

Scalasca provides accumulated metric values for every combination of call path and thread. A call path is defined as the list of code regions entered but not yet left on the way to the currently active one, typically starting from the main function, such as in the call chain `main() → foo() → bar()`. Which regions actually appear on a call path depends on which regions have been instrumented. When execution is complete, all locally executed call paths are combined into a global dynamic call tree for interactive exploration (as shown in the middle of Fig. 2, although the screen shot actually visualizes a wait-state report).

## Wait-State Analysis

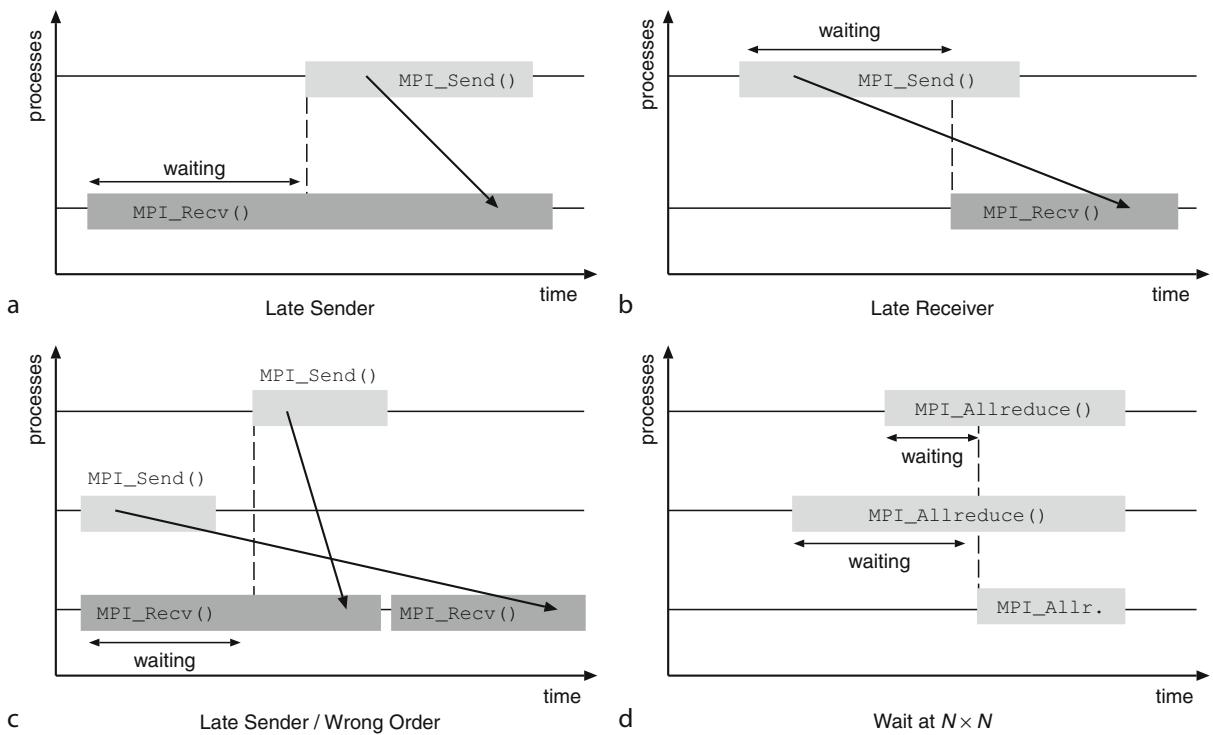
In message-passing applications, processes often require access to data provided by remote processes, making the progress of a receiving process dependent upon the progress of a sending process. If a rendezvous protocol is used, this relationship also applies in the opposite direction. Collective synchronization is similar in that its completion requires each participating process to have reached a certain point. As a consequence, a significant fraction of the time spent in communication and synchronization routines can often be attributed to wait

states that occur when processes fail to reach implicit or explicit synchronization points in a timely manner. Scalasca provides a diagnostic method that allows the localization of wait states by automatically searching event traces for characteristic patterns.

Figure 3 shows several examples of wait states that can occur in message-passing programs. The first one is the *Late Sender* pattern (Fig. 3a), where a receiver is blocked while waiting for a message to arrive. That is, the receive operation is entered by the destination process before the corresponding send operation has been entered by the source process. The waiting time lost as a consequence is the time difference between entering the send and the receive operations. Conversely, the *Late Receiver* pattern (Fig. 3b) describes a sender that is likely to be blocked while waiting for the receiver when a rendezvous protocol is used. This can happen for several reasons. Either the MPI implementation is working in synchronous mode by default, or the size of the message to be sent exceeds the available MPI-internal buffer space and the operation is blocked until the data is transferred to the receiver. The *Late Sender / Wrong Order* pattern (Fig. 3c) describes a receiver waiting for a message, although an earlier message is ready to be received by the same destination process (i.e., messages received in the wrong order). Finally, the *Wait at N×N* pattern (Fig. 3d) quantifies the waiting time due to the inherent synchronization in  $n$ -to- $n$  operations, such as `MPI_Allreduce()`. A full list of the wait-state types supported by Scalasca including explanatory diagrams can be found online in the Scalasca documentation [11].

## Parallel Wait-State Search

To accomplish the search in a scalable way, Scalasca exploits both distributed memory and parallel processing capabilities available on the target system. After the target application has terminated and the trace data have been flushed to disk, the trace analyzer is launched with one analysis process per (target) application process and loads the entire trace data into its distributed memory address space. Future versions of Scalasca may exploit persistent memory segments available on systems such as Blue Gene/P to pass the trace data to the analysis stage without involving any file I/O. While traversing the traces in parallel, the analyzer performs a replay of the application's original communication. During the replay, the analyzer



**Scalasca.** Fig. 3 Four examples of wait states (a–d) detected by Scalasca. The combination of MPI functions used in each of these examples represents just one possible case. For instance, the blocking receive operation in wait state (a) can be replaced by a non-blocking receive followed by a wait operation. In this case, the waiting time would occur during the wait operation

identifies wait states in communication and synchronization operations by measuring temporal differences between local and remote events after their time stamps have been exchanged using an operation of similar type. Detected wait-state instances are classified and quantified according to their significance for every call path and system resource involved. Since trace processing capabilities (i.e., processors and memory) grow proportionally with the number of application processes, good scalability can be achieved even at previously intractable scales. Recent scalability improvements allowed Scalasca to complete trace analyses of runs with up to 294,912 cores on a 72-rack IBM Blue Gene/P system [23].

A modified form of the replay-based trace analysis scheme is also applied to detect wait states occurring in MPI-2 RMA operations. In this case, RMA communication is used to exchange the required information between processes. Finally, Scalasca also

provides the ability to process traces from hybrid MPI/OpenMP and pure OpenMP applications. However, the parallel wait-state search does not yet recognize OpenMP-specific wait states, such as barrier waiting time or lock contention, previously supported by its predecessor.

#### Wait-State Search on Clusters without Global Clock

To allow accurate trace analyses on systems without globally synchronized clocks, Scalasca can synchronize inaccurate time stamps postmortem. Linear interpolation based on clock offset measurements during program initialization and finalization already accounts for differences in offset and drift, assuming that the drift of an individual processor is not time dependent. This step is mandatory on all systems without a global clock, such as Cray XT and most PC or compute blade clusters. However, inaccuracies and drifts

varying over time can still cause violations of the logical event order that are harmful to the accuracy of the analysis. For this reason, Scalasca compensates for such violations by shifting communication events in time as much as needed to restore the logical event order while trying to preserve the length of intervals between local events. This logical synchronization is currently optional and should be performed if the trace analysis reports (too many) violations of the logical event order.

## Future Directions

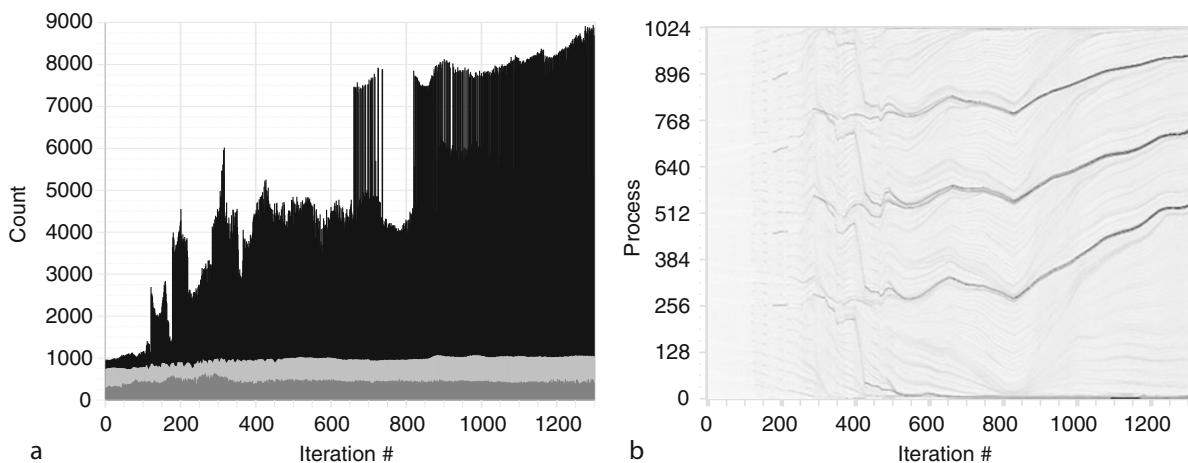
Future enhancements of Scalasca will aim at further improving both its functionality and its scalability. In addition to supporting the more advanced features of OpenMP such as nested parallelism and tasking as an immediate priority, Scalasca is expected to evolve toward emerging programming models and architectures including partitioned global address space (PGAS) languages and heterogeneous systems. Moreover, optimized data management and analysis workflows including in-memory trace analysis will allow Scalasca to master even larger processor configurations than it does today. A recent example in this direction is the substantial reduction of the trace-file creation overhead that was achieved by mapping large numbers of logical process-local trace files onto a small number of

physical files [4], a feature that will become available in future releases of Scalasca.

In addition to keeping up with the rapid new developments in parallel hardware and software, research is also undertaken to expand the general understanding of parallel performance in simulation codes. The examples below summarize two ongoing projects aimed at increasing the expressive power of the analyses supported by Scalasca. The description reflects the status of March 2011.

## Time-Series Call-Path Profiling

As scientific parallel applications simulate the temporal evolution of a system, their progress occurs via discrete points in time. Accordingly, the core of such an application is typically a loop that advances the simulated time step by step. However, the performance behavior may vary between individual iterations, for example, due to periodically reoccurring extra activities or when the state of the computation adjusts to new conditions in so-called adaptive codes. To study such time-dependent behavior, Scalasca can distinguish individual iterations in profiles and event traces. Figure 4 shows the distribution of point-to-point messages across the iteration-process space in the MPI-based PEPC [7] particle simulation code. Obviously, during later stages of the simulation the majority of messages are sent by only a small collection of processes with time-dependent



**Scalasca. Fig. 4** Gradual development of a communication imbalance along 1,300 timesteps of PEPC on 1,024 processors. (a) Minimum (bottom), median (middle), and maximum (top) number of point-to-point messages sent or received by a process in an iteration. (b) Number of messages sent by each process in each iteration. The higher the number of messages the darker the color on the value map

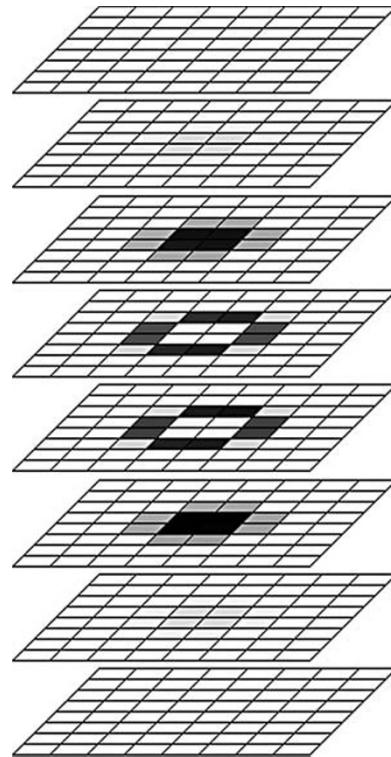
constituency, inducing wait states on other processes (not shown).

However, even generating call-path profiles (as opposed to traces) separately for thousands of iterations to identify the call paths responsible may exceed the available buffer space – especially when the call tree is large and more than one metric is collected. For this reason, a runtime approach for the semantic compression of a series of call-path profiles based on incrementally clustering single-iteration profiles was developed that scales in terms of the number of iterations without sacrificing important performance details [16]. This method, which will be integrated in future versions of Scalasca, offers low runtime overhead by using only a condensed version of the profile data when calculating distances and accounts for process-dependent variations by making all clustering decisions locally.

## Identifying the Root Causes of Wait-State Formation

In general, the temporal or spatial distance between cause and symptom of a performance problem constitutes a major difficulty in deriving helpful conclusions from performance data. Merely knowing the locations of wait states in the program is often insufficient to understand the reason for their occurrence. Building on earlier work by Meira, Jr. et al. [12], the replay-based wait-state analysis was extended in such a way that it attributes the waiting times to their root causes [3], as exemplified in Fig. 5. Typically, these root causes are intervals during which a process performs some additional activity not performed by its peers, for example as a result of insufficiently balancing the load.

However, excess workload identified as the root cause of wait states usually cannot simply be removed. To achieve a better balance, optimization hypotheses drawn from such an analysis typically propose the redistribution of the excess load to other processes instead. Unfortunately, redistributing workloads in complex message-passing applications can have surprising side effects that may compromise the expected reduction of waiting times. Given that balancing the load statically or even introducing a dynamic load-balancing scheme constitute major code changes, such procedures should ideally be performed only if the prospective performance gain is likely to materialize. Other recent



**Scalasca.** Fig. 5 In the `lorentz_d()` subroutine of the Zeus MP/2 code, several processes primarily arranged on a small hollow sphere within the virtual process topology are responsible for wait states arranged on the enclosing hollow sphere shown earlier in Fig. 2. Since the inner region of the topology carries more load than the outer region, processes at the rim of the inner region delay those farther outside. The darkness of the color indicates the amount of waiting time induced by a process but materializing farther away. In this way, it becomes possible to study the root causes of wait-state formation

work [10] therefore concentrated on determining the savings we can realistically hope for when redistributing a given delay – before altering the application itself. Since the effects of such changes are hard to quantify analytically, they are simulated in a scalable manner via a parallel real-time replay of event traces after they have been modified to reflect the redistributed load.

## Related Entries

- ▶ [Intel® Thread Profiler](#)
- ▶ [MPI \(Message Passing Interface\)](#)
- ▶ [OpenMP](#)

- ▶ [OpenMP Profiling with OmpP](#)
- ▶ [Performance Analysis Tools](#)
- ▶ [Periscope](#)
- ▶ [PMPI Tools](#)
- ▶ [Synchronization](#)
- ▶ [TAU](#)
- ▶ [Vampir](#)

## Bibliographic Notes and Further Reading

Scalasca is available for download including documentation under the New BSD license at [www.scalasca.org](http://www.scalasca.org).

Scalasca emerged from the KOJAK project, which was started in 1998 at the Jülich Supercomputing Centre in Germany to study the automatic evaluation of parallel performance data, and in particular, the automatic detection of wait states in event traces of parallel applications. The wait-state analysis first concentrated on MPI [20] and later on OpenMP and hybrid codes [21], motivating the definition of the POMP profiling interface [13] for OpenMP, which is still used today even beyond Scalasca by OpenMP-enabled profilers such as ompP (▶ [OpenMP Profiling with OmpP](#)) and ▶ [TAU](#). A comprehensive description of the initial trace-analysis toolset resulting from this effort, which was publicly released for the first time in 2003 under the name KOJAK, is given in [19].

During the following years, KOJAK's wait-state search was optimized for speed, refined to exploit virtual process topologies, and extended to support MPI-2 RMA communication. In addition to the detection of wait states occurring during a single run, KOJAK also introduced a framework for comparing the analysis results of different runs [14], for example, to judge the effectiveness of optimization measures. An extensive snapshot of this more advanced version of KOJAK including further literature references is presented in [22]. However, KOJAK still analyzed the traces sequentially after the process-local trace data had been merged into a single global trace file, an undesired scalability limitation in view of the dramatically rising number of cores employed on modern parallel architectures.

In 2006, after the acquisition of a major grant from the Helmholtz Association of German Research Centres, the Scalasca project was started in Jülich as the successor to KOJAK with the objective of improving

the scalability of the trace analysis by parallelizing the search for wait states. A detailed discussion of the parallel replay underlying the parallel search can be found in [6]. Variations of the scalable replay mechanism were applied to correct event time stamps taken on clusters without global clock [1], to simulate the effects of optimizations such as balancing the load of a function more evenly across the processes of a program [10], and to identify wait states in MPI-2 RMA communication in a scalable manner [9]. Moreover, the parallel trace analysis was also demonstrated to run on computational grids consisting of multiple geographically dispersed clusters that are used as a single coherent system [2]. Finally, a very recent replay-based method attributes the costs of wait states in terms of resource waste to their original cause [3].

Since the enormous data volume sometimes makes trace analysis challenging, runtime summarization capabilities were added to Scalasca both as a simple means to obtain a performance overview and as a basis to optimally configure the measurement for later trace generation. Scalasca integrates both measurement options in a unified tool architecture, whose details are described in [5]. Recently, a semantic compression algorithm was developed that will allow Scalasca to take time-series profiles in a space-efficient manner even if the target application performs large numbers of timesteps [16].

Major application studies with Scalasca include a survey of using it on leadership systems [24], a comprehensive analysis of how the performance of the SPEC MPI2007 benchmarks evolves as their execution progresses [17], and the investigation of a gradually developing communication imbalance in the PEPC particle simulation [18]. Finally, a recent study of the Sweep3D benchmark demonstrated performance measurements and analyses with up to 294,912 processes [23].

From 2003 until 2008, KOJAK and later Scalasca was jointly developed together with the Innovative Computing Laboratory at the University of Tennessee. During their lifetime, the two projects received funding from the Helmholtz Association of German Research Centres, the US Department of Energy, the US Department of Defense, the US National Science Foundation, the German Science Foundation, the German Federal Ministry of Education and Research, and the European Union. Today, Scalasca is a joint project between Jülich

and the German Research School for Simulation Sciences in nearby Aachen.

The following individuals have contributed to Scalasca and its predecessor: Erika Ábrahám, Daniel Becker, Nikhil Bhatia, David Böhme, Jack Dongarra, Dominic Eschweiler, Sebastian Flott, Wolfgang Frings, Karl Fürlinger, Christoph Geile, Markus Geimer, Marc-André Hermanns, Michael Knobloch, David Krings, Guido Kruschwitz, André Kühnla, Björn Kuhlmann, John Linford, Daniel Lorenz, Bernd Mohr, Shirley Moore, Ronald Muresano, Jan Mußler, Andreas Nett, Christian Rössel, Matthias Pfeifer, Peter Philippen, Farzana Pulatova, Divya Sankaranarayanan, Pavel Savankou, Marc Schlüter, Christian Siebert, Feng-guang Song, Alexandre Strube, Zoltán Szebenyi, Felix Voigtländer, Felix Wolf, and Brian Wylie.

## Bibliography

1. Becker D, Rabenseifner R, Wolf F, Linford J (2009) Scalable timestamp synchronization for event traces of message-passing applications. *Parallel Comput* 35(12):595–607
2. Becker D, Wolf F, Frings W, Geimer M, Wylie B, Mohr B (2007) Automatic trace-based performance analysis of metacomputing applications. In: Proceedings of the international parallel and distributed processing symposium (IPDPS), Long Beach, CA, USA. IEEE Computer Society, Washington, DC
3. Böhme D, Geimer M, Wolf F, Arnold L (2010) Identifying the root causes of wait states in large-scale parallel applications. In: Proceedings of the 39th international conference on parallel processing (ICPP), San Diego, CA, USA. IEEE Computer Society, Washington, DC, pp 90–100
4. Frings W, Wolf F, Petkov V (2009) Scalable massively parallel I/O to task-local files. In: Proceedings of the ACM/IEEE conference on supercomputing (SC09), Portland, OR, USA, Nov 2009
5. Geimer M, Wolf F, Wylie B, Ábrahám E, Becker D, Mohr B (2010) The Scalasca performance toolset architecture. *Concurr Comput Pract Exper* 22(6):702–719
6. Geimer M, Wolf F, Wylie B, Mohr B (2009) A scalable tool architecture for diagnosing wait states in massively-parallel applications. *Parallel Comput* 35(7):375–388
7. Gibbon P, Frings W, Dominiczak S, Mohr B (2006) Performance analysis and visualization of the n-body tree code PEPC on massively parallel computers. In: Proceedings of the conference on parallel computing (ParCo), Málaga, Spain, Sept 2005 (NIC series), vol 33. John von Neumann-Institut für Computing, Jülich, pp 367–374
8. Hayes JC, Norman ML, Fiedler RA, Bordner JO, Li PS, Clark SE, ud-Doula A, Mac Low M-M (2006) Simulating radiating and magnetized flows in multiple dimensions with ZEUS-MP. *Astrophys J Suppl* 165(1):188–228
9. Hermanns M-A, Geimer M, Mohr B, Wolf F (2009) Scalable detection of MPI-2 remote memory access inefficiency patterns. In: Proceedings of the 16th European PVM/MPI users' group meeting (EuroPVM/MPI), Espoo, Finland. Lecture notes in computer science, vol 5759. Springer, Berlin, pp 31–41
10. Hermanns M-A, Geimer M, Wolf F, Wylie B, Mohr B (2009) Verifying causality between distant performance phenomena in large-scale MPI applications. In: Proceedings of the 17th Euromicro international conference on parallel, distributed, and network-based processing (PDP), Weimar, Germany. IEEE Computer Society, Washington, DC, pp 78–84
11. Jülich Supercomputing Centre and German Research School for Simulation Sciences. Scalasca parallel performance analysis toolset documentation (performance properties). <http://www.scalasca.org/download/documentation/>
12. Meira W Jr, LeBlanc TJ, Poulos A (1996) Waiting time analysis and performance visualization in Carnival. In: Proceedings of the SIGMETRICS symposium on parallel and distributed tools (SPDT'96), Philadelphia, PA, USA. ACM
13. Mohr B, Malony A, Shende S, Wolf F (2002) Design and prototype of a performance tool interface for OpenMP. *J Supercomput* 23(1):105–128
14. Song F, Wolf F, Bhatia N, Dongarra J, Moore S (2004) An algebra for cross-experiment performance analysis. In: Proceedings of the international conference on parallel processing (ICPP), Montreal, Canada. IEEE Computer Society, Washington, DC, pp 63–72
15. Szebenyi Z, Gamblin T, Schulz M, de Supinski BR, Wolf F, Wylie B (2011) Reconciling sampling and direct instrumentation for unintrusive call-path profiling of MPI programs. In: Proceedings of the international parallel and distributed processing symposium (IPDPS), Anchorage, AK, USA. IEEE Computer Society, Washington, DC
16. Szebenyi Z, Wolf F, Wylie B (2009) Space-efficient time-series call-path profiling of parallel applications. In: Proceedings of the ACM/IEEE conference on supercomputing (SC09), Portland, OR, USA, Nov 2009
17. Szebenyi Z, Wylie B, Wolf F (2008) SCALASCA parallel performance analyses of SPEC MPI2007 applications. In: Proceedings of the 1st SPEC international performance evaluation workshop (SIPEW), Darmstadt, Germany. Lecture notes in computer science, vol 5119. Springer, Berlin, pp 99–123
18. Szebenyi Z, Wylie B, Wolf F (2009) Scalasca parallel performance analyses of PEPC. In: Proceedings of the workshop on productivity and performance (PROPER) in conjunction with Euro-Par, Las Palmas de Gran Canaria, Spain, August 2008. Lecture notes in computer science, vol 5415. Springer, Berlin, pp 305–314
19. Wolf F (2003) Automatic Performance Analysis on Parallel Computers with SMP Nodes. PhD thesis, RWTH Aachen, Forschungszentrum Jülich. ISBN 3-00-010003-2
20. Wolf F, Mohr B (2001) Specifying performance properties of parallel applications using compound events. *Parallel Distrib Comput Pract* 4(3):301–317
21. Wolf F, Mohr B (2003) Automatic performance analysis of hybrid MPI/OpenMP applications. *J Syst Archit* 49(10–11):421–439
22. Wolf F, Mohr B, Dongarra J, Moore S (2007) Automatic analysis of inefficiency patterns in parallel applications. *Concurr Comput Pract Exper* 19(11):1481–1496

23. Wylie BJN, Geimer M, Mohr B, Böhme D, Szebenyi Z, Wolf F (2010) Large-scale performance analysis of Sweep3D with the Scalasca toolset. *Parallel Process Lett* 20(4):397–414
24. Wylie BJN, Geimer M, Wolf F (2008) Performance measurement and analysis of large-scale parallel applications on leadership computing systems. *Sci Program* 16(2–3):167–181

## Scaled Speedup

► Gustafson's Law

## Scan for Distributed Memory, Message-Passing Systems

JESPER LARSSON TRÄFF  
University of Vienna, Vienna, Austria

### Synonyms

All prefix sums; Prefix; Parallel prefix sums; Prefix reduction

### Definition

Among a group of  $p$  consecutively numbered processing elements (nodes) each node has a data item  $x_i$  for  $i = 0, \dots, p - 1$ . An associative, binary operator  $\oplus$  over the data items is given. The nodes in parallel compute all  $p$  (or  $p - 1$ ) *prefix sums* over the input items: node  $i$  computes the *inclusive prefix sum*  $\oplus_{j=0}^i x_j$  (or, for  $i > 0$ , the *exclusive prefix sum*  $\oplus_{j=0}^{i-1} x_j$ ).

### Discussion

For a general discussion, see the entry on “► Reduce and Scan.”

For distributed memory, message-passing parallel systems, the scan operation plays an important role for load-balancing and data-redistribution operations. In this setting, each node possesses a local data item  $x_i$ ,

typically an  $n$ -element vector, and the given associative, binary operator  $\oplus$  is typically an element-wise operation. The required prefix sums could be computed by arranging the input items in a linear sequence and “scanning” through this sequence in order, carrying along the corresponding inclusive (or exclusive) prefix sum. In a distributed memory setting, this intuition would be captured by arranging the nodes in a linear array, but much better algorithms usually exist. The inclusive scan operation is illustrated in Fig. 1.

The scan operation is included as a collective operation in many interfaces and libraries for distributed memory systems, for instance MPI [5], where both an inclusive (`MPI_Scan`) and an exclusive (`MPI_Exscan`) general scan operation is defined. These operations work on vectors of consecutive or non-consecutive elements and arbitrary (user-defined) associative, possibly also commutative operators.

### Algorithms

On distributed memory parallel architectures parallel algorithms typically use tree-like patterns to compute the prefix sums in a logarithmic number of parallel communication rounds. Four such algorithms are described in the following. Algorithms also exist for hypercube and mesh- or torus-connected communication architectures [1, 4, 8].

The  $p$  nodes are consecutively numbered from 0 to  $p - 1$ . Each node  $i$  has an input data item  $x_i$ . Nodes must communicate explicitly by send and receive operations. For convenience a fully connected model in which each node can communicate with all other nodes at the same cost is assumed. Only one communication operation per node can take place at a time. The data items  $x_i$  are assumed to all have the same size  $n = |x_i|$ . For scalar items,  $n = 1$ , while for vector items  $n$  is the number of elements. The communication cost of transferring a data item of size  $n$  is assumed to be  $O(n)$  (linear).

### Linear Array

The simplest scan algorithm organizes the nodes in a linear array. For  $i > 0$  node  $i$  waits for a *partial sum*

| Before    |           |           | After                    |                          |                          |
|-----------|-----------|-----------|--------------------------|--------------------------|--------------------------|
| Node 0    | Node 1    | Node 2    | Node 0                   | Node 1                   | Node 2                   |
| $x^{(0)}$ | $x^{(1)}$ | $x^{(2)}$ | $\oplus_{j=0}^0 x^{(j)}$ | $\oplus_{j=0}^1 x^{(j)}$ | $\oplus_{j=0}^2 x^{(j)}$ |

Scan for Distributed Memory, Message-Passing Systems. Fig. 1 The inclusive scan operation

$\oplus_{j=0}^{i-1} x_j$  from node  $i - 1$ , adds its own item  $x_i$  to arrive at the partial sum  $\oplus_{j=0}^i x_j$ , and sends this to node  $i + 1$ , unless  $i = p - 1$ . This algorithm is strictly serial and takes  $p - 1$  communication rounds for the last node to finish its prefix computation, for a total cost of  $O(pn)$ . This is uninteresting in itself, but the algorithm can be pipelined (see below) to yield a time to complete of  $O(p + n)$ . When  $n$  is large compared to  $p$  this easily implementable algorithm is often the fastest, due to its simplicity.

### Binary Tree

The *binary tree* algorithm arranges the nodes in a balanced binary tree  $T$  with in-order numbering. This numbering has the property that the nodes in the subtree  $T(i)$  rooted at node  $i$  have consecutive numbers in the interval  $[s, \dots, i, \dots, e]$ , where  $s$  and  $e$  denote the first (start) and last (end) node in the subtree  $T(i)$ , respectively. The algorithm consists of two phases. It suffices to describe the actions of a node in the tree with parent, right and left children. In the *up-phase*, node  $i$  first receives the *partial result*  $\oplus_{j=s}^{i-1} x_j$  from its left child and adds its own item  $x_i$  to get the partial result  $\oplus_{j=s}^i x_j$ . This value is stored for the *down-phase*. Node  $i$  then receives the partial result  $\oplus_{j=i+1}^e x_j$  from its right child and computes the partial result  $\oplus_{j=s}^e x_j$ . Node  $i$  sends this value upward in the tree without keeping it. In the *down-phase*, node  $i$  receives the partial result  $\oplus_{j=0}^{s-1} x_j$  from its parent. This is first sent down to the left child and then added to the stored partial result  $\oplus_{j=s}^i x_j$  to form the final result  $\oplus_{j=0}^i x_j$  for node  $i$ . This final result is sent down to the right child.

With the obvious modifications, the general description covers also nodes that need not participate in all of these communications: Leaves have no children. Some nodes may only have a leftmost child. Nodes on the path between root and leftmost leaf do not receive data from their parent in the *down-phase*. Nodes on the path between rightmost child and root do not send data to their parent in the *up-phase*.

Since the depth of  $T$  is logarithmic in the number of nodes, and data of size  $n$  are sent up and down the tree, the time to complete the scan with the binary tree algorithm is  $O(n \log p)$ . This is unattractive for large  $n$ . If the nodes can only either send or receive in a communication operation but otherwise work simultaneously, the number of communication operations per node is

at most 6, and the number of communication rounds is  $6\lceil\log_2 p\rceil$ .

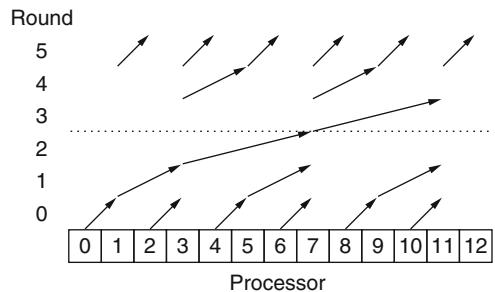
### Binomial Tree

The *binomial tree* algorithm likewise consists of an *up-phase* and a *down-phase*, each of which takes  $k$  rounds where  $k = \lceil\log_2 p\rceil$ . In round  $j$  for  $j = 0, \dots, k - 1$  of the up-phase each node  $i$  satisfying  $i \wedge (2^{j+1} - 1) \equiv 2^{j+1} - 1$  (where  $\wedge$  denotes “bitwise and”) receives a partial result from node  $i - 2^j$  (provided  $0 \leq i - 2^j$ ). After sending to node  $i$ , node  $i - 2^j$  is inactive for the remainder of the up-phase. The receiving nodes add the partial results, and after round  $j$  have a partial result of the form  $\oplus_{\ell=i-2^{j+1}+1}^i x_\ell$ . The down-phase counts rounds downward from  $k$  to 1. Node  $i$  with  $i \wedge (2^j - 1) \equiv 2^j - 1$  sends its partial result to node  $i + 2^{j-1}$  (provided  $i + 2^{j-1} < p$ ) which can now compute its final result  $\oplus_{\ell=0}^{i+2^{j-1}} x_\ell$ . The communication pattern is illustrated in Fig. 2.

The number of communication rounds is  $2\lceil\log p\rceil$ , and the time to complete is  $O(n \log p)$ . Each node is either sending or receiving data in each round, with no possibility for overlapping of sending and receiving due to the computation of partial results.

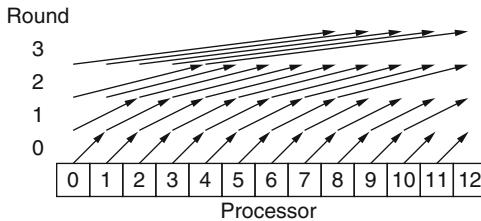
### Simultaneous Trees

The number of communication rounds can be improved by a factor of two by the *simultaneous binomial tree* algorithm. Starting from round  $k = 0$ , in round  $k$ , node  $i$  sends a computed, partial result  $\oplus_{j=i-2^k+1}^i x_j$  to node  $i + 2^k$  (provided  $i + 2^k < p$ ) and receives a likewise computed, partial result from node  $i - 2^k$  (provided  $i - 2^k \geq 0$ ). Before the next round, the previous and received partial



Scan for Distributed Memory, Message-Passing Systems.

Fig. 2 The communication pattern for the binomial tree algorithm for  $p = 13$



#### Scan for Distributed Memory, Message-Passing Systems.

**Fig. 3** The communication patterns for the simultaneous binomial tree algorithm for  $p = 13$

results are added. It is easy to see that after round  $k$ , node  $i$ 's partial result is  $\oplus_{j=\max(0,i-2^{k+1}+1)}^i x_j$ . Node  $i$  terminates when both  $i - 2^k < 0$  (nothing to receive) and  $i + 2^k \geq p$  (nothing to send). Provided that the nodes can simultaneously send and receive an item in the same communication operation this happens after  $\lceil \log_2 p \rceil$  communication rounds for a total time of  $O(n \log p)$ . This is again not attractive for large  $n$ . This algorithm dates back at least to [2] and is illustrated in Fig. 3.

For large  $n$  the running time of  $O(n \log p)$  is not attractive due to the logarithmic factor. For the binary tree algorithm pipelining can be employed, by which the running time can be improved to  $O(n + \log p)$ . Neither the binomial nor the simultaneous binomial tree algorithms admit pipelining. A pipelined implementation breaks each data item into a certain number of blocks  $N$  each of roughly equal size of  $n/N$ . As soon as a block has been processed and sent down the pipeline, the next block can be processed. The time of such an implementation is proportional to the depth of the pipeline plus the number of blocks times the time to process one block. From this an optimal block size can be determined, which gives the running time claimed for the binary tree algorithm. For very large vectors relative to the number of nodes,  $n \gg p$ , a linear pipeline seems to be the best practical choice due to very small constants. Pipelining can only be applied if the binary operator can work independently on the blocks. This is trivially the case if the items are vectors and the operator works element-wise on these.

#### Related Entries

- [Collective Communication](#)
- [Message Passing Interface \(MPI\)](#)
- [Reduce and Scan](#)

#### Bibliographic Notes and Further Reading

The parallel prefix problem was early recognized as a fundamental parallel processing primitive, and its history goes back at least as far as the early 1970s. The simultaneous binomial tree algorithm is from [2], but may have been discovered earlier as well. The binary tree algorithm easily admits pipelining but has the drawback that leaf nodes are only using either send or receive capabilities, and that simultaneous send-receive communication can be only partially exploited. In [6] it was shown how to overcome these limitations, yielding theoretically almost optimal scan (as well as reduce and broadcast) algorithms with (two) binary trees. Algorithms for distributed systems in the  $\text{LogP}$  performance model were presented in [7]. An algorithm for multiport message-passing systems was given in [3]. For results on meshes and hypercubes see, for instance [1, 4, 8].

#### Bibliography

1. Akl SG (1999) Parallel computation: models and methods. Prentice-Hall, Upper Saddle River
2. Daniel Hillis W, Steele GL Jr (1986) Data parallel algorithms. Commun ACM 29(12):1170–1183
3. Lin Y-C, Yeh C-S (1999) Efficient parallel prefix algorithms on multiport message-passing systems. Inform Process Lett 71:91–95
4. Mayr EW, Plaxton CG (1993) Pipelined parallel prefix computations, and sorting on a pipelined hypercube. J Parallel Distrib Comput 17:374–380
5. MPI Forum (2009) MPI: A message-passing interface standard. Version 2.2. [www.mpiforum.org](http://www.mpiforum.org). Accessed 4 Sept 2009
6. Sanders P, Speck J, Träff JL (2009) Two-tree algorithms for full bandwidth broadcast, reduction and scan. Parallel Comput 35:581–594
7. Santos EE (2002) Optimal and efficient algorithms for summing and prefix summing on parallel machines. J Parallel Distrib Comput 62(4):517–543
8. Ziavras SG, Mukherjee A (1996) Data broadcasting and reduction, prefix computation, and sorting on reduces hypercube parallel computer. Parallel Comput 22(4):595–606

#### Scan, Reduce and

- [Reduce and Scan](#)

## Scatter

► [Collective Communication](#)

## Scheduling

► [Job Scheduling](#)  
 ► [Task Graph Scheduling](#)  
 ► [Scheduling Algorithms](#)

## Scheduling Algorithms

PATRICE QUINTON  
 ENS Cachan Bretagne, Bruz, France

### Synonyms

[Execution ordering](#)

### Definition

Scheduling algorithms aim at defining when operations of a program are to be executed. Such an ordering, called a *schedule*, has to make sure that the dependences between the operations are met. Scheduling for parallelism consists in looking for a schedule that allows a program to be efficiently executed on a parallel architecture. This efficiency may be evaluated in term of total execution time, utilization of the processors, power consumption, or any combination of this kind of criteria. Scheduling is often combined with *mapping*, which consists in assigning an operation to a resource of an architecture.

### Discussion

#### Introduction

Scheduling is an essential design step to execute an algorithm on a parallel architecture. Usually, the algorithm is specified by means of a program from which dependences between statements or operations (also called *tasks*) can be isolated: an operation A depends

on another operation B if the evaluation of B must precede that of A for the result of the algorithm to be correct.

Assuming the dependences of an algorithm are already known, scheduling has to find out a partial order, called a *schedule*, which says when each statement can be executed. A schedule may be resource-constrained, if the ordering must be such that some resource – for example, processor unit, memory, etc. – is available for the execution of the algorithm.

Scheduling algorithms are therefore classified according to the precision of the dependence analysis, the kind of resources needed, the optimization criteria considered, the specification of the algorithm, the granularity size of the tasks, and the kind of parallel architecture that is considered for execution.

Scheduling can also happen at run-time – *dynamic scheduling* – or at compile-time – *static scheduling*.

In the following, basic notions related to task graphs are first recalled, then scheduling algorithms for loops are presented. The presentation separates mono-dimensional and higher-dimensional loops. *Task Graph scheduling*, a topic that is covered in another essay of this site, is not treated here.

#### Task Graphs and Scheduling

Task graphs and their scheduling is the basis of many situations of parallel computations and has been the subject of a number of researches. Task graphs are a means of representing dependences between computations of a program: vertices  $V$  of the graph are elementary computations, called tasks, and oriented edges  $E$  represent dependences between computations. A task graph is therefore an acyclic, directed multigraph.

Assume that each task  $T$  has an estimated integer value  $d(T)$  representing its execution duration. A schedule for a task graph is a function  $\sigma$  from the vertex set  $V$  of the graph to the set of positive integers, such that  $\sigma(u) + d(u) \leq \sigma(v)$ , where  $d$  represents the duration associated to the evaluation of task  $u$ , and  $v$  depends on  $u$ . Since it is acyclic, a task graph always admits a schedule.

Scheduling a task graph may be constrained by the number of processors  $p$  available to run the tasks. It is assumed that each task  $T$ , when ready to be evaluated, is allocated to a free processor, and occupies this processor

during its duration  $d(T)$ , after which the processor is released and becomes idle. When  $p$  is unlimited, we say that the schedule is *resource free*, otherwise, it is *resource constrained*.

The total computation time for executing a task graph on  $p$  processors, is also called its *makespan*. If the number of processors is illimited, that is,  $p = +\infty$ , it can be shown that finding out the makespan amounts to a simple traversal of the task graph.

When the number of processors is bounded, that is,  $p < +\infty$ , it can be shown that the problem becomes NP-complete. It is therefore required to rely on heuristics. In this case, a simple *greedy* heuristics is efficient: try to schedule at a given time as many tasks as possible on processors that are available at this time.

A *list schedule* is a schedule such that no processor is deliberately kept idle; at each time step  $t$ , if a processor  $P$  is available, and if some task  $T$  is *free*, that is to say, when all its predecessors have been executed, then  $T$  is scheduled to be executed on processor  $P$ . It can be shown that a list schedule allows a solution within 50% of the optimal makespan to be found.

List scheduling plays a very important rôle in scheduling theory and is therefore worth mentioning. A generic *list scheduling* algorithm looks as follows:

1. Initialization
  - a. Assign to all free tasks a priority level (the choice of the priority function depends on the problem at hand).
  - b. Set the time step  $t$  to 0.
2. While there remain tasks to execute:
  - a. If the execution of a task terminates at time  $t$ , suppress this task from the predecessor list of all its successors. Add those tasks whose predecessor tasks has become empty to the free tasks, and assign them a priority.
  - b. If there are  $q$  available processors, and  $r$  free tasks, execute the  $\min(q, r)$  tasks of highest priority.
  - c. Increase the time step  $t$  by one.

## Scheduling One-Dimensional Loops

A particular field of interest for scheduling is how to execute efficiently a set of computations that re-execute a large amount of time, also called *cyclic scheduling*.

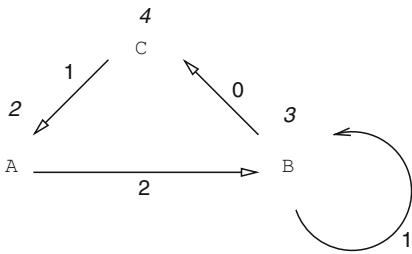
Such a problem is encountered when scheduling a one-dimensional loop. Consider for example, the following loop:

```
do $k = 0$, step 0, to $N - 1$
 A : $a(k) = c(k - 1)$
 B : $b(k) = a(k - 2) \times b(k - 1)$
 C : $c(k) = b(k) + 1$
endo
```

This loop could be modeled as a task graph where tasks would be  $A(k)$ ,  $B(k)$ , and  $C(k)$ ,  $k = 0, \dots, N - 1$ . But if  $N$  is large, this graph becomes difficult to handle, hence the idea of finding out a generic way to schedule the operations by following the structure of the loop.

Scheduling such a loop consists in finding out a time instant for each instruction to be executed, while respecting the dependences between these instructions. Call *operation* the execution of an instruction for a given value of index  $k$ , and denote  $A(k)$ ,  $B(k)$ , and  $C(k)$  these operations. Notice that  $A(k)$  depends on the value  $c(k - 1)$ , which is computed by operation  $C(k - 1)$  during the previous iteration of the loop. Similarly,  $B(k)$  depends on  $B(k - 1)$  and  $A(k - 2)$ ; finally,  $C(k)$  depends on  $B(k)$ , an operation of the same loop iteration. To model this situation, the notion of *reduced dependence graph*, which gather all information needed to schedule the loop, is introduced: its vertices are operation names (e.g., A, B, C etc.), and its edges represent dependences between operations. To each vertex  $v$ , associate an integer value  $d(v)$  that represents its execution duration, and to each edge, associate a positive integer  $w$  that is 0 if the dependence lies within an iteration, and the difference between the iterations number otherwise. Fig. 1 represents the reduced dependence graph of the above loop.

With this representation, a schedule becomes a function  $\sigma$  from  $V \times \mathbb{N}$  to  $\mathbb{N}$ , where  $\mathbb{N}$  denotes the set of integers;  $\sigma(v, k)$  gives the time at which instruction  $v$  of iteration  $k$  can be executed. Of course, a schedule must meet the dependence constraints, which can be expressed in the following way: for all edge  $e = (u, v)$  of the dependence graph, it is needed that  $\sigma(v, k + w(e)) \geq \sigma(u, k) + d(u)$ .



**Scheduling Algorithms.** Fig. 1 Reduced dependence graph of the loop

For this kind of problem, the performance of a schedule is defined as the *average cycle time*  $\lambda$  defined as

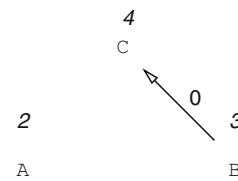
$$\lambda = \lim_{N \rightarrow \infty} \frac{\max\{\sigma(v, k) + d(v) \mid v \in V, 0 \leq k < N\}}{N}.$$

In order to define a *generic* schedule for such a loop, it is interesting to consider schedules of the form  $\sigma(v, k) = \lambda k + c(v)$  where  $\lambda$  is a positive integer and  $c(v)$  is a constant integer associated to instruction  $v$ . Constant  $\lambda$  is called the *initiation interval* of the loop, and it represents the number of cycles between the beginning of two successive iterations.

Finding out a schedule is quite easy: by restricting the reduced dependence graph to the intra-iteration dependences (i.e., dependences whose value  $w$  is 0), one obtains an acyclic graph that can be scheduled using a list scheduling algorithm. Let then  $\lambda$  be the makespan of such schedule, and let  $ls(v)$  be the schedule assigned to vertex  $v$  by the list scheduling; then, a schedule of the form  $\sigma(v, k) = \lambda k + ls(v)$  is a solution to the problem. Fig. 2 shows, for example, the task graph obtained when removing nonzero inter-iteration dependencies. One can see that the makespan of this graph is 7, and therefore, a schedule of the form  $\sigma(v, k) = 7k + ls(v)$  is a solution. Figure 3 shows a possible execution, on two processors, with an initiation interval of 7.

However, one can do better in general, as shall be seen. Again, the *cyclic scheduling* problem can be stated with or without resources. Assuming, for the sake of simplicity, that instructions are carried out by a set of  $p$  identical processing units, the number of available processors constitutes a resource limit.

First, one can give a lower bound for the *initiation interval*. Let  $\mathcal{C}$  be a cycle of the reduced dependence graph, and let  $d(\mathcal{C})$  be its duration (the sum of the durations of its instructions) and  $w(\mathcal{C})$  be its iteration weight, i.e., the sum of the iteration weights of its edges.



**Scheduling Algorithms.** Fig. 2 Reduced dependence graph without nonzero edges. This graph is necessarily acyclic, and a simple traversal provides a schedule

Then,  $\lambda \geq \frac{d(\mathcal{C})}{w(\mathcal{C})}$ . The intuition behind this bound is that the duration of the tasks of the cycle have to be spread on the number of iterations spanned by this cycle, and as cycles have all the same duration  $\lambda$ , one cannot do better than this ratio. Applying this bound to this example, it can be seen that  $\lambda \geq \frac{9}{3} = 3$  (cycle  $A \rightarrow B \rightarrow C \rightarrow A$ ) and  $\lambda \geq 3$  (self dependence of  $B$ ). Can the value  $\lambda = 3$  be reached?

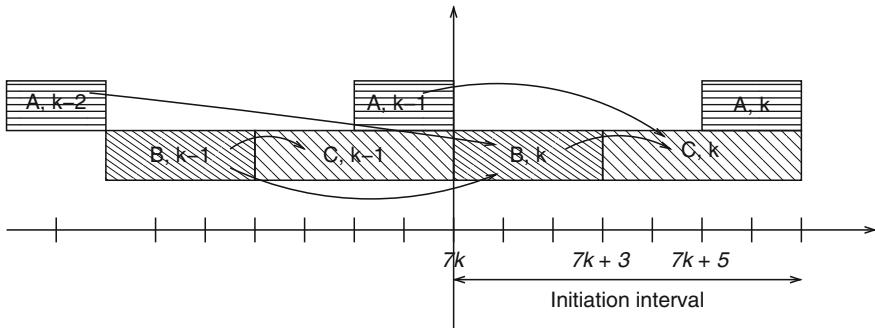
The answer depends again on the resource constraints. When an unlimited number of processors is available, one can show that the lower bound can be reached by solving a polynomial algorithm (essentially, Belman-Ford longest path algorithm). For this example, this would lead to the schedule  $\sigma(B, k) = 3k$ ,  $\sigma(C, k) = 3k+3$ , and  $\sigma(A, k) = 3k+4$ , which meets the constraints and allows the execution of the loop on 4 processors (see Fig. 4).

Solving the problem for limited resources is NP-complete. Another lower bound is related to the number of processors available. If  $p$  processors are available, then  $p\lambda \geq \sum_{v \in V} d(v)$ . Intuitively, the total execution time available during one iteration on all processors ( $p\lambda$ ) cannot be less than the total execution time needed to evaluate all tasks of an iteration.

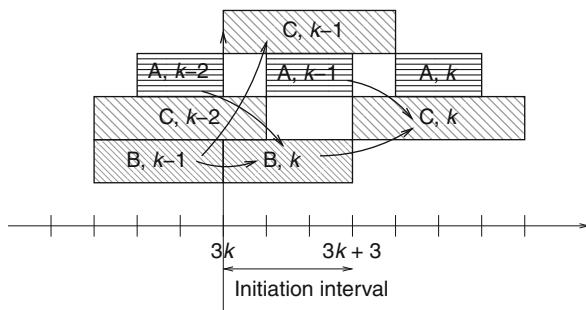
Several heuristics have been proposed in the literature to approach this bound: loop compaction, loop shifting, modulo scheduling, etc. All these methods rely upon list scheduling, as presented in the Sect. “Task graphs and Scheduling”. Notice, finally, that there exists *guaranteed heuristics* for solving this problem. One such heuristics combines loop compaction, loop shifting and retiming to obtain such a result.

### Scheduling Multidimensional Loops

Scheduling unidimensional loops is based on the idea of avoiding unrolling the loop and therefore, of keeping



**Scheduling Algorithms. Fig. 3** Execution of the loop with an initiation interval of 7. The corresponding schedule is  $\sigma(A, k) = 7k + 5$ ,  $\sigma(B, k) = 7k$ , and  $\sigma(C, k) = 7k + 3$



**Scheduling Algorithms. Fig. 4** Execution of the loop with an optimal initiation interval of 3. The corresponding schedule is  $\sigma(A, k) = 3k + 4$ ,  $\sigma(B, k) = 3k$ , and  $\sigma(C, k) = 3k + 3$

the model compact and tractable, independently on the loop bound  $N$ . To this end, the schedule is chosen to be a linear expression of the loop index. The same idea can be pushed further: instead of unrolling a multiple index loop, one tries to express the schedule of an operation by a linear (or affine) function of the loop indexes. This technique can be applied to imperative loops, but it interacts deeply with the detection of dependences in the loop, making the problem more difficult. Another way of looking at this problem is to consider algorithms expressed as recurrence equations, where parallelism is directly exposed.

Consider the following recurrences for the matrix multiplication algorithm  $C = AB$  where  $A$ ,  $B$ , and  $C$  are square matrices of size  $N$ :

$$\begin{aligned} 1 \leq i, j, k \leq N \rightarrow C(i, j, k) &= C(i, j, k-1) + a(i, k) \times b(k, j) \quad (1) \\ 1 \leq i, j \leq N \rightarrow c(i, j) &= C(i, j, N). \quad (2) \end{aligned}$$

Each one of these equations is a single assignment, and it expresses a recurrence, whose indexes belong to the set defined by the left-hand inequality. The first recurrence (1) defines how  $C$  is computed, where  $k$  represents the iteration index; the second recurrence (2) says that value  $c(i, j)$  is obtained as the result of the evaluation of  $C(i, j, k)$  for  $k = N$ .

An easy way to schedule these computations for parallelism is to find out a schedule  $t$  that is an affine function of all the indexes of these calculations. Assume that  $C$  has a schedule of the form  $t(i, j, k) = \lambda_1 i + \lambda_2 j + \lambda_3 k + \alpha$ ; also, assume that the evaluation of equations take at least one cycle, one must have, from Eq. 1:

$$t(i, j, k) > t(i, j, k-1), \forall i, j, k \text{ s.t. } 1 \leq i, j, k \leq N$$

or once simplified,  $\lambda_3 > 0$ . This defines a set of possible values for  $t$ , with simplest form  $t(i, j, k) = k$  (coefficient  $\alpha$  can be chosen arbitrarily equal to 0.) Once a schedule is found, it is possible to allocate the computations of a recurrence equation on a parallel architecture. A simple way of doing it is to use a linear allocation function: for the matrix multiplication example, one could take  $i$  and  $j$  as the number of the processors, which would result in a parallel matrix multiplication executed on a square grid-connected array of size  $N$ . But many other allocations are possible.

This simple example contains all the ingredients of loop parallelization techniques, as will be seen now.

First, notice that this kind of recurrence is particular: it is said to be *uniform*, as left-hand side operations depend uniformly on right-hand side expressions through a dependence of the form  $(0, 0, 1)$  (such a dependence is a simple translation of the left-hand side index). In general, one can consider *affine recurrences*

where any right-hand side index is an affine function of the left-hand side index.

Second, the set where the indexes must lie is an *integral convex polyhedron*, whose bounds may be parameterized by some symbolic values (here  $N$  for example).

This is not surprising: most of the loops that are of interest for parallelism share the property that array indexes are almost always affine, and that the index space of a loop can be modeled with a polyhedron.

Scheduling affine recurrences in general can be also done by extending the method. Assume that a recurrence has the form

$$\forall z \in P, V(z) = f(U(a(z)), \dots)$$

where  $U$  and  $V$  are variables of a program,  $z$  is a  $n$ -dimensional integer vector,  $P$  is a polyhedron, and  $a(z)$  is an affine mapping of  $z$ . The difference with the previous case is that the number of dependences may now be infinite: depending on the  $a$  function and the size of  $P$ , one may have a large, potentially infinite, number of dependences between instances of  $V(z)$  and  $U(a(z))$ , for which the schedule  $t$  must meet  $t(V(z)) > t(U(a(z)))$ . However, properties of polyhedra may be used. A given polyhedron  $P$  may be represented equivalently by a set of inequalities, or by its set of generators, that is to say, its vertices, rays, and lines. Generators are in finite number, and it turns out that it is necessary and sufficient to check the property of  $t$  on vertices and rays, and this leads to a finite set of linear inequalities involving the coefficients of the  $t$  function. This is the basis of the so-called *vertex method* for scheduling recurrences. Another, equivalent method, called the *Farkas method*, involves directly the inequalities instead of the generator system of the polyhedron. In general, such methods lead to high complexity algorithms, but in practice, they are tractable: a typical simple algorithm leads to a few hundreds of linear constraints.

Without going into the details, these methods can be extended to cover interesting problems. First of all, it is not necessary to assume that all variables of the recurrences have the same schedule. One may instead suppose that variable  $V$  is scheduled with a function  $t_V(i, j, \dots) = \lambda_{V,1}i + \lambda_{V,2}j + \dots + \alpha_V$  without making the problem more difficult. Secondly, one may also suppose that the schedule has several dimensions: if operation

$V(z)$  is scheduled at time  $\begin{pmatrix} t_{V,1}(z) \\ t_{V,2}(z) \end{pmatrix}$ , then the time is understood as a lexicographical ordering. This allows modeling situations where more than one dimension of the index space are evaluated sequentially on a single processor, thus allowing architectures of various dimensions to be found.

Very few researches have yet explored scheduling loops under constraints, as was the case for cyclic scheduling.

A final remark: cyclic scheduling is a particular case of multidimensional scheduling, where the function has the form  $\sigma(v, k) = \lambda k + \alpha_v$ . Indeed, the  $\alpha$  constant describes the relative time of execution of operation  $v(k)$  during an iteration. In a similar way for higher-level recurrences,  $\alpha_V$  plays the same rôle and allows one to refine the pipeline structure of one processor.

## From Loops to Recurrences

The previous section (Sect. Scheduling Multi-Dimensional Loops) has shown how a set of recurrence equations can be scheduled using linear programming techniques. Here it is sketched how loops expressed in a some imperative language can be translated into single-assignment form that is equivalent to recurrence equations. Consider the following example:

```
for i := 0 to 2n do
 {S1}: c[i] := 0. ;
for i := 0 to n do
 for j := 0 to n do
 {S2}: c[i+j] := c[i+j] +
 a[i]*b[j] ;
```

which computes the product of two polynomials of size  $n+1$ , where the coefficients of the input polynomials are in arrays  $a$  and  $b$  and the result in  $c$ . This program contains two loops, each one with one statement, labeled  $S1$  and  $S2$  respectively.

Each statement represents as many elementary *operations* as loop indexes surrounding it. Statement  $S1$  corresponds to  $2n+1$  operations, denoted by  $(S1, (i))$ , and statement  $S2$  corresponds to operations denoted  $(S2, (i, j))$ .

The left-hand side element of statement  $S2, c[i+j]$ , writes several times in the same memory cell and its right-hand side makes use of the content of the same

memory cell. In order to rewrite these loops as recurrence equations, one need to carry out a *data flow* analysis to find out which statement made the last modification to  $c[i+j]$ . Since both statement modify  $c$ , it may be either  $S2$  itself, or  $S1$ .

Consider first the case when  $S2$  is the source of the operation that modifies  $c[i+j]$  for operation  $(S2, (i, j))$ . Then this operation is  $(S2, (i', j'))$  where  $i' + j' = i + j$ , and indexes  $i'$  and  $j'$  must satisfy the limit constraints of the loops, that is,  $0 \leq i' \leq n$  and  $0 \leq j' \leq n$ . There is another condition, also called *sequential predicate*:  $(S2, (i', j'))$  must be the operation that was the last to modify cell  $c[i+j]$  before  $(S2, (i, j))$  uses it. But  $(i', j')$  is smaller than  $(i, j)$  in lexicographic order, and thus,  $(i', j')$  is the lexical maximum of  $(i, j)$ . All these conditions being affine inequalities, it turns out that finding out the source index can be done using *parameterized linear programming*, resulting in a so-called *quasi-affine selection tree* (quast) whose conditions on indices are also linear. One would thus find out that

$$(i', j') = \text{if } i > 1 \wedge j < n \text{ then } (i - 1, j + 1) \text{ else } \perp$$

where  $\perp$  represents the situation when  $(i', j')$  was defined by a statement that precedes the loops.

Now if  $S1$  is the source of the operation, this operation is  $(S1, i')$  where  $i' = i + j$ , and index  $i'$  must lay inside the bounds, thus  $0 \leq i' \leq 2n$ . Moreover, this operation must be executed before, which is obviously always true. Thus, it can be seen that  $(i') = (i + j)$ . Combining both expressions, one finds that the source of  $(i, j)$  is given by

$$\text{if } i > 1 \wedge j < n \text{ then } (S2, i - 1, j + 1) \text{ else } (S1, i + j).$$

Once the source of each operation is found, rewriting the program as a system of recurrence equations is easy. For each statement  $S$ , a variable  $S(i, j, \dots)$  with as many indexes as surrounding loops is introduced, and the right hand side equation is rewritten by replacing expressions by their source translation. An extra equation can be provided to express the value of output variables of the program, here,  $c$  for example.

$$0 \leq i \leq 2n \rightarrow S1(i) = 0. \quad (3)$$

$$0 \leq i \leq n \wedge 0 \leq j \leq n \rightarrow S2(i, j) = \text{if } i > 1 \wedge j < n \quad (4)$$

$$\text{then } S1(i - 1, j + 1) + a(i) * b(j) \quad (5)$$

$$\text{else } S1(i + j) + a(i) * b(j) \quad (5)$$

$$0 \leq i \leq 2n \rightarrow c(i) = \text{if } j = 0 \text{ then } S2(i, j) \text{ else } S2(n, j) \quad (6)$$

The exact data flow analysis is a means of expressing implicit dependences that result from the encoding of the algorithm as loops. Although other methods exist in order to schedule loops without single assignment, this method is deeply related to scheduling techniques.

## Scheduling Today and Future Directions

Loop scheduling is a central process of autoparallelization techniques, either implicit – when loops are rewritten after dependence analysis and then compiled, – or explicit, when the scheduling function is calculated and explicitly used during the parallelization process. Compilers incorporate more and more often sophisticated dependence and analysis tools to produce efficient code for target parallel architecture. Mono-dimensional loop scheduling is both used to massage loops for instruction-level parallelism, but also to design special-purpose hardware for embedded calculations. Finally, multi-dimensional scheduling is included in some autoparallelizers to produce code for parallel processors.

The need to produce efficient compilers is a limitation to the use of complex scheduling methods, as it has been shown that most practical problems are NP-complete. Progress in the knowledge of scheduling algorithms has made more and more likely for methods, previously rejected because of their complexity, to become common ingredients of parallelization tools. Moreover, the design of parallel programs tolerates a slightly longer production time than compiling everyday's program does, in particular because many applications of parallelism target embedded systems. Thus, the spreading of parallel computers – for example, GPU or multi-core – will certainly make these techniques useful.

## Bibliographic Notes and Further Reading

Darte, Robert, and Vivien [2] wrote a reference book on scheduling for parallelism; it provides an in-depth coverage of the techniques presented here, in particular task scheduling and cyclic scheduling. It also provides a survey and comparison of various methods for dependence analysis. This entry borrows a large part of its material from this book.

El-Rewini, Lewis, and Ali give in [3] a survey on scheduling in general and task scheduling in particular. In [1], Benoit and Robert give complexity results for scheduling problems that are related to recent parallel platforms.

Lam [9] presents *software pipelining*, one of the most famous applications of cyclic scheduling to loops. Theoretical results on cyclic scheduling can be found in [7].

Karp, Miller, and Winograd [8] were among the first authors to consider scheduling for parallel execution, in a seminal paper on recurrence equations. Lampert [10] published one of the most famous papers on loop parallelization. Loop dependence analysis and loop transformations for parallelization are extensively presented by Zima and Chapman in [12]. Maura, Quinton, Rajopadhye, and Saouter presented in [11] the vertex method to schedule affine recurrences, whereas Feautrier ([5] and [6]) describes the Farkas method for scheduling affine loop nests.

Data flow analysis is presented in great detail by Feautrier in [4].

## Bibliography

1. Benoit A, Robert Y (Oct 2008) Complexity results for throughput and latency optimization of replicated and data-parallel workflows. *Algorithmica* 57(4):689–724
2. Darte A, Robert Y, Vivien F (2000) Scheduling and automatic parallelization. Birkhäuser, Boston
3. El-Rewini H, Lewis TG, Ali HH (1994) Task scheduling in parallel and distributed systems. Prentice Hall, Englewood Cliffs, New Jersey
4. Feautrier P (February 1991) Dataflow analysis of array and scalar references. *Int J Parallel Program* 20(1):23–53
5. Feautrier P (Oct 1992) Some efficient solutions to the affine scheduling problem, Part I, one dimensional time. *Int J Parallel Program* 21(5):313–347
6. Feautrier P (December 1992) Some efficient solutions to the affine scheduling problem, Part II, multidimensional time. *Int J Parallel Program* 21(6):389–420
7. Hanen C, Munier A (1995) Cyclic Scheduling on parallel processors: An overview. In: Chrétienne P, Coffman EG Jr, Lenstra K, Lia Z (eds) Scheduling theory and its applications. John Wiley & Sons
8. Karp RM, Miller RE, Winograd S (July 1967) The organization of computations for uniform recurrence equations. *J Assoc Comput Machin* 14(3):563–590
9. Lam MS (1988) Software pipelining: an effective scheduling technique for VLIW Machines. In: SIGPLAN'88 conference on programming language, design and implementation, Atlanta, GA. ACM Press, pp 318–328
10. Lampert L (Feb 1974) The parallel execution of DO loops. *Commun ACM* 17(2):83–93
11. Maura C, Quinton P, Rajopadhye S, Saouter Y (September 1990) Scheduling affine parameterized recurrences by means of variable dependent timing functions. In: Kung SY, Schwartzlander EE, Fortes JAB, Przytula KW (eds) Application Specific Array Processors, Princeton University, IEEE Computer Society Press, pp 100–110
12. Zima H, Chapman B (1989) Supercompilers for Parallel and Vector Computers. ACM Press, New York

## SCI (Scalable Coherent Interface)

HERMANN HELLWAGNER

Klagenfurt University, Klagenfurt, Austria

### Synonyms

International standard ISO/IEC 13961:2000(E) and IEEE Std 1596, 1998 edition

### Definition

Scalable Coherent Interface (SCI) is the specification (standardized by ISO/IEC and the IEEE) of a high-speed, flexible, scalable, point-to-point-based interconnect technology that was implemented in various ways to couple multiple processing nodes. SCI supports both the message-passing and shared-memory communication models, the latter in either the cache-coherent or non-coherent variants. SCI can be deployed as a system area network for compute clusters, as a memory interconnect for large-scale, cache-coherent, distributed-shared-memory multiprocessors, or as an I/O subsystem interconnect.

### Discussion

#### Introduction

SCI originated in an effort of bus experts in the late 1980s to define a very high performance computer bus (“Superbus”) that would support a significant degree of multiprocessing, i.e., number of processors. It was soon realized that backplane bus technology would not be able to meet this requirement, despite advanced concepts like split transactions and sophisticated implementations of the latest bus standards and products. The committee thus abandoned the bus-oriented view

and developed novel, distributed solutions to overcome the shared-resource and signaling problems of buses, while retaining the overall goal of defining an interconnect that offers the convenient services well known from centralized buses [3, 6].

The resulting specification of SCI, approved initially in 1992 [8] and finally in the late 1990s [9], thus describes hardware and protocols that provide processors with the shared-memory view of buses. SCI also specifies related transactions to read, write, and lock memory locations without software protocol involvement, as well as to transmit messages and interrupts. Hardware protocols to keep processor caches coherent are defined as an implementation option. The SCI interconnect, the memory system, and the associated protocols are fully distributed and scalable: an SCI network is based on point-to-point links only, implements a distributed shared memory (DSM) in hardware, and avoids serialization in almost any respect.

### Goals of SCI's Development

Several ambitious goals guided the specification process of SCI [3, 5, 8].

**High performance.** The primary objective of SCI was to deliver high communication performance to parallel or distributed applications. This comprises:

- High sustained throughput;
- Low latency;
- Low CPU overhead for communication operations.

The performance goals set forth were in the range of Gbit/s link speeds and latencies in the low microseconds range in loosely-coupled systems, even less in tightly-coupled multiprocessors.

**Scalability.** SCI was devised to address scalability in many respects, among them [4]:

- Scalability of *performance* (aggregate bandwidth) as the number of nodes grows;
- Scalability of interconnect *distance*, from centimeters to tens of meters, depending on the media and physical layer implementation, but based on the same logical layer protocols;
- Scalability of the *memory system*, in particular of the *cache coherence protocols*, without a practical limit on the number of processors or memory modules that could be handled;

- *Technological scalability*, i.e., use of the same mechanisms in large-scale and small-scale as well as tightly-coupled and loosely-coupled systems, and the ability to readily make use of advances in technology, e.g., high-speed links;
- *Economic scalability*, i.e., use of the same mechanisms and components in low-end, high-volume and high-end, low-volume systems, opening the chance to leverage the economies of scale of mass production of SCI hardware;
- No short-term practical limits to the *addressing capability*, i.e., an addressing scheme for the DSM wide enough to support a large number of nodes and large node memories.

**Coherent memory system.** Caches are crucial for modern microprocessors to reduce average access time to data. This specifically holds for a DSM system with NUMA (non-uniform memory access) characteristics where remote accesses can be roughly an order of magnitude more expensive than local ones. To support a convenient programming model as known, e.g., from symmetric multiprocessors (SMPs), the caches should be kept coherent in hardware.

**Interface characteristics.** The SCI specification was intended to describe a standard interface to an interconnect to enable multiple devices from multiple vendors to be attached and to interoperate. In other words, SCI should serve as an “open distributed bus” connecting components like processors, memory modules, and intelligent I/O devices in a high-speed system.

### Main Concepts of SCI

In the following, the main concepts and features of SCI will be summarized and the achievements, as represented by several implementations of SCI networks, will be assessed.

**Point-to-point links.** An SCI interconnect is defined to be built only from unidirectional, point-to-point links between participating nodes. These links can be used for concurrent data transfers, in contrast to the one-at-a-time communication characteristics of buses. The number of the links grows as nodes are added to the system, increasing the aggregate bandwidth of the network. The links can be made fast and their performance can scale with improvements in the underlying technology. They can be implemented in a bit-parallel manner

(for small distances) or in a bit-serial fashion (for larger distances), with the same logical layer protocols. Most implementations use parallel links over distances of a few centimeters or meters.

**Sophisticated signaling technology.** The data transfer rates and lengths of shared buses are inherently limited due to signal propagation delays and signaling problems on the transmission lines. The unidirectional, point-to-point SCI links avoid such signaling problems. Since there is only a single transmitter and a single receiver rather than multiple devices, the signaling speed can be increased significantly. High speeds are also fostered by low-voltage differential signals.

Furthermore, SCI strictly avoids back-propagating signals, even reverse flow control on the links, in favor of high signaling speeds and scalability. Flow control information becomes part of the normal data stream in the reverse direction, leading to the requirement that an SCI node must at least have one outgoing link and one incoming link.

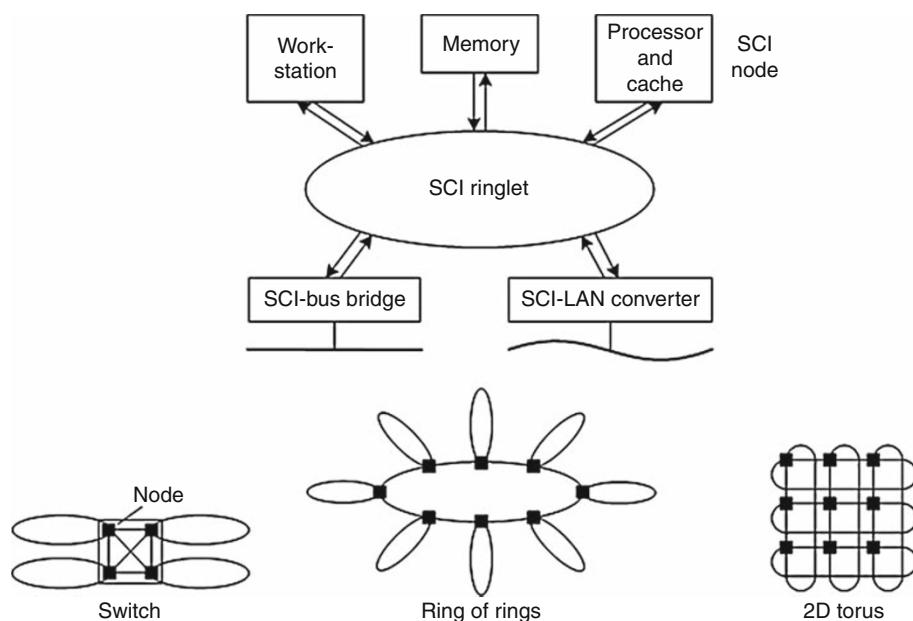
Already in the mid 1990s, SCI link implementation speeds reached 500 Mbyte/s in system area networks (distances of a few meters, 16-bit parallel links) and 1 Gbyte/s in closely-coupled, cache-coherent shared-memory multiprocessors; transfer rates of 1 Gbyte/s

have also been demonstrated over a distance of about 100 m, using parallel fiber-optic links; see [6].

**Nodes.** SCI was designed to connect up to 64 k nodes. A node can be a complete workstation or server machine, a processor and its associated cache only, a memory module, I/O controllers and devices, or bridges to other buses or interconnects, as illustrated exemplarily in Fig. 1. Each node is required to have a standard interface to attach to the SCI network, as described in [3, 5, 8]. In most SCI systems implemented so far, nodes are complete machines, often even multiprocessors.

**Topology independence.** In principle, SCI networks with complex topologies could be built; investigations into this area are described in several chapters of [6]. However, the standard anticipates simple topologies to be used. For small systems, for instance, the preferred topology is a small ring (a so-called *ringlet*); for larger systems, topologies like a single switch connecting multiple ringlets, rings of rings, or multidimensional tori are feasible; see Fig. 1. Most SCI systems implemented used single rings, a switch, multiple rings, or two-dimensional tori.

**Fixed addressing scheme.** SCI uses the 64-bit fixed addressing scheme defined by the Control and Status



SCI (Scalable Coherent Interface). Fig. 1 Simple SCI network topologies

Register (CSR) Architecture standard (IEEE Std 1212-1991) [7]. The 64-bit SCI address is divided into two fixed parts: the most significant 16 address bits specify the node ID (node address) so that an SCI network can comprise up to 64 k nodes; the remaining 48 bits are used for addressing within the nodes, in compliance with the CSR Architecture standard.

**Hardware-based distributed shared memory (DSM).** The SCI addressing scheme spans a global, 64-bit address space; in other words, a physically addressed, distributed shared memory system. The distribution of the memory is transparent to software and even to processors, i.e., the memory is logically shared. A memory access by a processor is mediated to the target memory module by the SCI hardware.

The major advantage of this feature is that inter-node communication can be effected by simple load and store operations by the processor, without invocation of a software protocol stack. The instructions accessing remote memory can be issued at user level; the operating system need not be involved in communication. This results in very low latencies for SCI communication, typically in the low microseconds range.

A major implementation challenge, however, is how to integrate the SCI network (and, thus, access to the system-wide SCI DSM) with the memory architecture of a standard single-processor workstation or a multiprocessor node. The common solutions, attaching SCI to the I/O bus or to the memory bus, will be outlined below.

**Bus-like services.** To complete the hardware DSM, SCI defines transactions to read, write, and lock memory locations, functionality well-known from computer buses. In addition, message passing and global time synchronization are supported, both as defined by the CSR Architecture; interrupts can be delivered remotely as well. Broadcast functionality is also defined.

Transactions can be tagged with four different priorities. In order to avoid starvation of low-priority nodes, fair protocols for bandwidth allocation and queue allocation have been developed. Bandwidth allocation is similar in effect to bus arbitration in that it assigns transfer bandwidth (if scarce) to nodes willing to send. Queue allocation apportions space in the input queues of heavily loaded, shared nodes, e.g., memory modules or switch ports, which are targeted by many nodes

simultaneously. Since the services have to be implemented in a fully distributed fashion, the underlying protocols are rather complex.

**Split transactions.** Like multiprocessor buses, SCI strictly splits transactions into *request* and *response* phases. This is a vital feature to avoid scalability impediments; it makes signaling speed independent of the distance a transaction has to travel and avoids monopolizing network links. Transactions therefore have to be self-contained and are sent as packets, containing a transaction ID, addresses, commands, status, and data as needed. A consequence is that multiple transactions can be outstanding per node. Transactions can thus be pumped into the network at a high rate, using the interconnect in a pipeline fashion.

**Optional cache coherence.** SCI defines distributed cache coherence protocols, based on a distributed-directory approach, a multiple readers – single writer sharing regime, and write invalidation. The memory coherence model is purposely left open to the implementer. The standard provides optimizations for common situations such as pair-wise sharing that improve performance of frequent coherence operations.

The cache coherence protocols are designed to be implemented in hardware; however, they are highly sophisticated and complex. The complexity stems from a large number of states of coherent memory and cache blocks, correspondingly complex state transitions, and the advanced algorithms that ensure atomic modifications of the distributed-directory information (e.g., insertions, deletions, invalidations). The greatest complication arises from the integration of the SCI coherence protocols with the snooping protocols typically employed on the nodes' memory buses. An implementation is highly challenging and incurs some risks and potentially high costs. Not surprisingly, only a few companies have done implementations, as described below.

The cache coherence protocols are provided as options only. A compliant SCI implementation need not cover coherence; an SCI network even cannot participate in coherence actions when it is attached to the I/O bus as is the case in compute clusters. Yet, a common misconception was that cache coherence was required functionality at the core of SCI. This

misunderstanding has clearly hindered SCI's proliferation for non-coherent uses, e.g., as a system area network.

**Reliability in hardware.** In order to enable high-speed transmission, error detection is done in hardware, based on a 16-bit CRC code which protects each SCI packet. Transactions and hardware protocols are provided that allow a sender to detect failure due to packet corruption, and allow a receiver to notify the sender of its inability to accept packets (due to a full input queue) or to ask the sender to re-send the packet. Since this happens on a per-packet basis, SCI does not automatically guarantee in-order delivery of packets. This may have a considerable impact on software which would rely on a guaranteed packet sequence. An example is a message passing library delivering data into a remote buffer using a series of remote write transactions and finally updating the tail pointer of the buffer. Some SCI hardware provides functionality to enforce a certain memory access order, e.g., via memory barriers [6].

Various time-outs are provided to detect lost packets or transmission errors. Hardware retry mechanisms or software recovery protocols may be implemented based on transmission-error detection and isolation mechanisms; these are however not part of the standard. As a consequence, SCI implementations differed widely in the way errors are dealt with.

The protocols are designed to be robust, i.e., they should, e.g., survive the failure of a node with outstanding transactions. Among other mechanisms, error containment and logging procedures, ringlet maintenance functions and a packet time-out scheme are specified. Robustness is particularly important for the cache coherence protocols which are designed to behave correctly even if a node fails amidst the modification of a distributed-directory entry.

**Layered specification.** The SCI specification is structured into three layers.

At the *physical layer*, three initial physical link models are defined: a parallel electrical link operating at 1 Gbyte/s over short distances (meters); a serial electrical link that operates at 1 Gbit/s over intermediate distances (tens of meters); and a serial optical link that operates at 1 Gbit/s over long distances (kilometers).

Although the definitions include the electrical, mechanical, and thermal characteristics of SCI modules, connectors, and cables, the specifications were not adhered to in implementations. SCI systems typically used vendor-specific physical layer implementations, incompatible to others. It is fair to say, therefore, that SCI has not become the open, distributed interconnect system that the designers had envisaged to create.

The *logical layer* specifies transaction types and protocols, packet types and formats, packet encodings, the standard node interface structure, bandwidth and queue allocation protocols, error processing, addressing and initialization issues, and SCI-specific CSRs.

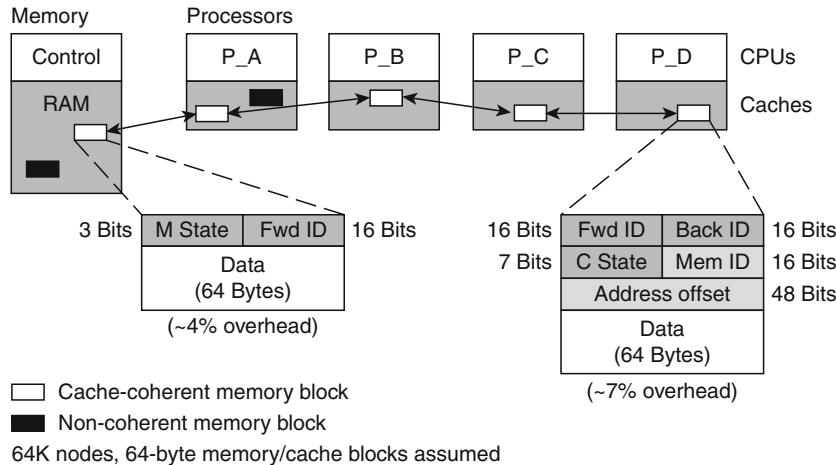
The *cache coherence layer* provides concepts and hardware protocols that allow processors to cache remote memory blocks while still maintaining coherence among multiple copies of the memory contents. Since an SCI network no longer has a central resource (i.e., a memory bus) that can be snooped by all attached processors to effect coherence actions, distributed-directory-based solutions to the cache coherence problem had to be devised.

At the core of the cache coherence protocols are the distributed sharing lists shown in Fig. 2. Each shared block of memory has an associated distributed, doubly-linked sharing list of processors that hold a copy of the block in their local caches. The memory controller and the participating processors cooperatively and concurrently create and update a block's sharing list, depending on the operations on the data.

**C code.** A remarkable feature of the SCI standard is that major portions are provided in terms of a "formal" specification, namely, C code. Text and figures are considered explanatory only, the definitive specification are the C listings. Exceptions are packet formats and the physical layer specifications. The major reasons for this approach are that C code is (largely) unambiguous and not easily misunderstood and that the specification becomes executable, as a simulation. In fact, much of the specification was validated by extensive simulations before release.

## Implementations and Applications of SCI

SCI was originally conceived as a shared-memory interconnect, but SCI's flexibility and performance potential



**SCI (Scalable Coherent Interface).** Fig. 2 Sharing list and coherence tags of SCI cache coherence protocols

also for other applications was soon realized and leveraged by industry. In the following, a few of these “classical” applications of SCI are introduced and examples of commercial systems that exploited SCI technology are provided.

### System Area Network for Clusters

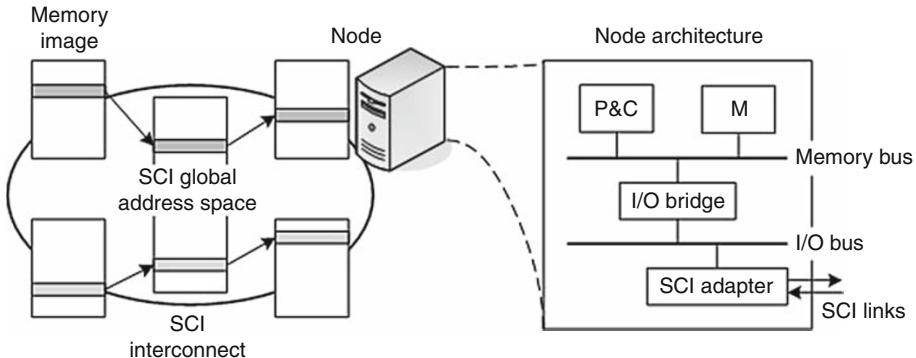
In compute clusters, an SCI system area network can provide high-performance communication capabilities. In this application, the SCI interconnect is attached to the I/O bus of the nodes (e.g., PCI) by a peripheral adapter card, very similar to a LAN; see Fig. 3. In contrast to “standard” LAN technology and most other system area networks, though, the SCI cluster network, by virtue of the common SCI address space and associated transactions, provides hardware-based, *physical* distributed shared memory. Figure 4 shows a high-level view of the DSM. An SCI cluster thus is more tightly coupled than a LAN-based cluster, exhibiting the characteristics of a NUMA parallel machine.

The SCI adapter cards, together with the SCI driver software, establish the DSM as depicted in Fig. 4. (This description pertains to the solutions taken by the pioneering vendor, Dolphin Interconnect Solutions, for their SBus-SCI and PCI-SCI adapter cards [2, 6].) A node that is willing to share memory with other nodes (e.g., A), creates shared memory segments in its physical memory and exports them to the SCI network (i.e., SCI address space). Other nodes (e.g., B) import these DSM segments into their I/O address space. Using on-board address translation tables (ATTs), the SCI

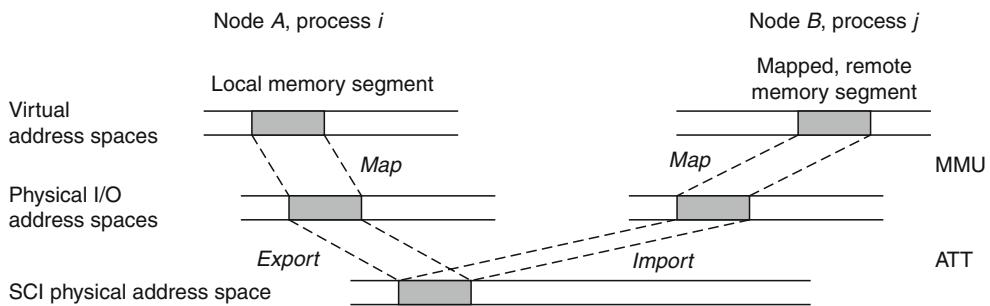
adapters maintain the mappings between their local I/O addresses and the global SCI addresses. Processes on the nodes (e.g.,  $i$  and  $j$ ) may further map DSM segments into their virtual address spaces. The latter mappings are conventionally being maintained by the processors’ MMUs.

Once the mappings have been set up, inter-node communication may be performed by the participating processes at *user level*, by simple CPU load and store operations into DSM segments mapped from remote memories. The SCI adapters translate I/O bus transactions that result from such memory accesses into SCI transactions, and vice versa, and perform them on behalf of the requesting processor. Thus, remote memory accesses are both transparent to the requesting processes and do not need intervention by the operating system. In other words, no protocol stack is involved in remote memory accesses, resulting in low communication latencies.

In the 1990s, the prevailing commercial implementations of such SCI cluster networks were the SBus-SCI and PCI-SCI adapter cards (and associated switches) offered by Dolphin Interconnect Solutions [2]. These cluster products were used by other companies to build key-turn cluster platforms; examples were Sun Microsystems, offering high-performance, high-availability server clusters, and Scalix Computers and Siemens providing clusters for parallel computing based on MPI; see [6]. Similar SCI adapters are still offered at the time of writing and are being used in embedded systems, e.g., medical devices such as CT scanners [2].



**SCI (Scalable Coherent Interface). Fig. 3** SCI cluster model



**SCI (Scalable Coherent Interface). Fig. 4** Address spaces and address translations in SCI clusters

In addition, there were research implementations of adapter cards, among them one developed at CERN as a prototype to investigate SCI's feasibility and performance for demanding data acquisition systems in high-energy physics, i.e., in one of the LHC experiments; see [6].

The description of an SCI cluster interconnect given above does not address many of the low-level problems and functionality that the implementation has to cover. For instance, issues like the translation of 32-bit node addresses to 64-bit SCI addresses and vice versa, the choice of the shared segment size, error detection and handling, high-performance data transfers between node hardware and SCI adapter, and the design and implementation of low-level software (SCI drivers) as well as message-passing software (e.g., MPI) represent a spectrum of research and development problems. Several contributions in [6] describe these problems and corresponding solutions in some detail.

An important property of such SCI cluster interconnect adapters is worth pointing out here. Since an SCI cluster adapter attaches to the I/O bus of a node,

it cannot directly observe, and participate in, the traffic on the memory bus of the node. This therefore precludes caching and coherence maintenance of memory regions mapped to the SCI address space. In other words, remote memory contents are basically treated as non-cacheable and are always accessed remotely. Standard SCI cluster interconnect hardware does not implement cache coherence capabilities therefore. This property raises a performance concern: remote accesses (round-trip operations such as reads) must be used judiciously since they are still an order of magnitude more expensive than local memory accesses (NUMA characteristics).

The basic approach to deal with the latter problem is to avoid remote operations that are inherently round-trip, i.e., reads, as far as possible. Rather, remote writes are used which are typically buffered by the SCI adapter and therefore, from the point of view of the processor issuing the write, experience latencies in the range of local accesses, several times faster than remote read operations. Again in [6], several chapters describe how considerations like this influence the design and

implementation of efficient message-passing libraries on top of SCI.

Finally, several contributions in [6] deal with how to overcome the limitations of non-coherent SCI cluster hardware, in particular for the implementation of shared-memory and shared-object programming models. Techniques from software DSM systems, e.g., replication and software coherence maintenance, are applied to provide a more convenient abstraction, e.g., a common virtual address space spanning all the nodes in the SCI cluster.

### Memory Interconnect for Cache-Coherent Multiprocessors

The use of SCI as a cache-coherent memory interconnect allows nodes to be even more tightly coupled than in a non-coherent cluster. This application requires SCI to be attached to the memory bus of a node, as shown in Fig. 5. At this attachment point, SCI can participate in and “export,” if necessary, the memory and cache coherence traffic on the bus and make the node’s memory visible and accessible to other nodes. The nodes’ memory address ranges (and the address mappings of processes) can be laid out to span a global (virtual) address space, giving processes transparent and coherent access to memory anywhere in the system. Typically, this approach is adopted to connect multiple bus-based commodity SMPs to form a large-scale, cache-coherent (CC) shared-memory system, often termed a CC-NUMA machine.

There were three notable examples of SCI-based CC-NUMA machines: the HP/Convex Exemplar series [1], the Sequent NUMA-Q multiprocessor [10], and the Data General AViiON scalable servers (see [6]). The

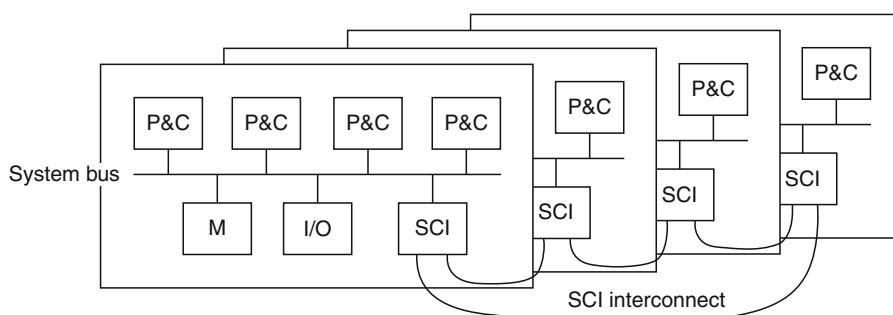
latter two systems comprised bus-based SMP nodes with Intel processors, while the Exemplar used HP PA-RISC processors and a non-blocking crossbar switch within the nodes. The inter-node memory interconnects were proprietary implementations of the SCI standard, with specific adaptations and optimizations incorporated to ease implementation and integration with the node architecture and to foster overall performance.

The major challenge in building a CC-NUMA machine is to bridge the cache coherence mechanisms on the intra-node interconnect (e.g., the SMP bus running a snooping protocol) and the inter-node network (SCI). Since it is well documented, the Sequent NUMA-Q machine is used as a case study in [6] to illustrate the essential issues in building such a bridge and making the protocols interact correctly.

### I/O Subsystem Interconnect

SCI can be used to connect one or more I/O subsystems to a computing system in novel ways. The shared SCI address space can include the I/O nodes which then are enabled to directly transfer data between the peripheral devices (in most cases, disks) and the compute nodes’ memories using DMA; software needs not be involved in the actual transfer. Remote peripheral devices in a cluster, for instance, thus can become accessible like local devices, resulting in an I/O model similar to SMPs; remote interrupt capability can also be provided via SCI. High bandwidth and low latency, in addition to the direct remote memory access capability, make SCI an interesting candidate for an I/O network.

There were two commercial implementations of SCI-based I/O system networks. One was the GigaRing channel from SGI/Cray [12], the other one the



SCI (Scalable Coherent Interface). Fig. 5 SCI based CC-NUMA Multiprocessor Model

external I/O subsystem interconnect of the Siemens RM600 Enterprise Servers, based on Dolphin's cluster technology; see again [6].

### Concluding Remarks

SCI addresses difficult interconnect problems and specifies innovative distributed structures and protocols for a scalable DSM architecture. The specification covers a wide spectrum of bus, network, and memory architecture problems, ranging from signaling considerations up to distributed directory-based cache coherence mechanisms.

In fact, this wide scope of SCI has raised criticism that the standard is actually “several standards in one” and difficult to understand and work with. This lack of a clear profile and the wide applicability of SCI have probably contributed to its relatively modest acceptance in industry. Furthermore, the SCI protocols are complex (albeit well devised) and not easily implemented in silicon. Thus, implementations are quite complex and therefore expensive.

In an attempt to reduce complexity and optimize speed, many of the implementers adopted the concepts and protocols which they regarded as appropriate for their application, and left out or changed other features according to their needs. This use of SCI led to a number of proprietary, incompatible implementations.

As a consequence, the goal of economic scalability has not been satisfactorily achieved in general. Further, SCI has clearly also missed the goal of evolving into an “open distributed bus” that multiple devices from different vendors could attach to and interoperate.

However, in terms of the technical objectives, predominantly high performance and scalability, SCI has well achieved its ambitious goals. The vendors that have adopted and implemented SCI (in various flavors), offered innovative high-throughput, low-latency interconnect products or full-scale cache-coherent shared-memory multiprocessing systems, as described above.

SCI had noticeable influence on further developments, e.g., interconnect standard projects like SyncLink, a high-speed memory interface, and Serial Express (SerialPlus), an extension to the SerialBus (IEEE 1394) interconnect. SCI also played a role in the debate on “future I/O systems” in the late 1990s, which then led to the InfiniBand™ Architecture. The

latest developments on SCI are explored in more detail in the final chapter of [6].

### Related Entries

- ▶ [Buses and Crossbars](#)
- ▶ [Cache Coherence](#)
- ▶ [Clusters](#)
- ▶ [Distributed-Memory Multiprocessor](#)
- ▶ [InfiniBand](#)
- ▶ [Myrinet](#)
- ▶ [NOW](#)
- ▶ [Nonuniform Memory Access \(NUMA\) Machines](#)
- ▶ [Quadrics](#)
- ▶ [Shared-Memory Multiprocessors](#)

### Bibliographic Notes and Further Reading

SCI was standardized and deployed in the 1990s. David B. Gustavson, one of the main designers, provides first-hand information on SCI's origins, development, and concepts [3, 4, 5]; even the standard document is stimulating material [8, 9]. Gustavson also maintained a Web-site that actively traced and documented the activities around SCI and related standards [11].

In the late 1990s, the interest in SCI largely disappeared. However, [6] was published as a comprehensive (and somehow concluding) summary of the technology, research and development projects, and achievements around SCI. For many projects and products, the corresponding chapters in [6] are the only documentation still available in the literature; in particular, information on products has disappeared from the Web meanwhile, but is still available to some extent in this book for reference.

This entry of the encyclopedia is a condensed version of the introductory chapter of [6].

### Bibliography

1. Brewer T, Astfalk G (1997) The evolution of the HP/Convex Exemplar. In: Proceedings of COMPCON, San Jose
2. Dolphin Interconnect Solutions (2009) <http://www.dolphinics.com>. Accessed October 2009
3. Gustavson DB, (1992) The Scalable Coherent Interface and related standards projects. IEEE Micro 12(1):10–22

4. Gustavson DB (1994) The many dimensions of scalability. In: Proceedings of the COMPCON Spring, San Francisco
5. Gustavson DB Li Q (1996) The Scalable Coherent Interface (SCI). IEEE Comm Mag 348:52–63
6. Hellwagner H, Reinefeld A (eds) (1999) SCI: Scalable Coherent Interface. Architecture and software for high-performance compute clusters. LNCS vol 1734, Springer, Berlin
7. IEEE Std 1212-1991 (1991) IEEE standard Control and Status Register (CSR) Architecture for microcomputer buses
8. IEEE Std 1596-1992 (1993) IEEE standard for Scalable Coherent Interface (SCI)
9. International Standard ISO/IEC 13961:2000(E) and IEEE Std 1596, 1998 Edition (2000) Information technology - Scalable Coherent Interface (SCI). ISO/IEC and IEEE
10. Lovett T, Clapp R (1996) STiNG: a CC-NUMA computer system for the commercial marketplace. In: Proceedings of the 23rd International Symposium on Computer Architecture (ISCA), May 1996, Philadelphia
11. SCIZZL: the local area memory port, local area multiprocessor, Scalable Coherent Interface and Serial Express users, developers, and manufacturers association. <http://www.scizzl.com>. Accessed October 2009
12. Scott S (1996) The GigaRing channel. IEEE Micro 16(1):27–34

## Semantic Independence

MARTIN FRÄNZLE<sup>1</sup>, CHRISTIAN LENGAUER<sup>2</sup>

<sup>1</sup>Carl von Ossietzky Universität, Oldenburg, Germany

<sup>2</sup>University of Passau, Passau, Germany

### Definition

Two program fragments are semantically independent if their computations are mutually irrelevant. A consequence for execution is that they can be run on separate processors in parallel, without any synchronization or data exchange. This entry describes a relational characterization of semantic independence for imperative programs.

### Discussion

#### Introduction

The most crucial analysis for the discovery of parallelism in a sequential, imperative program is the dependence analysis. This analysis tells which program operations must definitely be executed in the order prescribed by the source code (see the entries on dependences and dependence analysis). The conclusion is that all others can be executed in either order or in parallel.

The converse problem is that of discovering independence. Program fragments recognized as independent may be executed in either order or in parallel, all others must be executed in the order prescribed by the source code.

The search for dependences is at the basis of an automatic program parallelizer (see the entries on autoparallelization and on the polyhedron model). The determination of the independence of two program fragments is a software engineering technique that can help to map a specific application program to a specific parallel or distributed architecture or it can tell something specific about the information flow in the overall program.

While the search for dependences or independences has to be based on some form of analysis of the program text, the criteria underlying such an – in any Turing-complete programming language necessarily incomplete – analysis should be semantic. The reason is that the syntactic form of the source program will make certain (in)dependences easy to recognize and will occlude others. It is therefore much wiser to define independence in a model which does not entail such an arbitrary bias. An appropriate model for the discovery of independence between fragments of an imperative program is the relation between its input states and its output states.

#### Central Idea and Program Example

Program examples adhere to the following notational conventions:

- The value sets of variables are specified by means of a declaration of the form  $\text{var } x, y, \dots : \{0, \dots, n\}$ , for  $n \geq 1$  (that is, they are finite).
- If the value set is  $\{0, 1\}$ , Boolean operators are used with their usual meaning, where the number 0 is identified with false and 1 with true.
- Boolean operators bind more tightly than arithmetic ones.

The initial paper on semantic independence [2] contains a number of illustrative examples. The most complex one, Example 4, shall serve here as a guide:

```
var x, y, z : {0, 1};
α₁ : (x, y) := (x ∧ z, y ∧ ¬z)
α₂ : (x, y) := (x ∧ ¬z, y ∧ z)
```

The question is whether the effect of these two distinct multiple assignments can be achieved by two independent operations which could be executed mutually in parallel. At first glance, it does not seem possible, since  $x$  and  $y$  are subject to update and are both shared by both statements. However, this is due to the particular syntactic form of the program. One should not judge independence by looking at the program text but by looking at a semantic model.

One suitable model is a graph of program state transitions. The graph for the program above is depicted in Fig. 1. The nodes are program states. A program state consists of the three binary values of variables  $x$ ,  $y$ , and  $z$ . State transitions are modelled by arrows between states. Solid arrows depict transitions of statement  $\alpha_1$ , dashed arrows transitions of statement  $\alpha_2$ .

The question to be answered in order to determine independence on any platform is: is there a transformation of the program's state space which leads to variables that can be accessed independently? It is not sufficient to require updates to be independent but allow reading to be shared, because the goal is to let semantically independent program fragments access disjoint pieces of memory. This obviates a cache coherence protocol in a multiprocessor system and improves the

utilization of the cache space. Some special-purpose platforms, for example, systolic arrays, disallow shared reading altogether; see the entry on systolic arrays. Consequently, all accesses – writing and reading – must be independent.

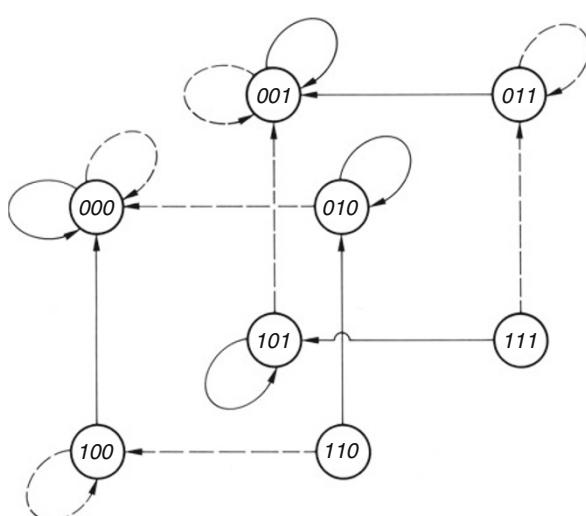
For the example, there is such a transformation of the variables  $x$ ,  $y$ , and  $z$  to new variables  $V$  and  $W$ :

```
var V : {0,1}, W : {0,1,2,3};
V = (x ∧ ¬z) ∨ (y ∧ z)
W = 2 * z + (x ∧ z) ∨ (y ∧ ¬z)
```

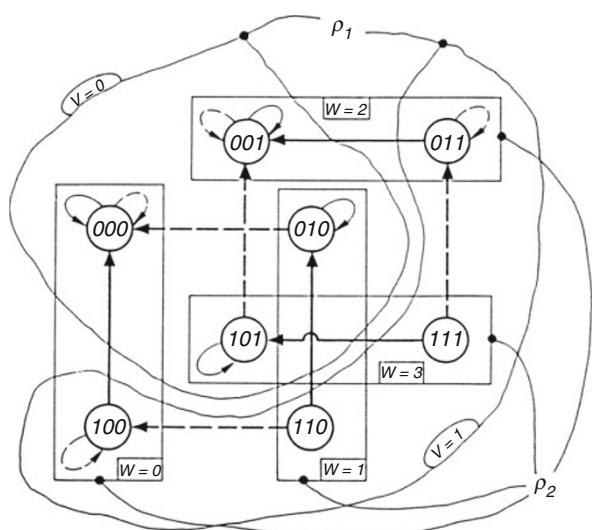
The transformation is depicted in Fig. 2. The figure shows the same state graph as Fig. 1 but imposes two partitionings on the state space. The set  $\rho_1$  of the two amorphously encased partitions yields the two-valued variable  $V$ . As denoted in the figure, each partition models one distinct value of the variable. The set  $\rho_2$  of the four rectangularly encased partitions yields the four-valued variable  $W$ . The partitionings must – and do – satisfy certain constraints to be made precise below.

The semantically equivalent program that results is:

```
var V : {0,1}, W : {0,1,2,3};
α'_1 : V := 0
α'_2 : W := 2 * (Wdiv 2)
```



**Semantic Independence. Fig. 1** State relations of the program example



**Semantic Independence. Fig. 2** State relations of the program example, partitioned

As is plainly evident, the new statements  $\alpha'_1$  and  $\alpha'_2$  access distinct variable spaces and are, thus, trivially independent. In Fig. 2, this property is reflected by the fact that  $\alpha_1$  (and, thus,  $\alpha'_1$ ) transitions are exclusively inside partitions of partitioning  $\rho_2$  and  $\alpha_2$  (and, thus,  $\alpha'_2$ ) transitions exclusively inside partitions of partitioning  $\rho_1$ .

The notion of semantic independence illustrated above and formalized below specifies a transformation of the state space that induces two partitionings, each covering one update in the transformed state space, such that both updates can be performed in parallel, without any need for shared access to program variables: they are completely independent in the sense that no information flows between them.

If the parallel updates depend on a common context, that is, they constitute only a part of the overall program, the transformations between the original and the transformed state space must become part of the implementation. Then, the transformation from the original to the transformed space may require multiple read accesses to variables in the original state space and the backtransformation may require multiple read accesses to variables in the transformed state space. Depending on the abilities of the hardware, the code for these transformations may or may not be parallelizable (see the section on disjoint parallelism and its limitations).

## Formal Definition

The first objective of this subsection is the independence of two imperative program fragments, as illustrated in the example. The generalization to more than two program fragments as well as to the interior of a single fragment follows subsequently.

### Independence Between Two Statements

A transformation of the state space can be modelled as a function

$$\eta : X \rightarrow Y$$

from a state space  $X$  to a state space  $Y$ . In this subsection, consider two program fragments,  $R$  and  $S$ , which are modelled as relations on an arbitrary (finite, countable, or uncountable) state space  $X$ :

$$R \subseteq X \times X \quad \text{and} \quad S \subseteq X \times X.$$

$\eta$  is supposed to expose binary independence. This leads to a partitioning of state space  $Y$ , that is,  $Y$  should be the Cartesian product of two nontrivial, that is, neither empty nor singleton state spaces  $A$  and  $B$ :

$$Y = A \times B.$$

Desired are two new program fragments  $R' \subseteq Y \times Y$  and  $S' \subseteq Y \times Y$ , such that  $R'$  operates solely on  $A$  and  $S'$  operates solely on  $B$ . Given two relations

$$r \subseteq A \times A \quad \text{and} \quad s \subseteq B \times B$$

denote their product relation as follows:

$$\begin{aligned} r \otimes s &= \{((a, b), (a', b')) \mid (a, a') \in r, (b, b') \in s\} \\ &\subseteq (A \times B) \times (A \times B) \end{aligned}$$

Also, write  $I_M$  for the identity relation on set  $M$ , that is, the relation  $\{(m, m) \mid m \in M\}$ . Then, the above requirement that  $R'$  and  $S'$  operate solely on  $A$  and  $B$ , respectively, amounts to requiring that

$$R' = (r \otimes I_B) \quad \text{and} \quad S' = (I_A \otimes s).$$

for appropriate relations  $r \subseteq A \times A$  and  $s \subseteq B \times B$ .

Furthermore, the composition of  $R'$  and  $S'$  should exhibit the same input–output behavior on the image  $\eta(X) \subseteq Y$  of the transformation  $\eta$  as the composition of  $R$  and  $S$  on  $X$ . Actually, this requirement has been strengthened to a mutual correspondence in order to avoid useless parallelism in which either  $R'$  or  $S'$  does the full job of both  $R$  and  $S$ :  $R'$  must behave on its part of  $\eta(X)$  like  $R$  on  $X$  and  $S'$  like  $S$ . With relational composition denoted as left-to-right juxtaposition, that is,

$$rs = \{(x, y) \mid \exists z : (x, z) \in r \wedge (z, y) \in s\},$$

the formal requirement is:

$$R \eta = \eta R' \quad \text{and} \quad S \eta = \eta S'.$$

In order to guarantee a correspondence between the resulting output state in  $Y$  and its equivalent in  $X$ , there is one more requirement:  $\eta$ 's relational inverse  $\eta^{-1} = \{(y, x) \mid y = \eta(x)\}$ , when applied to  $\eta(X)$ , must not incur a loss of information of a poststate generated by program fragments  $R$  or  $S$ . This is expressed by the two equations:

$$R = \eta R' \eta^{-1} \quad \text{and} \quad S = \eta S' \eta^{-1}$$

**Figure 3** depicts the previous two requirements as commuting diagrams, on the left for relation  $R$  and on the right for relation  $S$ . The dashed arrows represent the former and the dashed-dotted arrows the latter requirement. Together they imply:

$$R = R \eta \eta^{-1} \quad \text{and} \quad S = S \eta \eta^{-1}.$$

It is neither a consequence that  $\eta^{-1}$  is a function when confined to the images of  $R$  or  $S$ , nor that it is total on these images. However, if  $\eta \eta^{-1}$  contains two states  $(x, y)$  and  $(x, y')$ , then  $y$  and  $y'$  are  $R$ -equivalent (and  $S$ -equivalent) in the sense that  $(z, y) \in R$  iff  $(z, y') \in R$ , for each  $z \in X$  (and likewise for  $S$ ), that is,  $y$  and  $y'$  occur as poststates of exactly the same prestates. If  $R$  (or  $S$ ) is a total function, which is the case for deterministic programs, then  $\eta^{-1}$  restricted to the image of  $R$  (or  $S$ , respectively) must be a total function.

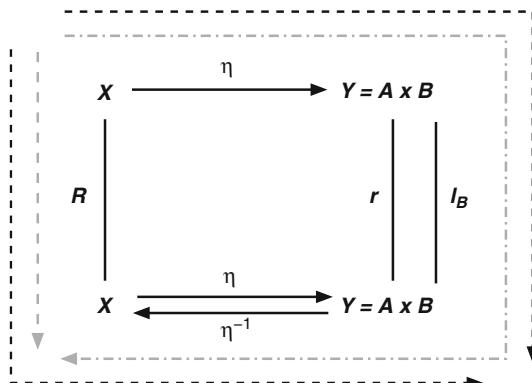
Putting all requirements together, one arrives at the following definition of semantic independence between two imperative program fragments.

**Definition 1** (Semantic independence [4]) *Two relations  $R$  and  $S$  on the state set  $X$  are called semantically independent if there are nontrivial (i.e., cardinality > 1) sets  $A$  and  $B$ , relations  $r$  on  $A$  and  $s$  on  $B$ , and a function  $\eta : X \rightarrow A \times B$  such that*

$$\begin{aligned} R\eta &= \eta(r \otimes I_B), & R\eta\eta^{-1} &= R, \quad \text{and} \\ S\eta &= \eta(I_A \otimes s), & S\eta\eta^{-1} &= S. \end{aligned} \quad (1)$$

Under the conditions of Eq. 1,  $R$  is simulated by  $r$  in the sense that

$$R = R\eta\eta^{-1} = \eta(r \otimes I_B)\eta^{-1}$$



and, likewise,  $S$  by  $s$  due to

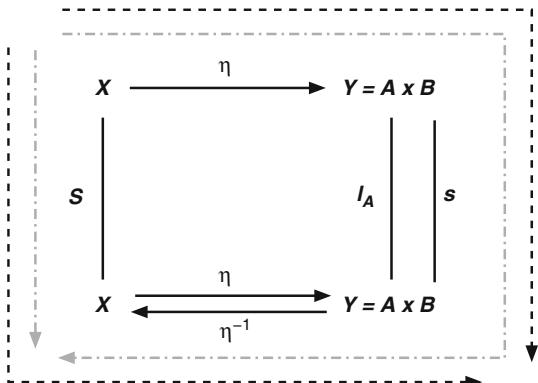
$$S = S\eta\eta^{-1} = \eta(I_A \otimes s)\eta^{-1}.$$

That is, after applying state transformation  $\eta$ , an effect equivalent to  $R$  can be obtained by executing  $r$ , with the result to be extracted by transforming the state space back via  $\eta^{-1}$ . The gain is that  $r$  only operates on the first component  $A$  of the transformed state space, not interfering with  $s$  which, in turn, simulates  $S$  on the second component  $B$ .

An immediate consequence of Definition 1 is that semantically independent relations commute with respect to relational composition, which models sequential composition in relational semantics. Take again  $R$  and  $S$ :

$$\begin{aligned} RS &= RS\eta\eta^{-1} \\ &= R\eta(I_A \otimes s)\eta^{-1} \\ &= \eta(r \otimes I_B)(I_A \otimes s)\eta^{-1} \\ &= \eta(I_A \otimes s)(r \otimes I_B)\eta^{-1} \\ &= S\eta(r \otimes I_B)\eta^{-1} \\ &= SR\eta\eta^{-1} \\ &= SR \end{aligned}$$

However, the converse is not true: semantic independence is more discriminative than commutativity. Intuitively, it has to be, since semantic independence



**Semantic Independence. Fig. 3** Two requirements on semantic independence

must ensure truly parallel execution, without the risk of interference, while commutativity may hold for interfering program fragments. A case of two program fragments which commute but are not semantically independent was provided as Example 5 in the initial paper [2]:

```
var x : {0,1,2,3};
 α_1 : $x := (x + 1) \text{ mod } 4$
 α_2 : $x := (x + 2) \text{ mod } 4$
```

To prove the dependence, consider the projection of a Cartesian product to its first component:

$$\begin{aligned}\downarrow_1 : A \times B &\rightarrow A \\ \downarrow_1((a, b)) &= a\end{aligned}$$

Since  $\alpha_1$  corresponds to a transition relation that is a total and surjective function on  $X = \{0, 1, 2, 3\}$ ,  $\eta \downarrow_1$  is necessarily a bijection between  $X$  and  $A$  whenever  $\eta$  satisfies Eq. 1. Consequently, relation  $S'$ , which implements  $\alpha_2$  in the transformed state space, cannot leave the first component  $A$  of the transformed state space  $A \times B$  unaffected, that is, it cannot be of the form  $(I_A \otimes s)$ , since  $\alpha_2$  is not the identity on  $X$ .

As pointed out before, the example in Fig. 1 is semantically independent in the sense of Definition 1. Taking  $A = \{0, 1\}$ , that is, as the domain of variable  $V$  from the transformation on page 2, and  $B = \{0, 1, 2, 3\}$ , that is, as the domain of variable  $W$ , and taking

$\eta(x, y, z) = ((x \wedge \neg z) \vee (y \wedge z), 2 * z + (x \wedge z) \vee (y \wedge \neg z))$ ,  
the two relations

$$s = \{(a, 0) \mid a \in A\} \quad \text{and} \quad r = \{(b, 2 * (b \text{ div } 2)) \mid b \in B\}$$

satisfy Eq. 1. These two relations  $s$  and  $r$  are the relational models of the assignments  $\alpha'_1$  and  $\alpha'_2$  on page 4.

## Mutual Independence Between More than Two Fragments

Semantic independence is easily extended to arbitrarily many program fragments. To avoid subscripts, consider only three program fragments given as relations  $R, S, T$  on the state set  $X$ , which illustrates the general case sufficiently. These relations are called semantically independent if they correspond to relations  $r, s, t$  on three nontrivial sets  $A, B, C$ , respectively, such that there is a function

$$\eta : X \rightarrow A \times B \times C$$

that can be used to represent  $R, S$  and  $T$  by the product relations

$$r \otimes I_B \otimes I_C, \quad I_A \otimes s \otimes I_C, \quad I_A \otimes I_B \otimes t$$

respectively. For  $R$ , this representation has the form

$$R \eta = \eta(r \otimes I_B \otimes I_C).$$

As before, the requirement is that  $R \eta \eta^{-1} = R$ . Relations  $S$  and  $T$  are represented analogously and are subject to corresponding requirements.

These conditions are considerably stronger than requiring the three pairwise independences of  $R$  with  $S T$ , of  $S$  with  $R T$ , and of  $T$  with  $R S$ . The individual checks of the latter may use up to three different transformations  $\eta_1$  to  $\eta_3$  in Eq. 1. This freedom of choice does not capture semantic independence adequately.

## Independence Inside a Single Statement

Up to this point, an underlying assumption has been that a limit of the granularity of parallelization is given by the imperative program fragments considered. At the finest possible level of granularity, these are individual assignment statements. If some such assignment statement contains an expression whose evaluation one would like to parallelize, one must break it up into several assignments to intermediate variables and demonstrate their independence.

One can go one step further and ask for the potential parallelism within a single statement. The question then is whether there exists a transformation that permits a slicing of the statement into various independent ones in order to obtain a parallel implementation.

An appropriate condition for semantic independence within a single statement given as a relation  $R \subseteq X \times X$  is that  $X$  can be embedded into a nontrivial Cartesian product  $A \times B$  via a mapping  $\eta : X \rightarrow A \times B$  such that

$$R \eta = \eta(r \otimes s), \quad R \eta \eta^{-1} = R \quad (2)$$

for appropriate relations  $r \subseteq A \times A$  and  $s \subseteq B \times B$ . The generalization to more than two slices is straightforward.

## Disjoint Parallelism and Its Limitations

One notable consequence of the presented notion of semantic independence is that, due to accessing disjoint pieces of memory, the concurrent execution of the transformed statements neither requires shared reading nor shared writing. These operations are eliminated

by the transformation or, if the context requires a state space representation in the original format  $X$  rather than the transform  $Y$ , localized in code preceding the fork of the concurrent statements or following their join. This is illustrated with our initial example, where the shared reading and writing of  $x$  and  $y$  as well as the shared reading of  $z$  present in the original statements

$$\begin{aligned}\alpha_1 : \quad & (x, y) := (x \wedge z, y \wedge \neg z) \\ \alpha_2 : \quad & (x, y) := (x \wedge \neg z, y \wedge z)\end{aligned}$$

disappear in the semantically equivalent form

$$\begin{aligned}\alpha'_1 : \quad & V := 0 \\ \alpha'_2 : \quad & W := 2 * (W \text{ div } 2)\end{aligned}$$

If the context requires the format of  $X$ , one will have to include the transformations  $\eta$  and  $\eta^{-1}$  in the implementation. For the example, this means that the code

$$\begin{aligned}\text{var } V : \{0,1\}, W : \{0,1,2,3\}; \\ \beta_1 : \quad V := (x \wedge \neg z) \vee (y \wedge z) \\ \beta_2 : \quad W := 2 * z + (x \wedge z) \vee (y \wedge \neg z)\end{aligned}$$

(or any alternative code issuing the same side effects on  $V$  and  $W$ ) has to be inserted before the concurrent statements  $\alpha'_1$  and  $\alpha'_2$  and that the code

$$\begin{aligned}\gamma_1 : \quad & x := \text{if } (W \text{ div } 2) = 1 \text{ then } (W \text{ mod } 2) \text{ else } V \\ \gamma_2 : \quad & y := \text{if } (W \text{ div } 2) = 0 \text{ then } (W \text{ mod } 2) \text{ else } V \\ \gamma_3 : \quad & z := W \text{ div } 2 \\ \text{endvar } V, W\end{aligned}$$

has to be appended behind their join. The keyword `endvar` indicates that, after the backtransformation, variables  $V$  and  $W$  are irrelevant to the remainder of the program. (The backtransformation  $\gamma_3$  of  $z$  from the transformed to the original state space could be omitted, since the original statements do not have a side effect on  $z$ , that is,  $\gamma_3$  turns out not to modify  $z$ ).

Whether implementations such as  $\beta_1$  and  $\beta_2$  of the projections  $\eta \downarrow_1$  and  $\eta \downarrow_2$ , which prepare the state spaces for the two parallel statements to be sparked, can themselves be executed in parallel depends on the ability to perform shared reading. Since these transformations have no side effects on the common state space  $X$  and only local ones on  $A$  or  $B$ , respectively, yet require shared reading on  $X$ , they can be executed in parallel if and only if the architecture supports shared reading on  $X$  – yet not necessarily on  $A$  or  $B$ , which need not

be accessible to the respective statement executed in parallel. Conversely, a parallel execution of the reverse transformation does, in general, require shared reads on  $A$  and  $B$ , yet no shared reads or writes of the individual variables spanning  $X$ , as illustrated by the statements  $\gamma_1$ ,  $\gamma_2$ , and  $\gamma_3$  above.

## Practical Relevance

In the parallelization of algorithms, the mapping  $\eta$  transforming the state space is often fixed, depending on the application, and, thus, left implicit. This leads to fixed parallelization schemes which can be applied uniformly, which makes the search for  $\eta$  and the calculation of the simulating relations  $r$  and  $s$  from the original state transformers  $R$  and  $S$  unnecessary. Instead, it suffices to justify the parallelization scheme once and for all. It is worth noting that, for many practical schemes, this can be done via the concept of semantic independence. Examples are the implementation of  $*$ -LISP on the Connection Machine [5] and Google's closely related MapReduce framework [3], which are based on a very simple transformation, or the use of residue number systems (RNS) in arithmetic hardware and signal processing [8], which exploits a much more elaborate transformation.

In the former two cases, transformation  $\eta$  maps a list of elements to the corresponding tuple of its elements, after which all elements are subjected to the same mapping, say  $f$ , in parallel. The implementation of  $\eta$  distributes the list elements to different processors. Then,  $f$  is applied independently in parallel to each list element and, finally, by the implementation of  $\eta^{-1}$ , the local results are collected in a new list.

In contrast, RNS, which has gained popularity in signal processing applications during the last two decades, transforms the state space rather intricately following the Chinese remainder theorem of modular arithmetic. In its simplest form, one considers assignments of the form  $x := y \odot z$ , where the variables are bounded-range, nonnegative integers and the operation  $\odot$  can be addition, subtraction, or multiplication. RNS transforms the variables in such assignments to vectors of remainders with co-primal bases as follows. (The numbers of a co-primal base have no common positive factors other than 1.)

Let  $x, y, z$  be variables with ranges  $\mathbb{N}_{\leq k}$ , where  $k$  is a nonzero natural number and  $\mathbb{N}_{\leq k}$  denotes the natural numbers up to  $k-1$ , and let  $f_1, \dots, f_n \ll k$  be co-primal natural numbers with  $\prod_{i=1}^n f_i \geq k$ . Then, each variable  $v$  of  $x, y, z$  is transformed to a vector  $(v_1, \dots, v_n)$  of variables such that  $v_i$  ranges over  $\mathbb{N}_{\leq f_i}$  and  $v_i = v \bmod f_i$ . Consequently,

$$\eta(x, y, z) = (x \bmod f_1, y \bmod f_1, z \bmod f_1, \dots, x \bmod f_n, y \bmod f_n, z \bmod f_n).$$

The independence of the slices of assignment  $x := y \odot z$ , one for each  $f_i$ , follows from the fact that

$$\begin{aligned} & (y \odot z) \bmod f_i \\ &= ((y \bmod f_i) \odot (z \bmod f_i)) \bmod f_i \\ &= (y_i \odot z_i) \bmod f_i \end{aligned}$$

for each of the three operations that can be substituted for  $\odot$ . This allows a parallel implementation in which the assignment  $x := y \odot z$  is replaced by  $n$  semantically independent assignments

$$x_i := (y_i \odot z_i) \bmod f_i \quad \text{for } i = 1, \dots, n.$$

These are executed in parallel without any need for information exchange. A significant speedup results from the reduced bit width of the arithmetic operations involved, which shrinks from  $\lceil \log_2 k \rceil$  to  $\lceil \log_2(\max_{i=1}^n f_i) \rceil$ . If chains of assignments are performed, the speedup can easily amortize the computational effort involved in the transformations  $\eta$  and  $\eta^{-1}$ . Then this parallelization scheme remains attractive even when the data are supplied and expected in standard binary encoding, as is normally the case in signal processing.

## Related Entries

- Connection Machine
- Dependences
- Dependence Analysis
- Parallelization, Automatic
- Polyhedron Model
- Systolic Arrays

## Bibliographic Notes and Further Reading

Semantic independence is a generalization of the syntactic independence criterion proposed by Arthur

J. Bernstein in 1966 [1]. This criterion states that independent program fragments are allowed to read shared variables at any time but must not update any shared variable.

Lengauer introduced a notion of independence based on Hoare logic in the early 1980s [6, 7]. It combines a semantic with a syntactic requirement. The semantic part of the independence relation is called full commutativity and permits an arbitrary interleaving of the program fragments at the statement level, the syntactic part is called non-interference and ensures the absence of shared updates within each statement.

The independence relation based on a program state graph, as proposed by Best and Lengauer in 1989 [2], is entirely semantic. Von Stengel [9] ported the model to universal algebra and Fränzle et al. [4] generalized it shortly thereafter to the notion presented here. Whether a general, automatic, and efficient method of finding  $\eta$  and constructing  $r$  and  $s$  in any practical programming language exists remains an open question.

Zhang [10] has ported the above notion of semantic independence to security. He adopts Definition 1 as the definition of a set of mutually secure transitions. Transformation  $\eta$  defines the local views of users and  $r$  and  $s$  are the locally visible effects of the globally secure transitions  $R$  and  $S$ .

The transformations of the state space, which make independence visible, reflect the same principle as the coordinate transformations of the iteration space in polyhedral loop parallelization. There are other correspondences between the two models, for example, the way in which one exposes parallelism in a single assignment statement. For a more detailed comparison, see the entry on the polyhedron model.

## Bibliography

1. Bernstein AJ (1966) Analysis of programs for parallel processing. *IEEE Trans Electr Comput EC-15(5):757–763*
2. Best E, Lengauer C (1989) Semantic independence. *Sci Comput Program* 13(1):23–50
3. Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. *Commun ACM*, 51(1):107–113, January 2008
4. Fränzle M, von Stengel B, Wittmüss A (1995) A generalized notion of semantic independence. *Info Process Lett* 53(1):5–9
5. Hillis WD (1986) The connection machine. MIT Press, Cambridge, MA

6. Lengauer C (1982) A methodology for programming with concurrency: the formalism. *Sci Comput Program* 2(1):19–52
7. Lengauer C, Hehner ECR (1982) A methodology for programming with concurrency: an informal presentation. *Sci Comput Program* 2(1):1–18
8. Omondi A, Premkumar B (2007) Residue number systems – theory and implementation, volume 2 of Advances in computer science and engineering. Imperial College Press, United Kingdom
9. von Stengel B (1991) An algebraic characterization of semantic independence. *Info Process Lett* 39(6):291–296
10. Zhang K (1997) A theory for system security. In: 10th IEEE computer security foundations workshop. IEEE Computer Society Press, June 1997, pp 148–155

## Semaphores

- Synchronization

## Sequential Consistency

- Memory Models

## Server Farm

- Clusters
- Distributed-Memory Multiprocessor

## Shared Interconnect

- Buses and Crossbars

## Shared Virtual Memory

- Software Distributed Shared Memory

## Shared-Medium Network

- Buses and Crossbars

## Shared-Memory Multiprocessors

LUIS H. CEZE

University of Washington, Seattle, WA, USA

### Synonyms

Multiprocessors

### Definition

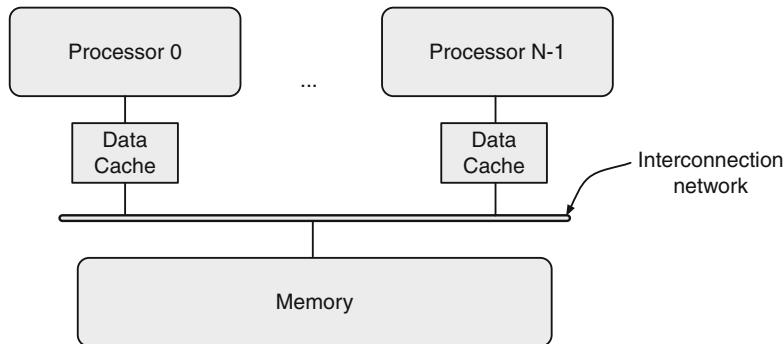
A shared-memory multiprocessor is a computer system composed of multiple independent processors that execute different instruction streams. Using Flynn's classification [1], an SMP is a multiple-instruction multiple-data (MIMD) architecture. The processors share a common memory address space and communicate with each other via memory. A typical shared-memory multiprocessor (Fig. 1) includes some number of processors with local caches, all interconnected with each other and with common memory via an interconnection (e.g., a bus).

Shared-memory multiprocessors can either be symmetric or asymmetric. Symmetric systems implies that all processors that compose the system are identical. Conversely, asymmetric systems have different types of processors sharing memory. Most multicore chips are single-chip symmetric shared-memory multiprocessors [2] (e.g., Intel Core 2 Duo).

### Discussion

Communication between processors in shared-memory multiprocessors happens implicitly via read and write operations to common memory address. For example, a data producer processor writes to memory location A, and a consumer processor reads from the same memory location:

| P1    | P2      |
|-------|---------|
| ----- | -----   |
| ...   | ...     |
| A = 5 | ...     |
| ...   | ...     |
| ...   | tmp = A |
| ...   | ...     |



**Shared-Memory Multiprocessors. Fig. 1** Conceptual overview of a typical shared-memory multiprocessor

Processor P1 wrote to location A and later processor P2 read from A and therefore read the value “5” into local variable “tmp,” written by P1. One way to think about the execution of a parallel program in a shared-memory multiprocessor is as some global interleaving of the memory operations of all threads. This global interleaving of instructions is nondeterministic, as there are no guarantees that if a given program is executed multiple times with the same input will lead to the same interleaving and therefore the same result.

Since communication happens nondeterministic, programs need to explicitly ensure that when communication happens, the data communicated is in a consistent state. This is done via synchronization, e.g., using mutual exclusion to prevent two threads from manipulating the same piece of data simultaneously.

One of the key components of a shared-memory multiprocessor is the cache coherence protocol. The cache coherence protocol ensures that all caches in each processor hold consistent data. The main job of the coherence protocol is to keep track of when cache lines are written to and propagating changes when that happens. Therefore, the granularity of data movement between processors is a cache line. There are many trade-offs in how cache coherence protocols are implemented. For example, whether updates should be propagated as they happen or if they should only be propagated when a remote processor tries to read the corresponding data. Another very important aspect of shared-memory multiprocessors is the memory consistency model, which determines when (what order) memory operations are made visible to processors in the system [3].

For the actual physical memory organization (and the interconnection network), it is common to have multiple partitions physically spread over the processors. This means that each partition is closer to some processors than others. Therefore, some regions of memory have faster access than others. Such organization is called non-uniform memory access, or NUMA. ccNUMA refers to NUMA multiprocessors with private caches per processor and cache coherence.

Given the general slower improvement in single-thread performance, processor manufacturers are now scaling raw compute performance in the forms of more cores (or processors) on a single chip, forming a symmetric shared-memory multiprocessor. Another recent trend points towards using specialized processors to further improve performance and energy efficiency, leading towards asymmetric shared-memory multiprocessor system.

S

## Related Entries

- ▶ Cache Coherence
- ▶ Cedar Multiprocessor
- ▶ Locality of Reference and Parallel Processing
- ▶ Memory Models
- ▶ Race Conditions
- ▶ Race Detection Techniques

## Bibliographic Notes and Further Reading

There are many notable shared-memory multiprocessor systems worth reading about, starting from mainframes

in the 60s (e.g., Burroughs B5500), many machines built in academia (e.g., UIUC CEDAR, Stanford Flash, Stanford DASH), and many systems manufactured and sold by Sequent and DEC (now defunct), as well as IBM, Sun, SGI, Fujitsu, HP, among several others.

## Bibliography

1. Flynn M (1972) Some computer organizations and their effectiveness. *IEEE Trans Comput* C-21:948
2. Olukotun et al (1996) The case for a single-chip multiprocessor. In: Proceedings of the 7th international symposium architectural support for programming languages and operating systems (ASPLOS VII), Cambridge, MA, October 1996
3. Adve S, Gharachorloo K (1996) Shared memory consistency models: a tutorial. *IEEE Comput Soc* 29(12):66–76

## SHMEM

► [OpenSHMEM - Toward a Unified RMA Model](#)

## SIGMA-1

KEI HIRAKI

The University of Tokyo, Tokyo, Japan

## Synonyms

[Dataflow supercomputer](#)

## Definition

SIGMA-1 is a large-scale computer based on fine-grain dataflow architecture designed to show feasibility of fine-grain dataflow computer to highly parallel computation over conventional von Neumann computers. SIGMA-1 project was started in 1984 and the 128 processing element (PE) started working in 1988. SIGMA-1 was designed and built at the Electrotechnical Laboratory (ETL for short), Ministry of International Trade and Industry, Japan. SIGMA-1 is still the largest scale dataflow computer so far built and it achieved more than 100 Mflops as a maximum measured performance of the total system. As for the language for SIGMA-1, ETL developed Dataflow-C language as a subset of C

programming language. Dataflow-C is a single assignment language that can compile C-like source programs to highly parallel executable dataflow machine codes (Fig. 1).

## Architecture

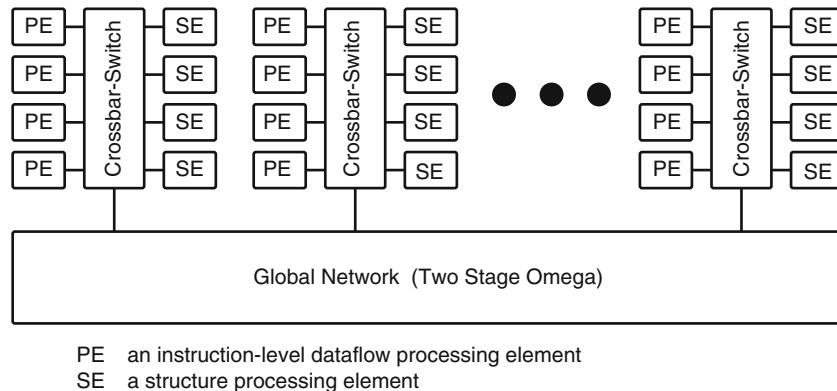
### Organization

The global organization of the SIGMA-1 is shown in Fig. 2. The SIGMA-1 consists of 128 processing elements, 128 structure elements, 32 local networks, a global network, 16 maintenance microprocessors, a service processor, and a host computer (Figs. 3 and 4). The processing elements and structure elements are divided into 32 groups, each of which consists of four processing elements, four structure elements, and a local network. All groups are connected via the global network. The global network consists of a two-stage omega network with a new adaptive load distribution mechanism. Its transfer rate is 2G bytes per second. A local network is a ten-by-ten crossbar packet switching network, eight ports of which are used for communication between processing elements and structure elements within a group and two to interface the global network. The transfer rate of a local network is 600M bytes per second. The whole system is synchronous and operates under a single clock (Fig. 5).

Figure 6 illustrates a processing element and a structure element. A processing element consists of an input buffer, a matching memory with a waiting matching function, an instruction memory, and a link register file. A processing element executes all the SIGMA-1 instructions except structure-handling instructions such as read, write, allocate, and deallocate used to access the structure memory in a structure element. A processing element works as a two-stage pipeline: the firing stage and the execution stage. In the firing stage, operand matching and instruction fetching are performed simultaneously. Input packets, stored in the input buffer, are sent to the matching memory. The matching memory enables the execution of instructions with two input operands. Successfully matched packets are sent to the execution stage with the instructions fetched from the instruction memory. If the match fails, the packet is stored in the matching memory and the instruction fetched is discarded. Chained hashing



SIGMA-1. Fig. 1 SIGMA-1 system

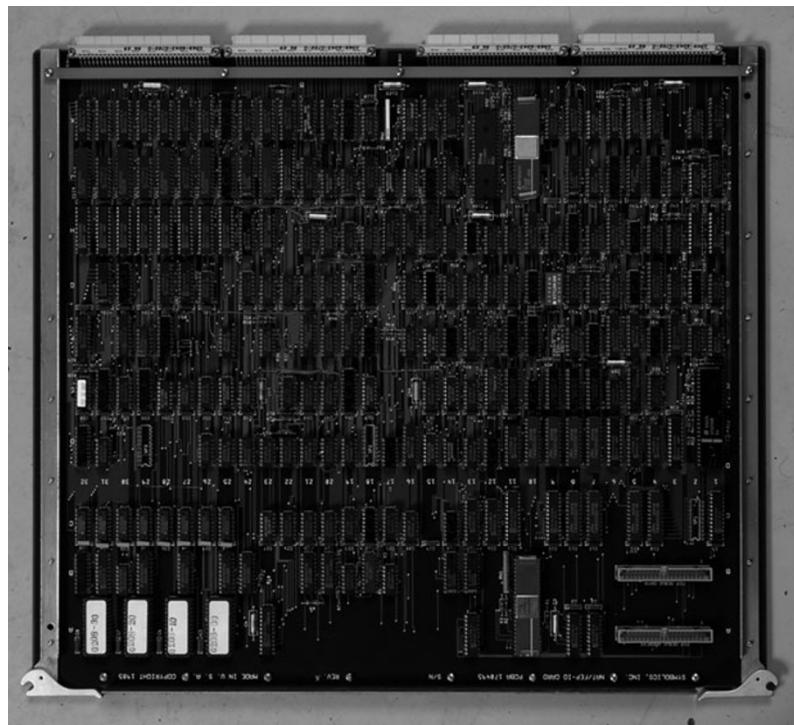


SIGMA-1. Fig. 2 Global architecture of the SIGMA-1

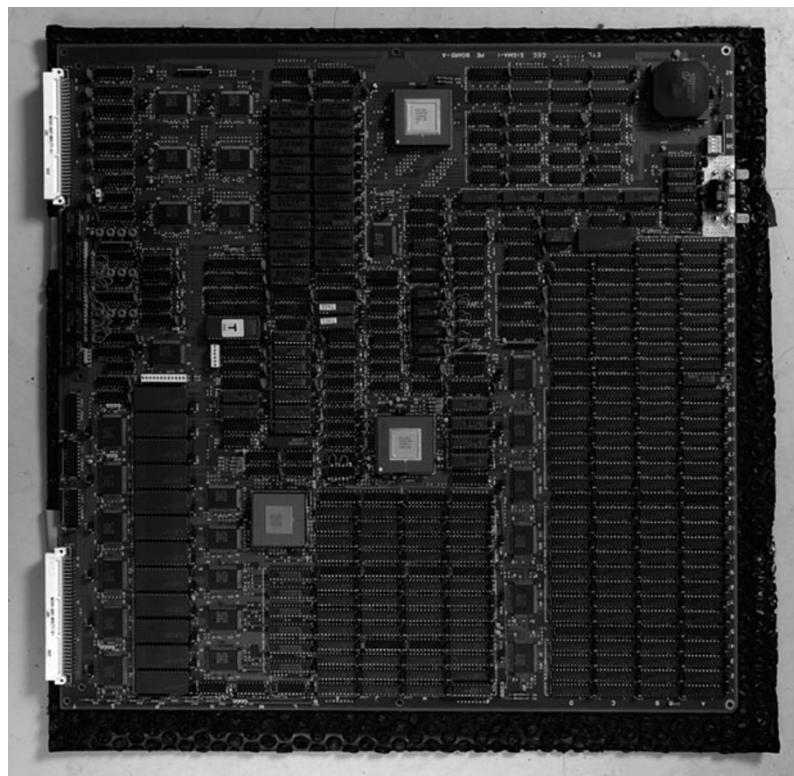
hardware is used to speed up the operand matching operation, where a first-in-first-out queue is attached to each key to enable multiple entries with the same key [Table 1](#).

The link register is used to identify a parallel activity in a program and hold a base address of a procedure. In the execution stage, instructions are executed in parallel with packet identifier generation, where a packet identifier keeps the destination address (next instruction address) as well as a parallel activity identifier (a procedure identifier and an iteration counter). These

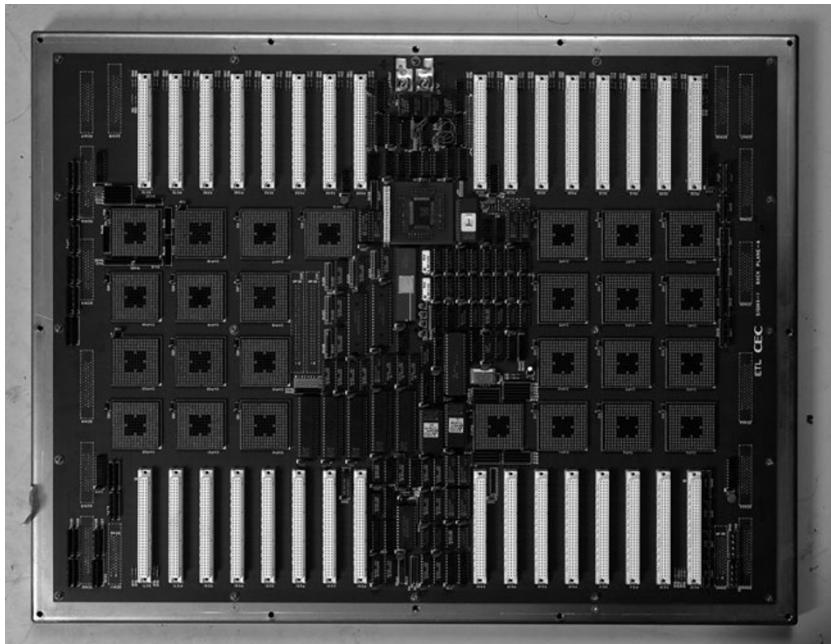
operations are similar to those in a conventional computer. After combining the result value of the execution unit with the packet identifier, an output packet is produced and sent to the input buffer of the same or another processing element via the local network. If an array is treated as a set of scalar values, the matching memory provides automatic synchronization and space management. However, heavy packet traffic causes a serious bottleneck at the matching memory, since each element of the array must be handled separately. This is the reason additional structure elements



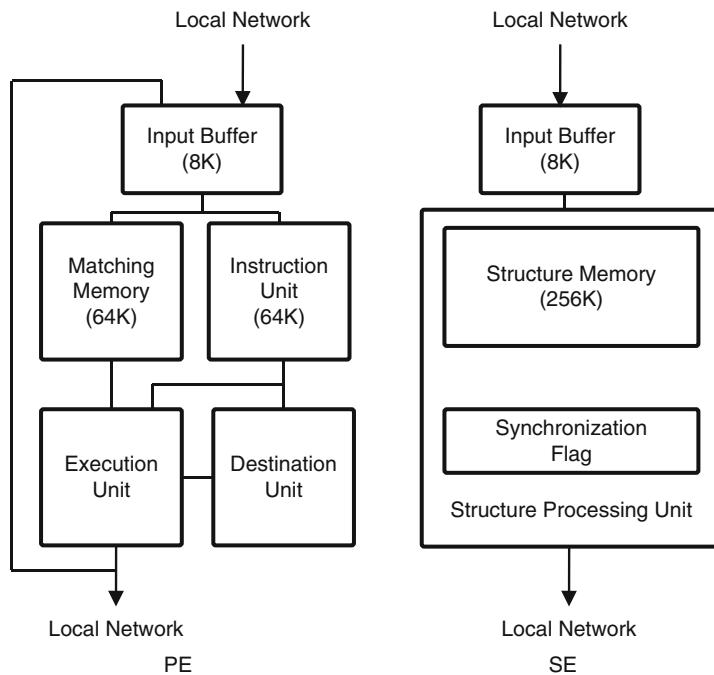
**SIGMA-1. Fig. 3** SIGMA-1 Processing element (optional)



**SIGMA-1. Fig. 4** SIGMA-1 Structure element (optional)



SIGMA-1. Fig. 5 SIGMA-1 Local network (optional)



SIGMA-1. Fig. 6 Processing element (PE) and SE of the SIGMA-1

are introduced in the SIGMA-1. A structure element consists of an input buffer, a structure memory, two flag logic functions, and a structure controller with a register file. The flag logic function is capable of test

and setting all flags in an area arbitrarily specified in a single clock. The structure controller is responsible for waiting queue control and memory management. A structure element handles arrays, which are the most

**SIGMA-1. Table 1** Hardware specification of the SIGMA-1

| Technology:            | CMOS Gate-array, SRAM, DRAM |
|------------------------|-----------------------------|
| Clock                  | 10 MHz                      |
| Input buffer           | 8 K words(80bits/word)      |
| Matching memory        | 64 K words(80bits/word)     |
| Instruction memory     | 64 K words(40bits/word)     |
| Structure memory       | 256 K words(40bits/word)    |
| Total amount of memory | 326 M Byte                  |
| Total no. of gates     | 19,013,447 gates            |
| Total no. of ICs       | 91,029                      |
| Total no. of PE/SE     | 128 + 128                   |

important data structure involved in numerical computations. Each structure cell is composed of a 40-bit data (data type and payload data) word and two flag bits used for read and write synchronization. Multiple read before write operations are realized by a waiting queue attached to each cell. The buddy system implemented in microprogramming is used to increase the speed of allocating and deallocating structures.

The architectural features of the SIGMA-1 are designed to achieve high speed by decreasing the parallel processing overheads in dataflow computers. For example, the short pipeline architecture enables quick response to small-sized programs with low parallelism. It was anticipated that the matching memory and the communication network would play a significant role in the system design from the viewpoint of synchronization and data transmission overhead. Besides the newly proposed mechanisms, high-speed and compact gate-array LSI elements have been developed for this purpose.

Testing, debugging, and maintenance are performed by a special purpose parallel architecture [6]. This maintenance architecture uses a set of conventional von Neumann microprocessors, since maintenance operations generally need history sensitivity utilizing side effects.

The architectural features of the SIGMA-1 are summarized as follows:

1. A short (two-stage) pipeline in a processing element, reducing the gap between the maximum and average performance
2. Chained hashing hardware for the matching memory, giving efficient and high-speed synchronization

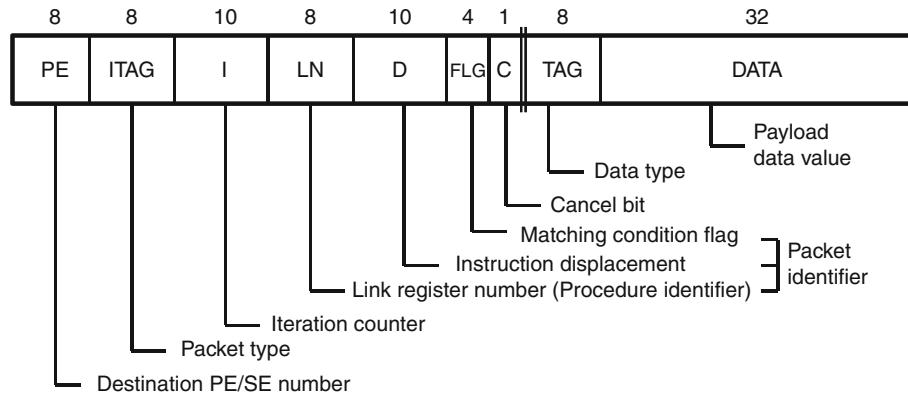
3. An array-oriented structure element, minimizing structure handling overhead
4. A hierarchical network with a dynamic load distribution function, reducing performance degradation caused by load imbalance
5. A maintenance architecture for testing and debugging, providing high-speed input and output operations and performing precise performance measurements

## Packet and Instruction Architecture

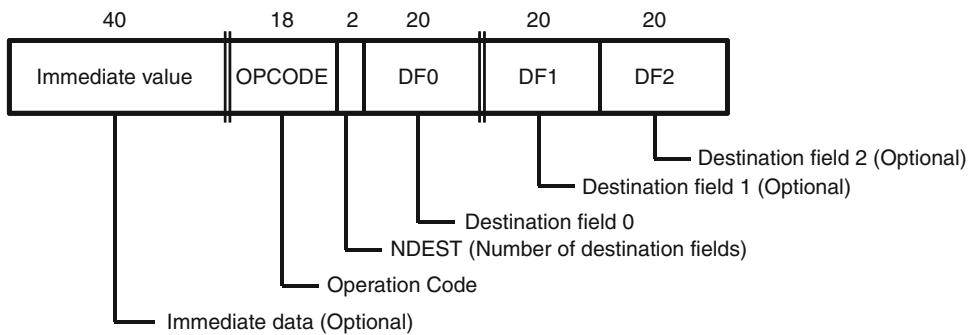
The SIGMA-1 adopts packet communication architecture. Processing elements and structure elements communicate using fixed length packets. Input and output to the host computer and maintenance system are also in packet form. Therefore, hardware initialization and hardware maintenance are carried out in packet form. As shown in Fig. 7, a packet contains a destination processing element number, a packet identifier and tagged data, and miscellaneous control information.

The 89-bit packet is divided into two 40-bit segments, a processing element number field (8 bits), and a cancel bit. A packet is transferred in two consecutive segments; the first segment contains a processing element or structure element number, a destination address in the processing element or structure element (32 bits), and packet type (8 bits). The second segment consists of data (32 bits), data type (8 bits), and one cancel bit. The cancel bit is used for canceling the preceding segment when a malfunction occurs. When the cancel bit is on, the whole packet becomes invalid.

A destination address consists of a procedure identifier (8 bits) for specifying a parallel activity, a relative instruction address (10 bits) within the activity, an iteration counter (10 bits), and some control information determining the firing rule in the matching memory (4 bits). The iteration counter is utilized to implement the loop construct efficiently. As in the ordinary model of dynamic dataflow, the concatenation of the procedure identifier and iteration counter is used to distinguish parallel activities in a program. The types of packets used are: result of instruction, procedure call and return, interrupt handling and system management, structure operation, and system initiation and maintenance for system resources.



SIGMA-1. Fig. 7 Packet format



SIGMA-1. Fig. 8 Instruction format

The minimum length of an instruction is 40 bits (one word) as illustrated in Fig. 8. The first 20 bits indicate the operation to be performed (18 bits) and the number of destination address fields (2 bits). The next 20 bits indicate the destination address of the result, which does not contain dynamically assigned information such as a procedure identifier and iteration counter. An immediate data operand can be located at the header part of each instruction. The maximum number of destination address fields is three, using two words. When an instruction contains an immediate data (constant) operand, another word is required.

The objective of the SIGMA-1 instruction set design is to execute efficiently application programs that cannot be efficiently executed on a vector-type supercomputer or a parallel von Neumann computer. Low efficiency in these program executions is caused by frequent procedure calls and returns, large amount of scalar arithmetic operations, and wide fluctuations of parallelism. Fine-grain parallelism has the advantage

over coarse-grain parallelism to overcome this inefficiency. As a result, 200 instructions on a processing element and 97 instructions on a structure element have been implemented (Fig. 9).

## Software

The DataFlow C (DFC) compiler is implemented using the language development tools of the UNIX operating system. The emphasis was not on efficiency but on ease of implementation. Efficiency and compactness are still to be considered. Basically, DFC is a subset of the C language with a single assignment rule. Therefore, assignment to a variable is generally allowed only once in each procedure. However, the loop construct, realized by the *for* statement, is an exception. A non-single assignment expression can be written as the third argument of the *for* statement as shown in the following summation program. The argument *n* of the main program is given by a trigger packet invoking the program.



**SIGMA-1. Fig. 9** SIGMA-1 PEs and SEs (optional)

```
#define N 10
main(n)
int n;
{int i; float a[N], retval, sigma();
for(i=0;i<5;i=I + 1) a[i] = i;
retval=sigma(n, a); print(retval);}

float sigma(n, a)
int n; float a[];
{int i; float s;
for (i=0, s=0; i<n; s=s+a[i], i=i+1);
return(s);}
```

DFC may have multiple return values as an extension to the C language. DFC programs without multiple return values can be compiled by a C compiler. The advantage is that DFC programs can be verified by a C compiler on a host computer. The DFC compiler comprises approximately 7,000 steps in C. A nested structure of *for* or *if* statements is not implemented and is inhibited. The DFC compiler generates a macro-assembly language program. This corresponds to a dataflow graph with no restriction on the number of arcs.

## System Performance

The whole system operates synchronously at a 10 MHz clock frequency. The execution time of an instruction is determined by the maximum execution time of the two pipeline stages. Single operand instructions are executed every two cycles and two operand instructions every three cycles. The firing stage normally takes three cycles. Since non-division arithmetic operations take at most two cycles, the execution stage completes most of the instruction execution in two cycles. This consideration implies that the maximum speed of a processing element is about 5 MIPS and 3 MFLOPS. For a structure element, each read and write instruction is carried out in two cycles. Since instructions for allocating and deallocating structures are implemented by a microprogrammed buddy system, the execution times estimated are between 10 and 340 cycles. The network transfers a packet every two cycles. Therefore, structure elements can utilize maximum performance of the network.

Performance is measured in terms of program execution time rather than raw machine cycles. The measurement study proved that there is no difference in computing ability of a single processing element between dataflow architecture and von Neumann architecture, in case of constructing them using the same device technology.

1. A speed degradation of 30% occurs when structure elements are adopted. This is due to the extra index and address calculation for structure handling.
2. The single processing element performance is almost constant over the vector length, because of the short pipeline architecture.
3. The speedup ratio for a single processing element to four processing element organization is 3.9 without structure elements and 3.5 with structure elements.

This performance evaluation shows that the average performance of the 128 processing element organization is more than 100 MFLOPS. For example, SIGMA-1 compute matrix multiplies at 117 Mflops.

## Discussion

Table 2 shows development history of major MIMD parallel computers and dataflow computers.

**SIGMA-1. Table 2** History of dataflow parallel processors

| Year <sup>a</sup> | System name                 | #PE           | Word length | Architecture                  |
|-------------------|-----------------------------|---------------|-------------|-------------------------------|
| 1976              | DDM1                        | 1 PE          | 4           | Static dataflow               |
| 1978              | TI DDP                      | 4 PE          | 16          | Static dataflow               |
| 1979              | LAU                         | 32 PE         | 32          | Static dataflow               |
| 1981              | Manchester dataflow machine | 1 PE          | 24          | Dynamic dataflow              |
| 1982              | OKI DDDP                    | 4 PE          | 16          | Dynamic dataflow              |
| 1982              | NTT Eddy                    | 16 PE         | 16          | Simulated dynamic dataflow    |
| 1984              | NEC NEDIPS                  | 8 PE          | 32          | Static dataflow               |
| 1983              | Q-p                         | 1 PE          | 16          | Static dataflow, Asynchronous |
| <b>1984</b>       | <b>SIGMA-1 Prototype</b>    | <b>1PE</b>    | <b>32</b>   | <b>Dynamic dataflow</b>       |
| <b>1988</b>       | <b>SIGMA-1</b>              | <b>128 PE</b> | <b>32</b>   | <b>Dynamic dataflow</b>       |
| 1988              | MIT monsoon prototype       | 1 PE          | 64          | Dynamic dataflow              |
| 1990              | ETL EM4                     | 80 PE         | 32          | Dynamic/Hybrid dataflow       |
| 1991              | MIT monsoon                 | 8 PE          | 64          | Dynamic dataflow              |
| 1996              | ETL EM-X                    | 80 PE         | 32          | Dynamic/Hybrid dataflow       |

<sup>a</sup>Year denote the date start working in its full configuration

As shown in Table 2, SI-1 is the largest dataflow computer so far built.

Illustrations, diagrams, and code examples.

## Bibliography

- Hiraki K, Nishida K, Sekiguchi S, Shimada T (1986) Maintenance architecture and its LSI implementation of a dataflow computer with a large number of processors. Proceedings of International Conference on Parallel Processing, IEEE Computer Society, University Park, pp 584–591
- Hiraki K, Sekiguchi S, Shimada T (1987) System architecture of a dataflow supercomputer. Proceedings of 1987 IEEE Region 10 Conference, IEEE computer Society, Los Alamitos, pp 1044–1049
- Hiraki K, Shimada T, Nishida K (1984) A hardware design of the SIGMA-1—A data flow computer for scientific computations. Proceedings of 1984 International Conference on Parallel Processing, IEEE Computer Society, Los Alamitos, pp 524–531
- Hiraki K, Shimada T, Sekiguchi S (1993) Empirical study of latency hiding on a fine-grain parallel processor. Proceedings of International Conference on Supercomputing, ACM, Tokyo, pp 220–229
- Sekiguchi S, Shimada T, Hiraki K (1991) Sequential description and parallel execution language DFCII dataflow supercomputers. Proceedings of International Conference on Supercomputing, ACM, New York, pp 57–66
- Shimada T, Hiraki K, Sekiguchi S, Nishida K (1986) Evaluation of a single processor of a prototype data flow computer SIGMA-1 for scientific computations. Proceedings of 12th Annual International Symposium on Computer Architecture, IEEE Computer Society, Chicago, pp 226–234

## SIMD (Single Instruction, Multiple Data) Machines

- Cray Vector Computers
- Floating Point Systems FPS-120B and Derivatives
- Flynn's Taxonomy
- Fujitsu Vector Computers
- Illiac IV
- MasPar
- MPP
- NEC SX Series Vector Computers
- Vector Extensions, Instruction-Set Architecture (ISA)

S

## SIMD Extensions

- Vector Extensions, Instruction-Set Architecture (ISA)

## SIMD ISA

- Vector Extensions, Instruction-Set Architecture (ISA)

## Single System Image

ROLF RIESEN<sup>1</sup>, ARTHUR B. MACCABE<sup>2</sup>

<sup>1</sup>IBM Research, Dublin, Ireland

<sup>2</sup>Oak Ridge National Laboratory, Oak Ridge, TN, USA

### Synonyms

Distributed process management

### Definition

A Single System Image (SSI) is an abstraction that provides the illusion that a multicomputer or cluster is a single machine. There are individual instances of the Operating Systems (OSs) running on each node of a multicomputer, processes working together are spread across multiple nodes, and files may reside on multiple disks. An SSI provides a unified view of this collection to users, programmers, and system administrators. This unification makes a system easier to use and more efficient to manage.

### Discussion

#### Introduction

Multicomputers consist of nodes, each with its own memory, CPUs, and a network interface. In the case of clusters, each node is a stand-alone computer made of commodity, off-the-shelf parts. Instead of viewing this collection of computers as individual systems, it is easier and more economical if users, programmers, and system administrators can treat the collection as a single machine. For example, users want to submit a single job to the system, even if it consists of multiple processes working in parallel.

The task of an SSI is to provide the illusion of a single machine under the control of the users and system administrators. The SSI is a layer of abstraction that provides functions and utilities to use, program, and manage the system. An SSI consists of multiple pieces. Some of them are implemented as individual utilities, other pieces are found in system libraries, there are additional processes running to hold up the illusion, OSs are modified, and in some systems hardware enables some of the abstraction. No SSI provides a complete illusion. There are always ways to circumvent the abstraction, and sometimes it is necessary to interact with specific

components of the system directly. Many SSI in daily use provide a limited abstraction that is enough to make use of the system practical, but do not add too much overhead or impede scalability.

An OS consists of a kernel, such as Linux, and a set of utilities and tools, such as a graphical user interface and tools to manipulate files. On a single computer, for example, a desktop workstation, the OS manages resources, such as CPUs, memory, disks, and other peripherals. It does that by providing a set of abstractions such as processes and files that allow users to interact with the system in a consistent manner independent of the specific hardware used in that system. For everyday tasks, it should not matter what particular CPU is installed, or whether files reside on a local or a network-attached disk drive. The goal of an SSI is to provide a similar experience to users of a parallel system.

That is not that difficult to achieve in a Symmetric Multi-Processor (SMP) where all memory is shared and all CPUs have, more or less, equally fast access to it. The OS for an SMP has additional features to manage the multiple CPUs, but the view it presents to a user is mostly the same that a user of a single-CPU system sees. For example, when launching an application, a user of an SMP does not need to specify which CPU should run it. The OS will select the least busy CPU and schedules the process to run there. There are additional functions that an SMP OS provides, for example, to synchronize processes, pin them to specific CPUs, and allow them to share memory. Nevertheless, the abstraction is still that of a single, whole machine.

In contrast, in a distributed memory system, such as a large-scale supercomputer or a cluster, each node runs its own copy of the OS. It is possible that not all nodes have the same capabilities; some may have access to a Graphic Processing Unit (GPU), the CPUs and memory sizes may be different, and some nodes may have connections to wide-area networks that are not available to all nodes in the system. The task of an SSI is to unify these individual components and present them to the user as if they were a single computer. This is difficult and sometimes not desired. Accessing data in the memory or disk of a remote node is much more time consuming than accessing data locally. It therefore makes sense for a user to specify that a process be run on the node where the data currently resides. It also makes

sense to give users the ability to specify that a process be run on a node that meets minimum memory requirements or has a certain type of CPU or GPU installed. On the other hand, for many tasks, or in a homogeneous system, the SSI should hide the complexity of the underlying system and present a view that lets users interact with it as if it were a large SMP.

An SSI is meant to make interaction with a parallel system easier and more efficient for three different groups of users: Application users who use the system to run applications such as a database or a simulation, programmers who create applications for parallel systems, and administrators who manage these systems. Each group has its own requirements for, and uses of, an SSI; and the view an SSI presents to each group is different. Of course, there is some overlap since there are tasks that need to be done by members of more than one group. The following sections discuss an SSI from these three different viewpoints. Functions that overlap are discussed in earlier viewpoints.

### Application User's View

A user of a parallel system uses it for an application such as computing the path of an asteroid, the weekly payroll, or maintaining the checking accounts of a bank. This type of user does this by running an application which provides the necessary functionality. In some cases, like the banking example, the user never directly interacts with the system. Those users only see the application interface, for example, the banking application that presents information about customers and their account balances. Parallel systems used in science and engineering cater to users who each have their own applications to solve their particular scientific problems. These users do interact with the system and an SSI assists them in making these interactions smooth and reliable.

Large systems that are shared among many users usually dedicate several nodes as login nodes. Smaller clusters sometimes let users log into any of the nodes, though this is less efficient for running parallel applications. An SSI gathers these login nodes under the umbrella of a single address. Users remotely log into that address and the SSI chooses the least busy node as the actual login node.

Once logged in, users need the ability to launch serial or parallel jobs (applications) and control them.

An SSI chooses the nodes to run the individual processes on and provides utilities to the user to obtain status information, stop, suspend, and resume jobs.

While an application is running, users need to interact with it. The SSI transparently directs user input to the application, independent of which node it is running on. Similarly, output from the application is funneled (if it comes from multiple processes) back to the user. This is the same as on any computing system. However, in a distributed system, the SSI needs to provide that functionality even if a process migrates to another node.

Input and output data for an application is often stored in files. Users must have the ability to see these files and manipulate them, for example, copy, move, and delete them. This requires a shared file system, since the application must be able to access the same files. Since the user, the application, and the files themselves may all reside on different nodes, a distributed file system is needed that presents a single root to all nodes in the system. That means the full path name of a file, that is, the hierarchy of directories (folders) above it and the file name itself, must be the same when viewed from any node.

Peripherals should be visible as if they were connected to the local node. The SSI must provide a view of that peripheral and transfer data to and from it, if the device is not local.

### Programmer's View

Programmers of parallel applications have many of the same requirements as application users. They also need to launch applications, to test them for example, and need access to the file system. The same is true for writing and compiling applications.

For debugging, the SSI may need to route requests from the debugger to processes on another node to suspend them and to inspect their memory and registers. If this functionality is embedded in the debugger itself, then this is no longer a requirement for the SSI, but it does require services that are part of the OS.

Programmers do have additional requirements that an SSI provides. The programs they write need to access low-level system services provided by the OS. That is done through system calls that allow applications to trap into an OS kernel to obtain a specific service. Typical services provided are calls to open a file for reading and

writing, creating and starting a new process, and interact through Inter-Process Communication (IPC) with another process. In a distributed system, these functions become more complex.

A process creation request in a desktop system is an operation carried out by the local OS. In a distributed system managed by an SSI, process creation may mean interaction with the OS of another node. If the original process and the new process need to communicate, then data has to move between nodes instead of just being copied locally into another memory location. For an application that is not aware of the distributed nature of the system it is running on, the SSI must provide this functionality so that it is transparent to the application.

Many applications consist of multiple processes that work together by sharing memory locations. On a single system, that is an efficient way to move data from one process to another and lets multiple processes read the same data. If these processes, in a distributed system, do not have access to the same memory anymore, this becomes more difficult and expensive, than it was before. Some SSI provide the illusion of a shared memory space. This is enabled by specialized hardware or through Distributed Shared Memory (DSM) implemented in software.

### **Administrator's View**

An administrator of a parallel system is responsible for managing user accounts, keeping system software up to date, replacing failed components, and in general have the system available for its intended use. Many of the administrators tasks are similar to the ones a user performs, for example, manipulating files, but with a higher privilege level.

In addition, an SSI should provide a single administrative domain. That means once the system administrator has logged in and has been authenticated by the system, the administrator should be able to perform all tasks for the entire system from that node. The SSI enables killing, suspending, and resuming any process, independent of its location. A system administrator further has the ability to trigger the migration of a process. This is sometimes necessary to vacate a portion of the system so it can be taken down for maintenance or repair. To do this, it must be possible to "down" network links, disks, CPUs, and whole nodes. Once down, they

wont be allocated and used again until they have been fixed or are done with maintenance.

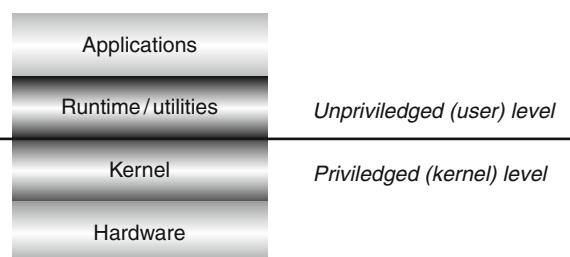
System administrators also need the ability to monitor nodes and links to troubleshoot problems or look for early signs of failures. An SSI can help with these tasks by providing utilities to search, filter, and rotate logs as an aggregate so an administrator does not need to login or access each node individually.

Often, each node has its own copy of system files, libraries, and the OS installed on a local disk. This increases performance and enhances scalability. An SSI provides functions to keep the versions of all these files synchronized and allows a system administrator to update the system software on all nodes with a single command.

### **Implementation of an SSI**

The previous section looked at the functionality an SSI provides to users, programmers, and system administrators. This section provides a little bit more detail on how these abstractions are implemented and at what level of the system hierarchy. [Figure 1](#) shows the layers of a system that contain SSI functions. User level is a CPU mode of operation where hardware prevents certain resource accesses. Applications and utilities run at this level. At the kernel level, all (local) resources are fully accessible. The OS kernel runs at this privilege level and manages resources on behalf of the applications running above it.

At the bottom of the hierarchy is the hardware. Some SSI features can only be implemented at that level. If the necessary hardware is missing, that feature will not be available or only very inefficiently in the form of a software layer. Shared memory on a system without hardware support for it is such an example.



**Single System Image. Fig. 1** Privilege and abstraction layers in a typical system

One level above the hardware resides the OS kernel. Some SSI functionality, for example, process migration, needs to be implemented at that level. Above the OS is the runtime layer and system utilities. This layer is sometimes called middle-ware. A batch and job control system which determines when and where (on which nodes) a job should run can be found at this level. Of course, utilities at that layer need the support of the OS and ultimately the underlying hardware to do their job.

Above that are the applications that make use of the system. Some SSI functionality, for example, application-level checkpoint and restart can be implemented at that level, for example, in a library. **Table 1** shows various SSI features and at what level they are implemented. Some can be implemented at more than one level. Only the levels that have been traditionally used, and where a feature is most efficient, are marked.

## Shared Memory

Sharing memory that is distributed among several nodes requires hardware support to work well. Implementations in software of so-called Distributed Shared Memory (DSM) systems have been done, but they all suffer from poor performance and scalability.

**Single System Image. Table 1** Features of an SSI and at what layer of the system hierarchy they are most commonly implemented

| Feature                | Implementation layer |        |         |      |
|------------------------|----------------------|--------|---------|------|
|                        | HW                   | Kernel | Runtime | App. |
| Shared memory          | •                    |        |         |      |
| Up/down sys components | •                    | •      | •       |      |
| Debugger support       |                      | •      | •       |      |
| Process migration      |                      | •      |         |      |
| Stdin/stdout           |                      | •      |         |      |
| File system            |                      | •      |         |      |
| System calls           |                      | •      |         |      |
| Checkpoint/restart     |                      | •      | •       | •    |
| Batch system           |                      |        | •       |      |
| Login load leveling    |                      |        | •       |      |
| Sys log management     |                      |        | •       |      |
| SW maintenance         |                      |        | •       |      |

## Management of System Components

In larger systems, it is desirable to mark components as being down so they won't be used anymore. Marking a node down, for example, should prevent the job scheduling system from allocating that node. In some systems, it is possible to mark links as being down and have the system route messages around that link. This is useful for system maintenance. Components that need repair won't be used until the system is brought down during the regular maintenance period, when all failed components can be repaired at the same time. Depending on the sophistication of this management feature, it requires hardware and kernel support. Simply telling the node allocator to avoid certain nodes can be done at user level in the runtime system.

## Debugger Support

A debugger is a user-level program like any other application. However, in order to signal and control processes, and to inspect their memory and registers, the debugger needs support from the OS. Some SSI modify kernels such that standard Unix facilities, for example, signals and process control, are extended to any process in the system; independent of which node they are currently running on. In that case, the debugger does not need to change, although adding support to debug multiple processes at the same time makes it a more useful tool.

If the OS is not capable of providing these services, they can be implemented in the debugger with the help of runtime system features such as daemons and remote procedure calls (RPC).

## Process Migration

Migrating a process to another node is useful for load balancing and as a precautionary measure before a node is marked down. Migrating a process requires that all data structures, such as the Process Control Block (PCB), which the OS maintains for this process, migrate in addition to the code and user-level data structures the process has created. A completely transparent migration also requires that file handles to open files and those used for IPC are migrated as well. That, in return, requires that some information remains in the local OS about where the process has migrated to. (Or that all connections to the migrating process are updated before the migration.) For example, another

process on the local node which was communicating with the migrated process via IPC will continue to do so. When data arrives for the migrated process, the OS needs to forward the data to the new location. If a process migrates multiple times, forwarding becomes more expensive and convoluted.

### Stdin/Stdout

The default channels for input to and output from a process under UNIX are called stdin and stdout. If a user is logged into one node and starts a process on another, then the OS transfers data between these two nodes on behalf of the user. If an application consists of multiple processes, the OS ensures that output from all of them gets funneled through the stdout channel to the user. Data from the stdin channel usually does not get replicated and only the first process that reads it, will receive a copy. The OS must maintain these data channels even if processes migrate to other nodes, otherwise the illusion of a single system image would be broken.

### File System

In a system managed by an SSI, all processes should see the same directory (folder) and file hierarchy. This feature, called shared root, is what is available on a desktop system where multiple processes can access files on the local disk. In a parallel system, files are often distributed among multiple disks to enhance performance and scalability. Distributing files like that complicates things, since metadata (information about files and directories) may not be located on the same device as the files themselves. Yet, metadata must be kept up-to-date to preserve file semantics. For example, two processes appending data to the end of a file must not overwrite each other's data. That requires that the file pointer always points to the true end of the file.

Some files which the OS frequently accesses should be local to the node. However, some files, such as /etc/passwd, should be the same on all nodes and a system administrator should have the ability to update only one of them and then propagate the changes to all copies. On the other hand, some (pseudo) files that describe local devices for example, need to be unique on each node. If a process migrates and had one of these files open, it must continue to interact with the device on the original node, even though a file with the same name exists on the new node as well. Yet for other files, temporary files

for example, it may be desirable that they are only visible locally. File systems that can handle all these cases are complex and need to be tightly integrated with the OS.

### System Calls

System calls are functions an application uses to interact with the OS. Calls for opening, closing, reading, and writing a file are typical. So are calls to create and control other processes. An SSI that wants to maintain these calls unchanged in a distributed system requires changes to the OS kernel. For example, process creation may not be strictly local anymore, if the system tries to balance load by creating new processes on less busy nodes.

### Checkpoint/Restart

Long-running applications often do not finish, due to an interruption by a faulty component or a software problem. This is especially true for parallel applications, where it is more likely that at least one of the many nodes it runs on will fail. To combat this problem, an application occasionally writes intermediate data and its current state to disk. When a failure occurs, the application is restarted on a different node, or on the same node after it has been rebooted or repaired. The application then reads the checkpoint data written earlier and continues its computation.

This can be done by the application itself, in the runtime system, or in the OS kernel. The latter has the nice property that application writers do not need to worry about it, but it has the potential for waste, since the kernel cannot know what data the application will need to restart. Therefore, it has to checkpoint the complete application state and all of its data. Checkpoint/restart needs to be integrated with the SSI system, even if checkpointing is done at the application level.

### Batch System

Many parallel systems use a batch scheduler to increase throughput of jobs through the system and to allow it to continue processing unattended, for example, at night or on weekends. Users submit scripts that describe the application they want to run, how much time they require, and how many nodes the application needs. The batch scheduler uses the job priority and knowledge about available nodes to select and run the jobs currently enqueued for processing.

A batch system is one of the most common SSI components in a cluster. It allows the user to treat the collection of machines as a single system and use a single command to submit jobs to it. A batch system provides additional commands to view status information about the currently enqueued and running jobs and to manage them. All these commands treat jobs as if they were a single process on the local system.

### Login Load Leveling

Parallel systems usually have more than one node where users can log in. In a partitioned system, there is a service partition comprised of the nodes dedicated to user login and common tasks such as editing files and compiling applications. Parallel jobs are then run on nodes that have been assigned to the compute partition. Often, nodes in the service partition are of a different type; they may have more memory or local disks attached. Smaller clusters may not use partitioning, though that is less efficient. In either case, the SSI should present a single address to the outside world. When a user logs in at that address, the SSI picks one of the service nodes for that user to log in. The choice is usually made based on the current work load of each node in the service partition. As far as the user is concerned, there is only one login node.

### System Log Management

A lot of information about the current state of the system is stored in log files that are unique to each node of a parallel system. System administrators consult these logs to diagnose problems and to detect and prevent faults before they occur (e.g., warnings about the rising temperature inside a CPU). An SSI should provide functions to aggregate these individual files and let system administrators search through, purge, or archive them with a single command.

### Software Maintenance

The software versions of the system utilities, libraries, and applications installed on each node must be the same system-wide or at least be compatible with each other. As with system log management, an SSI should provide functions to system administrators that allow them to upgrade and check software packages on multiple nodes with a single command.

### Imperfect SSI

The previous sections listed the features an SSI must provide and a glance at what implementing them entails. Many of these features are independent components and useful even when none of the other components are available. In fact, no SSI running on more than a couple hundred of nodes implements all features described thus far. The reason for that is scalability. Some features work well when used on eight nodes, but become unusably slow when extended much beyond that number.

Among the most costly features described earlier is process migration, especially if an SSI tries to achieve complete transparency. Once a process starts interacting with other processes on a node and obtains file handles to node-local resources, it becomes difficult to migrate that process. After migration, a forwarding mechanism must be in place to route information to the new location of the migrated process. If the original node retains information about the new location of the migrated process, a failure of that node will disrupt the flow of information. If no information is kept at the original location, then all processes that are communicating with the migrated process must be told about the new location. If migration is frequent, and interaction among processes is high, this becomes very costly. Furthermore, it does not always work, for example, if the migrated process was using a peripheral that is only available on the original node. Another high-cost SSI item is shared memory. Even when supported by hardware, the illusion of a single, large memory starts to fall apart as the number of nodes (CPUs) increases.

Not all is lost, though. Some systems deploy additional hardware to assist with some SSI features. For example, some systems have an additional network and disks dedicated to system software maintenance. Furthermore, complete transparency is not always required, and users are willing to use different commands and utilities when dealing with parallel processes, in return for better performance and scalability. For example, using a command such as `qsub` to submit a parallel job does not seem a high burden compared to the transparent method of just naming the application on the command line to start it. The UNIX command `ps` is used to obtain process status information. In a transparent SSI system, it would return that information even if the process is not local, or return information

about many processes, if they are part of a parallel application.

In contrast, users often only want to know whether their job is running and whether it is making progress. The batch system has a command that can answer at least the former question and a `ps` output for hundreds of processes scattered throughout the system would not provide any additional useful information.

Therefore, even a partial SSI can be of benefit. Application users benefit most from a batch scheduling system. This also assists system administrators who also need a way to efficiently manage the hardware and software of the system. Most other functionality, especially less scalable features, are not strictly necessary to make a parallel system useful. That may mean additional utilities that expose the distributed aspects of the system and modifications to applications to allow them to work in parallel and with greater efficiency.

## Future Directions

Desktop systems with two or four CPU cores are common now. High-performance systems will be the first to receive CPUs with tens of cores, but those CPUs will soon be common. Desktop OSs will adapt to these systems and provide SSI features to manage and use the cores in a single system. Users will expect to see these features in parallel systems as well, which will drive the development of SSI features and integration into parallel OSs.

At the very high-end, exascale systems are on the horizon. Implementing SSI for these systems will be a challenge due to the size of these systems. Displaying the status of a hundred-thousand nodes in a readable fashion without endless scrollbars is only feasible with intelligent software that zooms in on interesting events and features of the system. Commercial tools for applications that span that many nodes do not exist yet. Simple tools have a better chance at scaling to these future systems, but managing and debugging such large applications and systems will remain a challenge.

## Related Entries

- [Checkpointing](#)
- [Clusters](#)
- [Fault Tolerance](#)
- [Operating System Strategies](#)
- [Scalability](#)

## Bibliographic Notes and Further Reading

Overviews of SSI are presented in [3], Chapter 11 of [13], and [8]. The descriptions of actual systems are out of date or the systems are not in use anymore. Nevertheless, the fundamental features and issues described are still valid.

Many papers discuss specific aspects of SSI. For example, [5] and [4] discuss partitioning and present job launch mechanisms. In [12], the authors discuss specific SSI challenges for petascale systems. SSI tools to make system administration easier are presented in [2, 7, 15].

Process migration is discussed in detail in [10] and specific implementations appear in MOSIX [1], BProc [6], and IBM's WPAR [9], among others. The survey in [16] is a little bit dated, but it gives a good description of so-called worms, which are processes specifically designed to migrate.

Implementations of SSI, most of them partial, abound even before the term SSI was coined. Examples include LOCUS [17], Plan 9 [14], MOSIX [1], and Kerrighed [11].

## Bibliography

1. Barak A, La'adan O (1998) The MOSIX multicomputer operating system for high performance cluster computing. Future Gener Comput Syst 13(4–5):361–372
2. Brightwell R, Fisk LA, Greenberg DS, Hudson T, Levenhagen M, Maccabe AB, Riesen R (2000) Massively parallel computing using commodity components. Parallel Comput 26(2–3):243–266
3. Buyya R, Cortes T, Jin H (2001) Single system image. Int J High Perform Comput Appl 15(2):124–135
4. Frachtenberg E, Petriini F, Fernandez J, Pakin S, Coll S (2002) STORM: lightning-fast resource management. In: Supercomputing '02: proceedings of the 2002 ACM/IEEE conference on supercomputing, Baltimore. IEEE Computer Society, Los Alamitos, pp 1–26
5. Greenberg DS, Brightwell R, Fisk LA, Maccabe AB, Riesen R (1997) A system software architecture for high-end computing. In: SC'97: high performance networking and computing: proceedings of the 1997 ACM/IEEE SC97 conference, San Jose, Raleigh, 15–21 Nov 1997. ACM/IEEE Computer Society, New York
6. Hendriks E (2002) BPROC: the Beowulf distributed process space. In: ICS '02: proceedings of the 16th international conference on supercomputing, pp 129–136. ACM, New York
7. Hendriks EA, Minnich RG (2006) How to build a fast and reliable 1024 node cluster with only one disk. J Supercomput 36(2):171–181
8. Hwang K, Jin H, Chow E, Wang C-L, Xu Z (1999) Designing SSI clusters with hierarchical check pointing and single I/O space. IEEE Concurr 7(1):60–69

9. Kharat S, Mishra R, Das R, Vishwanathan S (2008) Migration of software partition in UNIX system. In: Compute '08: proceedings of the first Bangalore annual compute conference, Bangalore. ACM, New York, pp 1–4
10. Milojčić DS, Dougulis F, Paindaveine Y, Wheeler R, Zhou S (2000) Process migration. ACM Comput Surv 32(3):241–299
11. Morin C, Gallard P, Lottiaux R, Vallée G (2004) Towards an efficient single system image cluster operating system. Future Gener Comput Syst 20(4):505–521
12. Ong H, Vetter J, Studham RS, McCurdy C, Walker B, Cox A (2006) Kernel-level single system image for petascale computing. SIGOPS Oper Syst Rev 40(2):50–54
13. Pfister GF (1998) In search of clusters: the ongoing battle in lowly parallel computing, 2nd edn. Prentice-Hall, Upper Saddle River
14. Pike R, Presotto D, Thompson K, Trickey H, Winterbottom P (1993) The use of name spaces in Plan 9. SIGOPS Oper Syst Rev 27(2):72–76
15. Skjellum A, Dimitrov R, Angaluri SV, Lifka D, Coulouris G, Uthayopas P, Scott SL, Eskicioglu R (2001) Systems administration. Int J High Perform Comput Appl 15(2): 143–161
16. Smith JM (1988) A survey of process migration mechanisms. SIGOPS Oper Syst Rev 22(3):28–40
17. Walker B, Popok G, English R, Kline C, Thiel G (1983) The LOCUS distributed operating system. In: SOSP '83: proceedings of the ninth ACM symposium on operating systems principles, Bretton Woods. ACM, New York, pp 49–70

## Singular-Value Decomposition (SVD)

► [Eigenvalue and Singular-Value Problems](#)

## Sisal

JOHN FEO  
Pacific Northwest National Laboratory, Richland,  
WA, USA

### Definition

Streams and Iterations in a Single Assignment Language (Sisal) was a general-purpose applicative language developed for shared-memory and vector supercomputer systems. It provided an hierarchical intermediate form, parallel runtime system, optimizing compiler,

and programming environment. The language was strongly typed. It supported both array and stream data structures, and had both iterative and parallel loop constructs.

## Discussion

### Introduction

Streams and Iterations in a Single Assignment Language (Sisal) was a general-purpose applicative language defined by Lawrence Livermore National Laboratory, Colorado State University, University of Manchester and Digital Equipment Corporation in the early 1980s [1]. Lawrence Livermore and Colorado State developed the language over the next 2 decades. They maintained the language definition, compiler and runtime system, programming tools, and provided education and user support services. The language version used widely in the 1980s and 1990s was Sisal 1.2 [2]. Three Sisal user conferences were held in 1991, 1992, and 1993 [3, 4].

The Sisal Language Project had four major accomplishments: (1) IFI, an intermediate form used widely in research [5]; (2) OSC, an optimizing compiler that preallocated memory for most data structures and eliminated unnecessary coping and reference count operations [7]; (3) a highly efficient, threaded runtime system that supported most shared-memory, parallel computer systems [6], and (4) execution performance and scalability equivalent to imperative languages [1, 8].

### Language Definition

Sisal's applicative semantics and keyword-centric syntax proved to be a simple, easy-to-use, easy-to-read parallel programming language that facilitated algorithm development and simplified compilation. Since the value of any expression depended only on the values of its subexpressions and not on the order of evaluation, Sisal programs defined the data dependencies among operations, but left the scheduling of operations, the communication of data values, and the synchronization of concurrent operations to the compiler and runtime system.

Sisal was strongly typed. Programmers specified the types of only function parameters; the compiler inferred all other types. All functions were mathematically sound. A function had access to only its arguments and there were no side effects. Each function invocation was independent and functions did not retain state

between invocations. Sisal programs were referentially transparent and single-assignment. Since a name was bound to one value and not a memory location, there was no aliasing. In general, a Sisal program defined a set of mathematical expressions where names stood for specific values, and computations progressed without state making the transformation from source code to dataflow graph trivial.

Sisal supported the standard scalar data types: boolean, character, integer, real, and double precision. It also included the aggregate types: array, stream, record, and union. All arrays were one-dimensional; multi-dimensional arrays were arrays of arrays. The type, size, and lower bound of an array was not defined explicitly, rather it was a consequence of execution. Arrays were created by gathering elements, “modifying” existing arrays, or catenations.

A stream was a sequence of values of uniform type. In Sisal, stream elements were accessible in order only; there was no random access. A stream could have only one producer, but many consumers. By definition, Sisal streams were nonstrict – each element was available as soon as it was produced. The Sisal runtime system supported the concurrent execution of the producer and consumer(s).

The two major language constructs were *for initial* and *for* expressions. The former expression substituted for sequential iteration in conventional languages, but retained single assignment semantics. The rebinding of loop names to values was implicit and occurred between iterations. Loop names prefixed with *old* referred to previous values. A returns clause defined the results and arity of the expression. Each result was either the final value of some loop name or a reduction of the values assigned to a loop name during loop execution. The following *for initial* expression returns the prefix sum of A

```
for initial
 i := 0;
 x := A[i]
while i < n repeat
 i := old i + 1;
 x := old x + A[i]
returns array of x
end for
```

Sisal supported seven intrinsic reductions: *array of*, *stream of*, *catenate*, *sum*, *product*, *least*, and *greatest*. The order of reduction was determinate.

The *for* expression provided a way to specify independent iterations. The semantics of the expression did not allow references to values defined in other iterations. The expression was controlled by a range expression that could be simple, a dot product of ranges, or a cross product of ranges. The next two expressions compute the dot and cross product of two arrays of size *n* and *m*

```
for i in 0 to n - 1 dot j
 in 0 to m - 1
 x := A[i] * B[j]
returns sum of x
end for

for i in 0 to n - 1 cross j
 in 0 to m - 1
 x := A[i] * B[j]
returns array of x
end for
```

Note that the expression does not indicate how the parallel iterations should be scheduled. The management of parallel threads was the responsibility of the compiler and runtime systems and was highly tuned for each architecture. This separation of responsibility greatly simplified Sisal programs, improved portability, and made the language easy to use.

### Build-in-Place Analysis

The Sisal compile process comprised eight phases shown in Fig. 1. The two most important phases concerned build-in-place (IF2MEM) and update-in-place analysis (IF2UP). IF2 was a superset of IF1. It was not applicative as it included operations (AT-nodes) that directly referenced and manipulated memory.

Prior to Sisal, functional programming languages were considered very inefficient. Since the size and structure of aggregate data types are a consequence of execution, without analysis aggregates must be built one element at a time possibly requiring elements to be copied many times as the aggregate grows in size. Moreover, strict adherence to single-assignment semantics requires construction of a new array whenever a single element is modified.



**Sisal.** Fig. 1 OSC phases

Build-in-place analysis attacked the incremental construction problem [6]. The algorithm had two-passes. Pass 1 visited nodes in data flow order, and built, where possible, an expression to calculate an array's size at run time. The expression was a function of the semantics of the array constructor and the size expressions of its inputs. Determining the size of an array built during loop execution required an expression to calculate the number of loop iterations before the loop executes. Deriving this expression was not possible for all loops. Pass 1 concluded by pushing, in the order of encounter, the nodes for the size expression onto a stack.

Pass 2 removed nodes from the stack, inserted them into the data flow graph, and appropriately wired memory references among them by inserting edges transmitting pointers to memory. If a node was the parent of an already converted node, it could build its result directly into the memory allocated to its child, thus eliminating the intermediate array. If the node was not the parent of an already converted node, it built its result in a new memory location. Note that the ordering of nodes on the stack guaranteed processing of children before parents.

### Update-in-Place Analysis

Update-in-place analysis attacked the aggregate update problem [7]. OSC introduced dependences in the data flow graph to schedule readers of an aggregate object before writers. It used reference counts to identify the last use of an object. If the last use was a write, then the write was executed in place. By re-ordering nodes and aggregating reference copy operations, OSC eliminated up to 98% of explicit reference count operations in larger programs without reducing parallelism. Consequently, the inefficiencies of reference counting largely disappeared and the benefits of updating aggregate objects were fully enjoyed by Sisal programs.

OSC considered an array as two separately reference-counted objects: a dope vector defining the array's logical extent and the physical space containing its constituents. As array elements are computed and stored, dope vectors cycle between dependent nodes to communicate the current status of the regions under construction. As a result, multiple dope vectors might reference the same or different regions of the physical space and the individual participants in the construction would produce new dope vectors to communicate the current status of the regions to their predecessors. Without update-in-place optimization, each stage in construction would copy a dope vector.

### Runtime System

The runtime software supported the parallel execution of Sisal programs, provided general purpose dynamic storage allocation, implemented operations on major data structures, and interfaced with the operating system for input/output and command line processing [7].

The Sisal runtime made modest demands of the host operating system. At program initiation a command line option specified the number of operating system processes (workers) to instantiate for the duration of the program. The runtime system maintained two queues of threads: the For Pool and the Ready List. A worker was always in one of three modes of operation: executing a thread from the For Pool, executing a thread from the Ready Pool, or idle. The runtime system allocated stacks for threads on demand, but every effort was made to reuse previously allocated stacks and thus reduce allocation and deallocation overhead.

The For Pool maintained slices of *for* expressions. A worker would pick up a slice, execute it, and return to the pool for another slice. The worker that executed the last slice of an expression placed the parent thread on the Ready List, and returned to the For Pool to look for slices from other expressions. By persisting in executing slices of *for* expressions, a worker avoided deallocating and allocating its stack.

The Ready List maintained all other threads made executable by events, such as the main program or function calls. If a worker found no work on either Pool, it examined the Storage Deallocation List for deferred storage deallocation work.

Sisal relied on dynamic storage allocation for many data values such as arrays and streams, and for internal objects such as thread descriptors and execution stacks. It used a two-level allocation mechanism that was fast, efficient, and parallelizable. A standard boundary tag scheme was augmented with multiple entry points to a circular list of free blocks. Both the free list search and deallocation of blocks were parallelized, even though the former sometimes completely removed blocks from the list and the latter coalesced physically adjacent but logically distant blocks. To increase speed, an exact-fit caching mechanism was interposed before the boundary tag scheme. It used a working set of different sizes of recently freed blocks.

## Performance

As noted above, most Sisal programs executed as fast as their imperative language counterparts. Consequently,

Sisal attracted a wide user community that developed many different types of parallel applications, including benchmark codes, mathematical methods, scientific simulations, systems optimization, and bioinformatics.

[Table 1](#) lists the execution times of four applications and [Table 2](#) gives compilation statistics for each program. Columns 2–4 give the number of static arrays built, preallocated, and built-in-place; columns 5–8 list the number of copy, conditional copy and reference count operations after optimization, and the number of artificial dependency edges introduced by the compiler.

It is instructive to examine two of the applications. SIMPLE was a two-dimensional Lagrangian hydrodynamics code developed at Lawrence Livermore National Laboratory to simulate the behavior of a fluid in a sphere. The hydrodynamic and heat condition equations are solved by finite difference methods. A tabular ideal gas equation determines the relation between state variables. The implementation of SIMPLE in Sisal 1.2 was straightforward and highly parallel.

The compiler preallocated and built all arrays in place (261 of them), eliminated all absolute copy operations, marked 19 copy operations for run time check, and eliminated 2,005 out of 2,066 reference count operations. The 19 conditional copy operations were a result of row sharing. Since they always tested false at runtime, no copy operations actually occurred. For 62 iterations of a  $100 \times 100$  grid problem, the Sisal and Fortran versions of SIMPLE on one processor executed in 3,099.3 and 3,081.3 s, respectively. On ten processors, the Sisal code realized a speedup of 7.3. An equivalent of 1.5 processors

**Sisal. Table 1** Execution times

| Program             | Fortran  | Sisal (#p)  | Sisal (#p)  | Speedup |
|---------------------|----------|-------------|-------------|---------|
| GaussJordan         | 54.0 s   | 54.5 s (1)  | 8.8 s (10)  | 6.2     |
| RICARD              | 30.6 h   | 31.0 h      | 3.5 h (10)  | 9.0     |
| SIMPLE              | 3081.3 s | 3099.3 s(1) | 422.0 (10)  | 7.3     |
| Simulated annealing | 476.6 s  | 956.2 s (1) | 267.8 s (5) | 3.6     |

**Sisal. Table 2** OSC optimizations

| Program             | Arrays | PreAllocated | Built inplace | Copy | CCopy | Ref counts | Artificial edges |
|---------------------|--------|--------------|---------------|------|-------|------------|------------------|
| GaussJordan         | 7      | 7            | 7             | 0    | 0     | 1          | 9                |
| RICARD              | 29     | 29           | 28            | 0    | 6     | 7          | 5                |
| SIMPLE              | 261    | 261          | 261           | 0    | 19    | 61         | 347              |
| Simulated annealing | 46     | 46           | 42            | 0    | 4     | 41         | 168              |

was lost in allocating and deallocating two-dimensional arrays. The implementation of true multi-dimensional arrays was proposed in both Sisal 2.0 [9] and Sisal 90. [These language definitions were defined near the end of the project, but never implemented.]

Simulated annealing is a generic Monte Carlo optimization technique for solving many difficult combinatorial problems. In the case of the school timetable problem, the objective is to assign a set of tuples to a fixed set of time slots (periods) such that no critical resource is scheduled more than once in any period. Each tuple is a record of four fields: class, room, subject, and teacher. Classes, rooms, and teachers are critical resources; subjects are not. At each step of the procedure, a tuple is chosen at random and moved to another period. If the new schedule has equivalent or lower cost, the move is accepted. If the new schedule has higher cost, the move is accepted with a certain exponential probability. If the move is not accepted, the tuple is returned to its original period. The procedure can be parallelized by simultaneously choosing one tuple from each nonempty period and applying the move criterion to each. The accepted moves are then carried out one at a time.

OSC preallocated memory for all the arrays, and built all but four of the arrays in place. It removed all absolute copy operations, marked four copy operations for run time check, and removed all but 41 reference count operations. The four arrays not built in place resulted from adding a tuple to a period. Although the compiler did not mark the new periods for build-in-place, the periods were rarely copied. When an element was removed from the high-end of an array, the array's logical size shrunk by one, but its physical size remained constant. Thus, when an element was added to a period, there was often space to add the element without copying. Whenever copying occurred, extra space was allocated to accommodate future growth. This foresight saved over 15,000 copies in this application at the cost of a few hundred bytes of storage.

## Bibliographic Notes and Further Reading

For a summary of the project and details on the compiler and runtime system see [1]. The language definition can be found in [2]. IFI is described [5].

Build-in-place analysis and update-in-place analysis are described in more detail in [6] and [7], respectively.

In 1991 and then again in 1992 two new versions of the language, Sisal 2.0 and Sisal 90, were defined. These versions included: higher-order functions, user-defined reductions, parameterized data types, foreign language modules, array syntax, and rectangular arrays. While compilers for the version were never released there is some information available on the web.

## Bibliography

1. Feo JT, Cann D, Oldehoeft R (1990) A report of the Sisal Language Project. *J Parallel Distrib Comput* 10(4):349–366
2. McGraw J et al (1985) Sisal: streams and iterations in a single-assignment language, reference manual version 1.2. Lawrence Livermore National Laboratory Manual M-146, Livermore, CA, September 1985
3. Feo J (ed) Proceedings of the 2nd Sisal User Conference. Lawrence Livermore National Laboratory, CONF-9210270, San Diego, CA, October 1992
4. Feo J (ed) Proceedings 3rd Sisal User Conference. Lawrence Livermore National Laboratory, CONF-9310206, San Diego, CA, October 1993
5. Skedzielewski S, Glauert J (1985) IFI – an intermediate form for applicative languages. Lawrence Livermore National Laboratory Manual M-170, Livermore National Laboratory, Livermore, CA, September 1985
6. Ranelletti J (1996) Graph transformation algorithms for array memory optimization in applicative languages. PhD dissertation, University of California at Davis, CA, May 1996
7. Cann D (1998) Compilation techniques for high performance applicative computation. PhD dissertation, Colorado State University, CO, May 1998
8. Cann D (1992) Retire Fortran?: A debate rekindled. *Commun ACM* 35(8):81–89
9. Oldehoeft R et al (1991) The Sisal 2.0 reference manual. Lawrence Livermore National Laboratory, Technical Report UCRL-MA-109098, Livermore, CA, December 1991

## Small-World Network Analysis and Partitioning (SNAP) Framework

►SNAP (Small-World Network Analysis and Partitioning) Framework

## SNAP (Small-World Network Analysis and Partitioning) Framework

KAMESH MADDURI

Lawrence Berkeley National Laboratory, Berkeley,  
CA, USA

### Synonyms

Graph analysis software; Small-world network analysis and partitioning (SNAP) framework

### Definition

SNAP (Small-world Network Analysis and Partitioning) is a framework for exploratory analysis of large-scale complex networks. It provides a collection of optimized parallel implementations for common graph-theoretic problems.

## Discussion

### Introduction

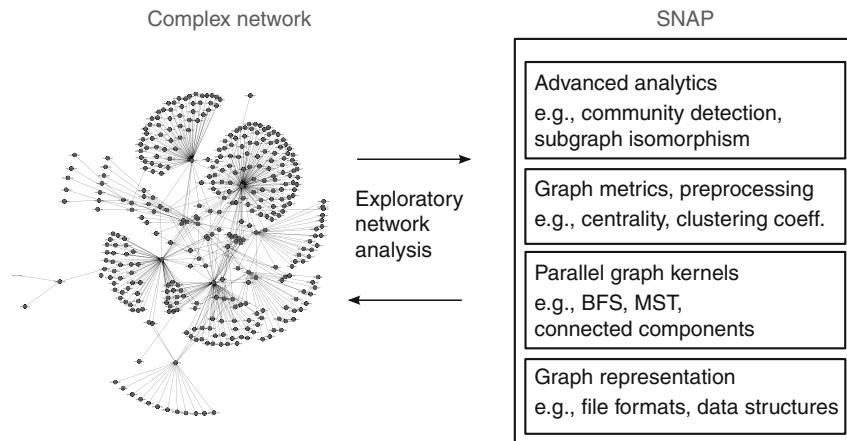
Graphs are a fundamental abstraction for modeling and analyzing data, and are pervasive in real-world applications. Transportation networks (road and airline traffic), socio-economic interactions (friendship circles, organizational hierarchies, online collaboration networks), and biological systems (food webs, protein interaction networks) are a few examples of data that can be naturally represented as graphs. Understanding the dynamics and evolution of real-world network abstractions is an interdisciplinary research challenge with wide-ranging implications. Empirical studies on networks have led to a variety of models to characterize their topology and evolution. Quite surprisingly, it has been shown that graph abstractions arising from diverse systems such as the Internet, social interactions, and biological networks exhibit common structural features such as a low diameter, unbalanced degree distributions, self-similarity, and the presence of dense subgraphs [1, 11, 14]. Some of these topological features are captured by what is known as the *small-world* model or phenomenon.

The analysis of large graph abstractions, particularly small-world complex networks, raises interesting computational challenges. Graph algorithms are typically

highly memory intensive, make heavy use of data structures such as lists, sets, queues, and hash tables, and exhibit a combination of data and task-level parallelism. On current workstations, it is infeasible to do exact in-core computations on graphs larger than 100 million vertex/edge entities (*large* in general refers to the typical problem size for which the graph and the associated data structures do not fit in 2–4 GB of main memory). In such cases, parallel computers can be utilized to obtain solutions for memory and compute-intensive graph problems quickly. Due to power constraints and diminishing returns from instruction-level parallelism, the computing industry is rapidly converging towards widespread use of multicore chips and accelerators. Unfortunately, several known parallel algorithms for graph-theoretic problems do not easily map onto clusters of multicore systems. The mismatch arises due to the fact that current systems lean towards efficient execution of regular computations with low memory footprints and working sets, and heavily penalize memory-intensive applications with irregular memory accesses; however, parallel graph algorithms in the past were mostly designed assuming an underlying, well-balanced compute-memory platform. The small-world characteristics of real networks, and the load-balancing constraints they impose during parallelization, represent an additional challenge to the design of scalable graph algorithms.

SNAP [2, 10] is an open-source computational framework for graph-theoretic analysis of large-scale complex networks. It is intended to be an optimized collection of computational kernels (or algorithmic building-blocks) that the end-user could readily use and compose to answer higher-level, ad-hoc graph analysis queries. The target platforms for SNAP are shared-memory multicore and symmetric multiprocessor systems. SNAP kernels are implemented in C and use OpenMP for parallelization. On distributed memory systems, SNAP can be used for intra-node parallelization, and the user has to manage inter-node communication, and also identify and implement parallelism at a coarser node-level granularity.

The parallel graph algorithms in SNAP are significantly faster than alternate implementations in other open-source graph software. This is due to a combination of the use of memory-efficient data structures, preprocessing kernels that are tuned for small-world



**SNAP (Small-World Network Analysis and Partitioning) Framework.** Fig. 1 A schematic of the SNAP graph analysis framework [8]

networks, as well as algorithms that are designed to specifically target cache-based multicore architectures. These issues are discussed in more detail in the following sections of this article.

As the project title suggests, the initial design goals of SNAP were to provide scalable parallel solutions for community structure detection [4], a problem variant of graph partitioning that is of great interest in social, and in general, small-world network analysis. Community structure detection is informally defined as identifying densely connected sets of vertices in the network, thereby revealing latent structure in a large-scale network. It is similar to the problem of graph partitioning in scientific computing, as is usually formulated as a graph clustering problem. SNAP includes several different parallel algorithms for solving this problem of community detection.

## Graph Representation

The first issue that arises in the design of graph algorithms is the use of appropriate in-memory data structures for representing the graph. The data to be analyzed typically resides on disk in a database, or in multiple files. As the data is read from disk and loaded into main memory, a graph representation is simultaneously constructed. The minimal layout that would constitute an in-memory graph representation is a list of edge tuples with vertex identifiers indicating the source and destination vertices, and any attributes associated with the

edges and vertices. However, this does not give one easy access to lists of edges originating from a specific vertex. Thus, the next commonly used representation is to sort the edge tuple list by the source vertex identifier, and store all the adjacencies of a particular vertex in a contiguous array. This is the primary *adjacency list* representation of graphs that is supported in SNAP. Edges can have multiple attributes associated with them, in which case they can either be stored along with the corresponding adjacency vertex identifier, or in separate auxiliary arrays. This representation is space-efficient, has a low computational overhead for degree and membership queries, and provides cache-friendly access to adjacency iterators.

In cases where one requires periodic structural updates to the graph, such as insertions and deletions of edges, SNAP uses alternate graph representations. An extension to the static representation is the use of dynamic, resizable adjacency arrays. Clearly, this would support fast insertions. Further, parallel insertions can be supported using non-blocking atomic increment operations on most of the current platforms. There are two potential parallel performance issues with this data structure. Edge deletions are expensive in this representation, as one needs to scan the entire adjacency list in the worst case to locate the required tuple. The scan is particularly expensive for high-degree vertices. Second, there may be load-balancing issues with parallel edge insertions (for instance, consider the case where there are a stream of insertions to the adjacency list of

a single vertex). These problems can be alleviated by batching the updates, or by randomly shuffling the updates before scheduling the insertions. If one uses sorted adjacency lists to address the deletion problem, then the cost of insertions goes up due to the overhead of maintaining the sorted order.

An alternative to arrays is the use of tree structures to support both quick insertions and deletions. Treaps are binary search trees with a priority (typically a random number) associated with each node. The priorities are maintained in heap order, and this data structure supports insertions, deletions, and searching in average-case logarithmic time. In addition, there are known efficient parallel algorithms for set operations on treaps such as union, intersection, and difference. Set operations are particularly useful to implement kernels such as graph traversal and induced subgraphs, and also for batch-processing updates.

To process both insertions and deletions efficiently, and also given the power-law degree distribution for small-world networks, SNAP supports a hybrid representation that uses dynamically resizable arrays to represent adjacencies of *low-degree* vertices, and treaps for *high-degree* vertices. The threshold to determine low and high-degree vertices can be varied based on the data set characteristics. By using dynamic arrays for low-degree vertices (which will be a majority of vertices in the graph), one can achieve good performance for insertions. Also, deletions are fast and cache-friendly for low-degree vertices, whereas they take logarithmic time for high-degree vertices represented using treaps. Madduri and Bader [9] discuss the parallel performance and space-time trade-offs involved with each of these representations in more detail.

### Parallelization Strategies

There is a wide variety in the known approaches for exploiting parallelism in graph problems. Some of the easier ones to implement and analyze are computations involving iterations over vertex and edge lists, without much inter-iteration dependencies. For instance, queries such as determining the top- $k$  high-degree vertices, or the maximum-weighted edge in the graph, can be easily parallelized. However, most parallel algorithms require the use of data structures such as priority queues and multisets. Further, one needs support for fast parallel operations on these structures, such as

parallel insertions, membership queries, and batched deletions. Fine-grained, low overhead synchronization is an important requirement for several efficient parallel implementations. Further, the notion of *partitioning* a graph is closely related to several parallel graph algorithms. Given that inter-processor communication is expensive on distributed memory systems (compared to computation on a single node), several parallel graph approaches rely on a graph partitioning and vertex reordering preprocessing step to minimize communication in subsequent algorithm executions. Graph partitioning is relevant in the context of shared-memory SNAP algorithms as well, since it reduces parallel synchronization overhead and improves locality in memory accesses.

Consider the example of breadth-first graph traversal (BFS) as an illustration of more complex paradigms for parallelism in graph algorithms. Several SNAP graph kernels are designed to exploit fine-grained thread-level parallelism in graph traversal. There are two common parallel approaches to breadth-first search: *level-synchronous* graph traversal, where the adjacencies of vertices at each level in the graph are visited in parallel; and *path-limited searches*, where multiple searches from vertices that are *far apart* are concurrently executed, and the independent searches are aggregated. The level-synchronous approach is particularly suited for small-world networks due to their low graph diameter. Support for fine-grained efficient synchronization is critical in both these approaches. The SNAP implementation of BFS aggressively reduces locking and barrier constructs through algorithmic changes as well as architecture-specific optimizations. For instance, the SNAP BFS implementation uses a lock-free approach for tracking visited vertices and thread-local work queues, significantly reducing shared memory contention. While designing fine-grained algorithms for small-world networks, it is also important to take the unbalanced degree distributions into account. In a level-synchronized parallel BFS where vertices are statically assigned to multiple threads of execution, it is likely that there will be phases with severe work imbalance (due to the imbalance in vertex degrees). To avoid this, the SNAP implementation first estimates the processing work to be done at each vertex, and then assigns vertices accordingly to threads. Several other optimization techniques, and their relative performance

benefits, are discussed by Bader and Madduri in more detail [2].

Graph algorithms often involve performance trade-offs associated with memory utilization and parallelization granularity. In cases where the input graph instance is small enough, one could create several replicates of the graph or the associated data structures for multiple threads to execute concurrently, and thus reduce synchronization overhead. SNAP utilizes this technique for the exact computation of betweenness centrality, which requires a graph traversal from each vertex. The fine-grained implementation parallelizing each graph traversal requires space linear in the number of vertices and edges, whereas the coarse-grained approach (the traversals are distributed among  $p$  threads) incurs a  $p$ -way multiplicative factor memory increase. Depending on the graph size, one could choose an appropriate number of replicates to reduce the synchronization overhead.

## SNAP Kernels for Exploratory Network Analysis

Exploratory graph analysis often involves an iterative study of the structure and dynamics of a network, using a discriminating selection of topological metrics. SNAP supports fast exact and approximate computation of well-known social network analysis metrics, such as average vertex degree, clustering coefficient, average shortest path length, rich-club coefficient, and assortativity. Most of these metrics have a linear or sub-linear computational complexity, and are straightforward to implement. When used appropriately, they not only provide insight into the network structure, but also help speed up subsequent analysis algorithms. For instance, the average neighbor connectivity metric is a weighted average that gives the average neighbor degree of a degree- $k$  vertex. It is an indicator of whether vertices of a given degree preferentially connect to high- or low-degree vertices. Assortativity coefficient is a related metric, which is an indicator of community structure in a network. Based on these metrics, it may be possible to say whether the input data is representative of a specific common graph class, such as bipartite graphs or networks with pronounced community structure. This helps one choose an appropriate community detection algorithm and also the right clustering measure to optimize for. Other preprocessing kernels that are beneficial

for exposing parallelism in the problem include computation of connected and biconnected components of the graph. If a graph is composed of several large connected components, it can be decomposed and individual components can be analyzed concurrently. A combination of these preprocessing kernels before the execution of a compute-intensive routine often lead to a substantial performance benefit, either by reducing the computation by pruning vertices or edges, or by exploring more coarse-grained concurrency and locality in the problem.

## Community Identification Algorithms in SNAP

Several routines in SNAP are devoted to solving the community identification problem in small-world networks. Graph partitioning and community identification are related problems, but with an important difference: the most commonly used objective function in partitioning is minimization of edge cut, while trying to *balance the number of vertices* in each partition. The number of partitions is typically an input parameter for a partitioning algorithm. Clustering, on the other hand, optimizes for an appropriate application-dependent measure, and the number of clusters is unknown beforehand. Multilevel and spectral partitioning algorithms (e.g., Chaco [6] and Metis [7]) have been shown to be very effective for partitioning graph abstractions derived from physical topologies, such as finite-element meshes arising in scientific computing. However, the edge cut when partitioning small-world networks using these tools is nearly two orders of magnitude higher than the corresponding edge cut value for a nearly-Euclidean topology, for graph instances that are comparable in the number of vertices and edges. Clearly, small-world networks lack the topological regularity found in scientific meshes and physical networks, and current graph partitioning algorithms cannot be adapted as-is for the problem of community identification.

The parallel community identification algorithms in SNAP optimize for a measure called modularity. Let  $C = (C_1, \dots, C_k)$  denote a partition of the set of vertices  $V$  such that  $C_i \neq \emptyset$  and  $C_i \cap C_j = \emptyset$ . The cluster  $G(C_i)$  is identified with the induced subgraph  $G[C_i] := (C_i, E(C_i))$ , where  $E(C_i) := \{\langle u, v \rangle \in E : u, v \in C_i\}$ . Then,  $E(C) := \cup_{i=1}^k E(C_i)$  is the set of intra-cluster edges

and  $\tilde{E}(C) := E - E(C)$  is the set of inter-cluster edges. Let  $m(C_i)$  denote the number of inter-cluster edges in  $C_i$ . Then, the modularity measure  $q(C)$  of a clustering  $C$  is defined as

$$q(C) = \sum_i \left[ \frac{m(C_i)}{m} - \left( \frac{\sum_{v \in C_i} \deg(v)}{2m} \right)^2 \right].$$

Intuitively, modularity captures the idea that a *good division* of a network into communities is one in which the inter-community edge count is less than what is expected by random chance. This measure does not try to minimize the edge cut in isolation, nor does it explicitly favor a balanced community partitioning. The general problem of modularity optimization has been shown to be  $\mathcal{NP}$ -complete, and so there are several known heuristics to maximize modularity. Most of the known techniques fall into one of the two broad categories: *divisive* and *agglomerative* clustering. In the agglomerative scheme, each vertex initially belongs to a singleton community, and communities whose amalgamation produces an increase in the modularity score are typically merged together. The divisive approach is a top-down scheme where the entire network is initially in one community, and the network is iteratively broken down into subcommunities. This hierarchical structure of community resolution can be represented using a tree (referred to as a dendrogram). The final list of the communities is given by the leaves of the dendrogram, and internal nodes correspond to splits (joins) in the divisive (agglomerative) approaches.

SNAP includes several parallel algorithms for community identification that use agglomerative and divisive clustering schemes [2]. One divisive clustering approach is based on the greedy betweenness-based algorithm proposed by Newman and Girvan [12]. In this approach, the community structure is resolved by iteratively filtering edges with high *edge betweenness centrality*, and tracking the modularity measure as edges are removed. The compute-intensive step in the algorithm is repeated computation of the edge betweenness scores in every iteration, and SNAP performs this computation in parallel. SNAP also supports several greedy agglomerative clustering approaches, and the main strategy to exploit parallelism in these schemes is to concurrently resolve communities that do not share intercommunity edges. Performance results on real-world networks

indicate that the SNAP parallel community identification algorithms give substantial performance gains, without any loss in community structure (given by the modularity score) [2, 8].

## Related Entries

- [Chaco](#)
- [Graph Algorithms](#)
- [Graph Partitioning](#)
- [METIS and ParMETIS](#)
- [PaToH \(Partitioning Tool for Hypergraphs\)](#)
- [Social Networks](#)

## Bibliographic Notes and Further Reading

The community detection algorithms in SNAP are discussed in more detail in [3, 8]. Other popular libraries and frameworks for large-scale network analysis include Network Workbench [13], igraph [3], and the Parallel Boost Graph Library [5].

## Bibliography

1. Amaral LAN, Scala A, Barthélémy M, Stanley HE (2000) Classes of small-world networks. Proc Natl Acad Sci USA 97(21): 11149–11152
2. Bader DA, Madduri K (April 2008) SNAP: Small-world Network Analysis and Partitioning: an open-source parallel graph framework for the exploration of large-scale networks. In: Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008), IEEE, Miami, FL
3. Csárdi G, Nepusz T (2006) The igraph software package for complex network research. InterJournal Complex Systems 1695. <http://igraph.sf.net>. Accessed May 2011
4. Fortunato S (Feb 2010) Community detection in graphs. Physics Reports 486(3–5):75–174
5. Gregor D, Lumsdaine A (July 2005) The Parallel BGL: a generic library for distributed graph computations. In: Proceedings of the Parallel/High-Performance Object-Oriented Scientific Computing (POOSC ’05), IOS Press, Glasgow, UK
6. Hendrickson B, Leland R (Dec 1995) A multilevel algorithm for partitioning graphs. In: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (SC 1995), ACM/IEEE Computer Society, New York
7. Karypis G, Kumar V (1999) A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J Sci Comput 20(1):359–392
8. Madduri K (July 2008) A high-performance framework for analyzing massive complex networks. Ph.D. thesis, Georgia Institute of Technology
9. Madduri K, Bader DA (May 2009) Compact graph representations and parallel connectivity algorithms for massive dynamic

- network analysis. In: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2009), IEEE Computer Society, Rome, Italy
10. Madduri K, Bader DA, Riedy EJ (2011) SNAP: Small-world Network Analysis and Partitioning v0.4. <http://snap-graph.sf.net>. Accessed May 2011
  11. Newman MEJ (2003) The structure and function of complex networks. SIAM Rev 45(2):167–256
  12. Newman MEJ, Girvan M (2004) Finding and evaluating community structure in networks. Phys Rev E 69:026113
  13. NWB Team (2006) Network Workbench Tool. Indiana University, Northeastern University, and University of Michigan, <http://nwb.slis.indiana.edu>. Accessed May 2011
  14. Watts DJ, Strogatz SH (1998) Collective dynamics of small world networks. Nature 393:440–442

a single silicon die what was before spread on several circuits: processor, memory, bus, hardware device drivers, etc. SoC technology did not fundamentally change the functionality of the systems built, but drastically enlarged the *design space*: choosing the best way to assemble a complete system from beginning (system specification) to end (chip manufacturing) became a difficult task. In 1999, a book entitled *Surviving the SoC Revolution: A guide to platform-based design* was published. It was written by a group of people from Cadence Design System, an important computer-aided design tool company: SoC was driving a revolution of the *design methodologies* for digital systems.

The production of SoC has been driven by the emerging market of embedded systems, more precisely *embedded computing systems*: cellular phones, PDA, digital camera, etc. Embedded computing systems require strong integration, computing power and energy saving, features that were provided by SoC integration. Moreover, most of these systems required radio communication features; hence, a SoC integrates digital components and analog components: the radio subsystem. The success of mobile devices has increased the economic pressure on SoCs, design cost and time-to-market have become major issues, leading to new design methodologies such as IP-based design and hardware/software codesign.

Another consequence of integration is the exponential growth of embedded software, shifting the complexity of SoC design from hardware to software. Emulation, simulation at various level of precision, and high level design techniques are used because nowadays SoC are incredibly complex: hundreds of millions of gates and millions of lines of code. Increasing digital computing power enables software radio, providing everything everywhere transparently. SoCs are now used to provide convergence between computing, telecommunication, and multimedia. Pervasive environments will also make extensive use of embedded “intelligence” with SoCs.

## SoC (System on Chip)

TANGUY RISSET

INSA Lyon, Villeurbanne, France

### Definition

A System on Chip (SoC) refers to a single-integrated circuit (chip) composed of all the components of an electronic system. A SoC is heterogeneous, in addition to classical digital components: processor, memory, bus, etc.; it may contain analog and radio components. The SoC market has been driven by embedded computing systems: mobile phones and handheld devices.

### Discussion

#### Historical View

Gordon Moore predicted in 1965 the exponential growth of silicon integration and its consequences on the application of integrated circuits. Following this growth, known as *Moore's Law*, the number of transistor integrated on a single silicon chip has doubled every 18 months, leading to a constant growth in the semiconductor industry for over 30 years. This technological evolution implied constant changes in the design of digital circuits, with, for instance, the advent of gate level simulation and logic synthesis. Amongst these changes, the advent of System on Chip (SoC) represented a major technological shift.

The term SoC became increasingly widespread in the 1990s and is used to describe chips integrating on

#### What Is a SoC?

A system on chip or SoC is an integrated circuit composed of many components including: one or several processors, on-chip memories, hardware accelerators, devices drivers, digital/analog converters, and analog components. It was initially named SoC because all the features of a complete “system” were integrated together

on the same chip. At that time, a system was dedicated to a single application: video processing or wireless communication for instance. Thanks to the increasing role of software, SoCs are no longer specific, in fact many of them are reused in several different telephony or multimedia devices.

### Processors

The processor is the core of the SoC. Unlike desktop machine processors, a wide variety of processors is integrated in the SoC: general purpose processors, digital signal processors, micro-controllers, application specific processors, FPGA based soft cores, etc. The International Technology Road-Map for Semiconductors (ITRS) indicated that, in 2005, more than 70% of application specific integrated circuit (ASICs) contained a processor. Until 2010, most SoCs were composed of a single processor, responsible for the control of the whole system, associated with hardware accelerators and direct memory access components (DMA). In 2008, the majority of SoCs were built around a processor of one of the following form of processor architectures: ARM, MIPS, PowerPC, or x86. Multi-processor SoCs (MPSoC) are progressively appearing and making use of the available chip area to keep performance improvements. MPSoC appears to be a new major technological shift and MPSoC designers are facing problem traditionally associated with super-computing.

### Busses

The bus, or more generally the interconnection medium, is also a major component of the SoC. Many SoCs contain several busses, including a fast system bus connecting the major SoC components (processor, memory, hardware accelerator) and a slower bus for other peripherals. For instance, the Advanced Microcontroller Bus Architecture (Amba) proposes the AHB specification (Advanced High-performance Bus), and the APB (Advanced Peripheral Bus), STBus (ST-Microelectronics), and CoreConnect (IBM) are other examples of SoC busses. The use of commercial bus protocol has been a major obstacle to SoC standardization: a bus comes with a dedicated communication protocol which might be incompatible with other busses. Networks on Chip (NoC) are progressively used with MPSoC, the major problem of NoCs today is their considerable power consumption.

### Dedicated Components

A SoC will include standard control circuits such as UART for the serial port, USB controller, and control of specific devices: GPS, accelerometer, touch pad, etc. It might also include more compute intensive dedicated *intellectual properties* (IPs) usually targeted to signal or image processing: FFT, turbo decoder, H.263 coder, etc. Choosing between a dedicated component or a software implementation of the same functionality is done in early stages of the design, in the so-called *hardware/software partitioning*. A hardware IP will save power and execution time compared to a software implementation of the same functionality. Dedicated IPs are often mandatory to meet the performance requirements of the application (radio or multimedia), but they increase SoC price, complexity, and also reduce its flexibility.

### Other Components

Analog and mixed signal components may be included to provide radio connexion: audio front-end, radio to baseband interfaces, analog to digital converters. Other application domains may require very specific components such as microelectromechanical systems (MEMS), optical computing, biochips, and nanomaterials.

### Quality Criteria

The only objective indicator of the quality of a SoC is its economical success, which is obviously difficult to predict. Among the quality criteria, the most important are: power consumption, time to market, cost, and reusability.

Power consumption has been a major issue for decades. Integrating all components on the same chip reduces power consumption because inter chip wires have disappeared. On the other hand, increasing clock rate, program size, and number of computations has a negative impact on power consumption. Moreover, it is very difficult to statically predict the power consumption of a SoC as it heavily depends on low-level technological details. A precise electrical simulation of a SoC is extremely slow and cannot be used in practice for a complete system. Minimizing memory footprint and memory traffic, gating clocks for idle components, and dynamically adapting clock rate are techniques used to reduce power consumption in SoCs. In recent submicronic technologies, static power consumption is significant, leading to power consumption for idle devices.

It is usually said that, for a given product, the first commercially available SoC will capture half of the market. Hence, reducing the design time is a critical issue. Performing hardware and software design in parallel is a designer's dream: it becomes possible with virtual prototyping, that is, simulation of software on the virtual hardware. High-level design and refinement are also widely used: a good decision at a high level may save weeks of design.

The cost of a SoC is divided into two components: the nonrecurring engineering (NRE) cost which corresponds to the design cost and the unit manufacturing cost. Depending on the number of units manufactured and the implementation technology, a complex trade-off between the two can be made, bearing in mind that some technological choices can shorten the time to market such as field programmable gate arrays (FPGA) for instance. NRE cost integrates, in addition to the design cost of the hardware and software part of the SoC, the cost of computer-aided design tools, computing resource for simulation/emulation, and development of tools for the SoC itself: compilers, linkers, debuggers, simulation models of the IPs, etc. To give a very rough idea, in 2008, the development cost of a complex SoC could reach several million dollars and could need more than one hundred man years. In this context, it seems obvious that the reuse of IPs, tools, and SoCs is a major issue because it shrinks both NRE costs and time to market.

### An Example

Figure 1 shows the architecture of the Samsung S3CA400A01 SoC, designed to be associated with another chip containing processor and memory through the Cotulla interface. The Cotulla interface is associated with Xscale processor (strong ARM processor architecture manufactured by Intel). This chip was used for instance in association with the PXA250 chip of Intel in the PDA of Hewlett Packard iPaQ H5550. This SoC illustrates the different device driver components that can be found, one can also observe the hierarchy of busses mentioned before. It also shows that there is no *standard* SoC, this one has no processor inside. Thanks to the development of the open source software community, many SoC architectures have been detailed and Linux ports are available for many of them.

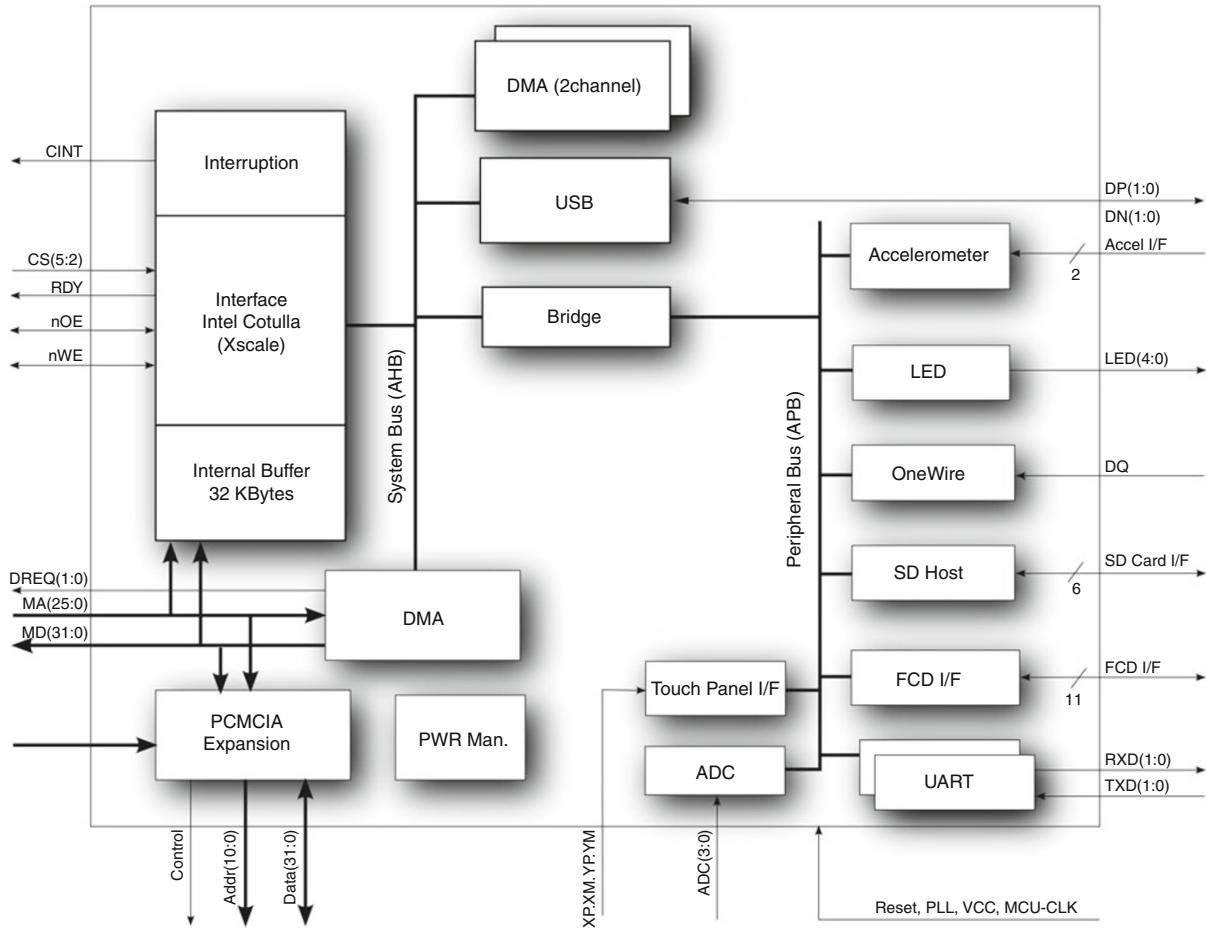
### Why Is It a SoC "Revolution"?

Although its advent was predicted, SoC really caused a revolution in the semiconductor industry because it changed design methods and brought many software design problems in the microelectronics community.

Hardware design has been impacted by the arrival of SoCs. The complexity of chip mechanically increases with the number of gates on the chip. But the design and verification effort increases more than linearly with the number of gates. This is known as the *design gap*: How can the designer efficiently use the additional gates available without breaking design time and design reliability? A SoC requires different technologies to be integrated on the same silicon die (standard cells, memory, FPGA). The clock distribution tree is a major source of power dissipation leading to the advent of globally asynchronous, locally synchronous systems (GALS). Routing wires became a real problem partially solved by using additional metal layers.

Hardware design methods have drastically changed with the constant arrival of new Electronic Design Automation (EDA) tools. RTL design methods (RTL stands for *register transfer level*) have become the standard for hardware developers replacing transistor-level circuit design. The hardware description languages VHDL and Verilog are widely used for hardware specification before synthesis. However, higher-level hardware description language are needed, partly because SoC simulation time became prohibitive, requiring huge FPGA based emulators. In addition, the networked nature of embedded applications, and the non-determinism introduced by parallelism in MPSoC greatly complicates SoC simulation. As mentioned before, cost and time to market imposes pressure on engineers productivity, leading managers to new management methods. The hardware/software nature of a SoC requires a constant dialog between hardware and software engineers.

Embedded software has been introduced in circuit design, this is, in itself, a revolution. In 2006, the Siemens company employed more software developers than Microsoft did, and since the mid-2000 s, more than half of the SoC design efforts were spent in software development. Originally composed of a very basic infinite loop waiting for interrupts, embedded software has evolved by adding many device drivers, standard or real time operating system services: tasks or



SoC (System on Chip). Fig. 1 Architecture of the Samsung S3CA400A01 SoC

threads, resource management, shared memory, communication protocols, etc. Even if an important part of embedded code is written in C, there is a trend to use component based software design methodologies that reduces the conceptual difference between hardware and software objects. Verifying software reliability is very difficult, even if an important effort is dedicated to that during the design process, complete formal verification is impossible because of the complexity of the software. This explains the bugs that everybody has experienced on their mobile devices.

Both hardware and software design methods have been drastically modified, but the most important novelty concerns the whole SoC design methodology that tries to associate in a single framework hardware and software design.

### SoC Design Methodology

There are many names associated with SoC design methodologies, all of them refer more or less to the same goal: designing hardware and software in the same framework such that the designer can quickly produce the most efficient implementation for a given system specification. The term frequently used today is *system level design*, the generic scheme of system level design is the following: (1) derive hardware components and software components from the specification, (2) map the software part on the hardware components, (3) prototype the resulting implementation, and (4) possibly provide changes at some stage of the design if the performance or cost is not satisfactory.

There is a global agreement on the fact that high-level specifications are very useful to reduce the

design time. Many design frameworks have been proposed for system level SoC design. The idea of refinement of the original specification down to hardware is present in many frameworks, high level synthesis (HLS) was once seen as the solution but appears to be a very difficult problem in general. In HLS, the hardware is completely derived from the initial specifications. During the 1990s, the arrival of Intellectual Properties (IP) introduced a clear distinction between circuit fabrication and circuit design, leading some companies to concentrate on IP design (ARM for instance). On the other hand, IP-based design and later *platform-based design* propose to start from fixed (or parameterizable) hardware library to reduce the design space exploration phase. Improvement in simulation techniques permits, with the use of SystemC language, to have an approach between the two by using virtual prototypes: cycle true simulation of complete SoC before hardware implementation.

### MPSOC and Future Trends

An open question remains at the present time: What will be the dominant model for MPSOCs that are predicted to include more than one hundred processors? There are two main trends: heterogeneous MPSOCs following actual heterogeneous SoC architecture or homogeneous SPMD-like multiprocessors on a single chip.

A heterogeneous MPSOC includes different types of processors with different instruction set architectures (ISA): general purpose processors, DSPs, application specific instruction set processors (ASIP). It also contains dedicated IPs that might not be considered as processors, DMAs, and memories. All these components are connected together through a network on chip.

Heterogeneous MPSOCs are a natural evolution of SoC, their main advantage is that they are more energy efficient than homogeneous MPSOCs. They have been driven by specific application domains such as software radio or embedded video processing. But, for the future, they suffer from many drawbacks. As they are usually tuned for a particular application, they are difficult to reuse in another context. Because of incompatibility between ISAs, tasks cannot migrate between processors inducing a much less flexible task mapping. Moreover, in the foreseen transistor technology, integrated circuits will have to be fault tolerant because of electronics defaults. Heterogeneous MPSOC are not

fault tolerant by nature. Finally, their scalability is not obvious and code reuse between different heterogeneous platforms is impossible. However, because heterogeneous MPSOCs provide better performance with lower power consumption and lower cost, there are lots of commercial activities on this model.

Homogeneous MPSOC are more flexible, they can implement well known operating system services such as code portability between different architectures, dynamic task migration, virtual shared memory. Homogeneity can help in being fault tolerant, as any processor can be replaced by another. However, with more than a hundred processor on a chip in 2020, the MPSOC architecture cannot be completely homogeneous, it has to be hierarchical: clusters of processors tightly linked, these clusters being interconnected with a network on chip. Memory must be physically distributed with non-uniform access.

Ensuring cache coherency and providing efficient compilation and parallelization tools for these architectures are real challenges. In 2009, the ITRS road-map predicted for 2013 the advent of a concurrent software compiler that will "Enable compilation and software development in highly parallel processing SoC," this will be a critical step towards the advent of homogeneous MPSOC. However, many unknowns remain concerning the programming model to use, the ability of compilation tools to efficiently use the available resources, the reusability of the chip and the reusability of the embedded code between platforms, and the real power consumption of these homogeneous multiprocessors on chip.

A trade-off would be a homogeneous SoC with some level of heterogeneity: dedicated IPs for domain-specific treatments. New technological techniques, such as 3D VLSI packaging technology or more generally the arrival of so-called *system in package*, mixing various nanotechnologies on a single chip might also open a new road to embedded computing systems. But, whatever will be the successful MPSOC model, it will bring highly parallel computing on a chip and should lead to a revival of parallel computing engineering.

### Bibliographic Notes and Further Reading

A number of books have been published on SoC, we have already talked about "surviving the SOC

revolution” [1]. Many interesting ideas were also present in the Polis project [4]. Daniel Gajski [9] worked on SoC from the beginning. Wayne Wolf published a general presentation [13]. More recently, Chris Rowen [11] proposed an interesting view of SoC design.

A good overview of the status of high-level synthesis can be found in Coussy and Morawiec book [2]. More practical details can be found on the Steve Furber book on ARM [8].

A number of useful information are present on the web, from the International Technology Roadmap for Semiconductors (ITRS [7]), from analyst or engineer [3] or open source software developer for embedded devices [5, 6]. A simple introduction to semiconductor industry can be found in Jim Turley’s book [12].

A good introduction on MPSoC is presented in [10], but most of the interesting work on this subject are presented in the proceedings of the international conferences on the subject: Design Automation Conference (DAC), Design automation and Test in Europe (DATE), International Forum on Embedded MPSoC and Multicore (MPSOC), etc.

## Bibliography

1. Chang H, Cooke L, Hunt M, Martin G, McNelly AJ, Todd L (1999) Surviving the SOC revolution: a guide to platform-based design. Kluwer Academic Publishers, Norwell, MA
2. Coussy P, Morawiec A (eds) (2008) High-level synthesis: from algorithm to digital circuit. Springer, Berlin, Germany
3. Embedded System Design (2009) <http://www.embedded.com/>
4. Balarin F et al (1997) Hardware-software co-design of embedded systems: the POLIS approach. Kluwer Academic Press, Dordrecht
5. Linux for device (2009) <http://www.linuxfordevices.com/>
6. Open Embedded: framework for embedded Linux (2009) <http://wiki.openembedded.net>
7. International Technology Roadmap for Semiconductors (2009) <http://www.itrs.net/>
8. Furber S (2000) ARM system-on-chip architecture. Addison Wesley, Boston, MA
9. Gajski DD, Abdi S, Gerstlauer A, Schirner G (2009) Embedded system design: modeling, synthesis and verification. Kluwer Academic Publishers, Dordrecht
10. Jerraya A, Wolf W (2004) Multiprocessor systems-on-chips (The Morgan Kaufmann Series in Systems on Silicon). Morgan Kaufmann, San Francisco, CA
11. Rowen C (2004) Engineering the complex SOC: fast, flexible design with configurable processors. Prentice-Hall Press, Upper Saddle River, NJ
12. Turley J (2002) The essential guide to semiconductors. Prentice Hall Press, Upper Saddle River, NJ
13. Wolf W (2002) Modern VLSI design: system-on-chip design. Prentice Hall Press, Upper Saddle River, NJ

## Social Networks

MALEQ KHAN, V. S. ANIL KUMAR, MADHA V. MARATHE,  
PAULA E. STRETZ  
Virginia Tech, Blacksburg, VA, USA

## Introduction

Networks are pervasive in today’s world. They provide appropriate representations for systems comprised of individual agents interacting locally. Examples of such systems are urban regional transportation systems, national electrical power markets and grids, the Internet, ad-hoc communication and computing systems, public health, etc. According to Wikipedia, A *social network* is a social structure made of individuals (or organizations) called “nodes,” which are tied (connected) by one or more specific types of interdependency, such as friendship, kinship, financial exchange, dislike, sexual relationships, or relationships of beliefs, knowledge or prestige. Formally, a social network induced by a set  $V$  of agents is a graph  $G = (V, E)$ , with an edge  $e = (u, v) \in E$  between individuals  $u$  and  $v$ , if they are interrelated or interdependent. Here, we use a more general definition of nodes and edges (that represent interdependency). The nodes represent living or virtual individuals. The edges can represent information flow, physical proximity, or any feature that induces a potential interaction. For example, the social contact networks, an edge signifies some form of physical co-location between the individuals – such a contact may either capture physical proximity, an explicit physical contact or a co-location in the virtual world, e.g., Facebook. Socio-technical networks are a generalization of social networks and consist of a large number of interacting physical, technological, and human/societal agents. The links in socio-technical networks can be physically real or a matter of convention such as those imposed by law or social norms, depending on the specific system being represented. This entry primarily deals with social networks.

Social networks have been studied for at least 100 years; see [9] for a detailed account of the history and development of social networks and the analytical tools that followed them. Scientists have used social networks to uncover interesting insights related to societies and social interactions. Social networks, their structural analysis, and dynamics on these networks (e.g., spread of diseases over social contact networks) are now a key part of social, economic, and behavioral sciences. Importantly, these concepts and tools are also becoming popular in other scientific disciplines such as public health epidemiology, ecology, and computer science due to their potential applications. For example, in computer science, interest in social networks has been spurred by web search and other online communication and information applications. Google and other search engines crucially use the structure of the web graph. Social networking sites, such as Facebook and Twitter, and blogging sites have grown at an amazing rate and play a crucial role in advertising and the spread of fads, fashion, public opinion, and politics. The growing importance of social networks in the scientific community can be gauged by a report from the National Academies [18] as well as a recent issue of Science [21]. The proliferation of Facebook, MySpace, Twitter, Blogosphere, and other online communities has made social networking a pervasive and essential part of our lingua.

An important and almost universal observation is that the network structure seems to have a significant impact on these systems and their associated dynamics. For instance, many infrastructure networks (such as the Internet) have been observed to be highly vulnerable to targeted attacks, but are much more robust to random attacks [2]. In contrast, social contact networks are known to be very robust to all types of attacks [7]. These robustness properties have significant implications for control and policies on such systems - for instance, protecting high degree nodes in Internet-like graphs has been found to be effective in stopping the spread of worms [19]. For many processes, e.g., the SIS model of diffusion (which models “endemic” diseases, such as Malaria and Internet malware), the dynamics have been found to be characterized quite effectively by the spectral properties of the underlying network [19]. Similarly, classical and new graph theoretic metrics, e.g., centrality (which, informally, determines the fraction of shortest paths passing through a node), have been found to be

useful in identifying “important” nodes in networks and community structure. Therefore, computing the properties of these graphs, and simulating the dynamical processes defined on them are important research areas.

Most networks that arise in practice are very large and heterogeneous, and cannot be easily stored in memory - for instance, the social contact graph of a city could have millions of nodes and edges [13], while the web graph has over 11 billion nodes [7] and several billion edges. Additionally, these networks change at very short time scales. They are also naturally labeled structures, which adds to the memory requirements. Consequently, simple and well-understood problems such as finding shortest paths, which have almost linear time algorithms, become challenging to solve on such graphs. High-performance computing has been successfully used in a number of scientific applications involving large data sets with complex processes, and is, therefore, a natural and necessary approach to overcome such resource limitations.

Most of the successful applications of high-performance computing have been in physical sciences, such as molecular dynamics, radiation transport,  $n$ -body dynamics, and sequence analysis, and researchers routinely solve very large problems using massive distributed systems. Many techniques and general principles have been developed, which have made parallel computing a crucial tool in these applications. Social networks present fundamentally new challenges for parallel computing, since they have very different structure, and do not fit the paradigms that have been developed for other applications. As we discuss later, some of the key issues that arise in parallelization of social network problems include highly irregular structure, poor locality, variable granularity, and data-driven computations. Most social contact networks have a scale-free and “small world” structure (described formally below), which is very different from regular structures such as grids. As a result, social network problems often cannot be decomposed easily into smaller independent subproblems with low communication, making traditional parallel computing techniques very inefficient.

The goal of this article is to highlight the challenges for high-performance computing posed by the growing area of social networks. New paradigms are needed at all levels of hardware architecture, software methodologies, and algorithmic optimization.

Section “Background and Notation” presents some background into social networks, their history, and important problems. Section “Petascale Computing Challenges for Social Network Problems” identifies the main challenges for parallel computing. We discuss some recent developments in Section “Techniques” and conclude in section “Conclusions”.

## Background and Notation

Many of the results in sociology that have now become folklore have been based on insights from social network analysis. One of such classic results is the “six-degree of separation” experiment, in which Stanley Milgram [22] asked 296 randomly chosen people to forward a letter to a stock broker in Boston, by sending it to one of their acquaintances, who is most likely to know the target. Milgram found that the median length of chains that successfully reached the target was about 6, suggesting that the global social contact network was very highly connected. Attempts to explain this phenomenon have led to the “small-world” graph models, which show that a small fraction of long-range contacts to individuals spatially located far away is adequate to bring the diameter of the network down. Another classic result is Granovetter’s concept of the “strength of weak ties” [13], which has become one of the most influential papers in sociology. He examined the role of information that individuals obtain as a result of their position in the social contact network, and found that information from acquaintances (weak ties), and not close friends, was the most useful, during major lifestyle changes, e.g., finding jobs. This finding has led to the notions of “homophily” (i.e., nodes with similar attributes form contacts) and “triadic closure” (i.e., nodes with a common neighbor are more likely to form a link), and the idea that weak ties form bridges that span different parts of the network.

Concepts such as these have been refined and formalized using graph theoretic measures; see [5, 19] for a formal discussion. Some of the key graph measures are: (1) (*Weighted*) *Degree distribution*: The degree of a node is the number of contacts it has, and the degree distribution is the frequency distribution of degrees. This measure is commonly used to characterize and distinguish various families of graphs. In particular, it has been found that many real networks have power law or lognormal degree distributions, instead of Poisson,

which has been used to develop models to explain the construction and evolution of these networks. (2) The *Clustering coefficient*,  $C(v)$ , of a node  $v$  is the probability that two randomly picked neighbors of node  $v$  are connected, and models the notion of triadic closure, and its extensions. (3) *Centrality*: Let  $f(s, t)$  denote the number of shortest paths from  $s$  to  $t$ , and let  $f_v(s, t)$  denote the number of shortest paths from  $s$  to  $t$  that pass through node  $v$ . The centrality of node  $v$  is defined as  $bc(v) = \sum_{s,t} f_v(s, t)/f(s, t)$ , and captures its “importance”. (4) *Robustness to node and edge deletions*, which is quantified in terms of the giant component size, as a result of node and edge deletions, and has been found to be a useful measure to compare graphs. (5) *Page Rank and Hubs* have been used to identify “important” web pages for search queries, but have been extended to other applications. The Page Rank of a node is, informally, related to the stationary probability of a node in a random walk model of a web surfer. (6) A *community* or a *cluster* is a “loosely connected” set of nodes with similar attributes, which are different from nodes in other communities. There are many different methods for identifying communities, including modularity and spectral structure [19, 22].

In applications such as epidemiology and viral marketing, the network is usually associated with a diffusion process, e.g., the spread of disease or fads. There are many models of diffusion, and some of the most widely used classes of models are *Stochastic cascade models* [11, 19]. In these models, we are given a probability  $p(e)$  for each edge  $e \in E$  in a graph  $G = (V, E)$ . The process starts at a node  $s$  initially. If node  $v$  becomes active at time  $t$ , it activates each neighbor  $w$  with probability  $p(v, w)$ , independently (and also independent of the history); no node can be reactivated in this process. In the case of viral marketing, active nodes are those that adopt a certain product, while in the SIR model of epidemics, the process corresponds to the spread of a disease, and the active nodes are the ones that get infected. Another class of models that has been studied extensively involves *Threshold functions*; one of the earliest uses of threshold models was by Granovetter and Schelling [10], who used it for modeling segregation. The *Linear Threshold Model* [14] is at the core of many of these models. In this model, each node  $v$  has a threshold  $\Theta_v$  (which may be chosen randomly), and each edge  $(v, w)$  has a weight  $b_{v,w}$ . A node  $v$  becomes active at time  $t$ , if  $\sum_{w \in N(v) \cap A_t} b_{v,w} \geq \Theta_v$ ,

where  $N(v)$  denotes the set of neighbors of  $v$  and  $A_t$  denotes the set of active nodes at time  $t$  (in this model, an active node remains active throughout). Such diffusion processes are instances of more general models of dynamical systems, called *Sequential Dynamical Systems* (SDSs) [7, 19], which generalize other models, such as cellular automata, Hopfield networks, and communicating finite state machines. An SDS  $S$  is defined as a tuple  $(G, F, \pi)$ , where: (a)  $G = (V, E)$  is the underlying graph on a set  $V$  of nodes, (b)  $F = (f_v)$  denotes a set of local functions for each node  $v \in V$ , on some fixed domain. Each node  $v$  computes its state by applying the local function  $f_v$  on the states of its neighbors. (c)  $\pi$  denotes a permutation (or a word) on  $V$ , and specifies the order in which the node states are to be updated by applying the local functions. One update of the SDS involves applying the local functions in the order specified by  $\pi$ . It is easy to see that the above diffusion models can be captured by suitable choice of the local functions  $f_v$  and the order  $\pi$ . It is easy to extend the basic definition of SDS to accommodate local functions that are stochastic, the edges that are dynamic, and the graph that is represented hierarchically (to capture organizational structure).

Fundamental questions in social networks include (1) understanding the structure of the networks and the associated dynamics, especially how the dynamics is affected by the network properties; (2) techniques to control the dynamics; (3) identifying the most critical and vulnerable nodes crucial for the dynamics; and (4) coevolution between the network and dynamics – this issue brings in behavioral changes by individuals as a result of the diffusion process. The above mentioned problems require large-scale simulations, and parallel computing is a natural approach for designing them.

## Petascale Computing Challenges for Social Network Problems

Lumsdaine et al. [15] justifiably assert that traditional parallel computing techniques (e.g., those developed in the context of applications such as molecular dynamics and sequencing) are not well suited for large-scale social network problems because of the following reasons: (1) Graph algorithms chiefly involve data driven computations, in which the computations are guided

by the graph structure. Therefore, the structure and sequence of computations are not known in advance and are hard to predict, making parallelization difficult. (2) Social networks are typically very irregular and strongly connected, which makes partitioning into “independent” sub-problems difficult [15]. As observed by many researchers, these networks usually have a small-world structure with high clustering, and such graphs have low diameter and large separators (unlike regular graphs such as grids, which commonly arise in other applications). (3) Locality is one of the key properties that helps in parallelization; however, computations on social networks tend to give low locality because of the irregular structure, leading to computations and data access patterns with global properties. (4) As noted in [15, 16], graph computations often involve exploration of the graph structure, which are highly memory intensive and there is very little computation to hide the latency to memory accesses. Thus, the performance of memory subsystem, rather than the memory clock frequency, dominates the performance of graph algorithms.

Traditional parallel architectures have been developed for applications that do not have these constraints, and therefore, are not completely suited for social network algorithms. The most common paradigm of distributed computing, the distributed memory architectures with message-passing interface (MPI), leads to high message passing, because of the lack of locality and high data access to computation ratio. The shared memory model is much more suited for the kinds of data-driven computations that arise in graph algorithms, and multithreaded approaches have been found to be more effective in exploiting the parallelism that arises. Some of the main hardware and software challenges that arise are as follows: (1) In many graph problems, e.g., shortest paths, parallelism can be found at a fairly fine level of granularity, though in other problems, e.g., computing centrality, there is coarse grained granularity, where each path computation is an independent task. (2) While multithreading is crucial for graph algorithms, the unstructured nature of these graphs implies that there are significant memory contention and cache-coherence problems. (3) It is difficult to achieve load balancing on graph computations, where the level of parallelism varies with time, e.g., as in breadth-first search.

## Techniques

Parallel algorithms for social network problems is an active area of research and we now discuss some of the key techniques that have been found to be effective for some problems that arise in analyzing social networks. These techniques are still application specific, and developing general paradigms is still an active research area.

### Massive Multithreading Techniques

In [12], Hendrickson and Berry discussed how massively multithreaded architectures, such as the Cray MTA-2 and its successor the XMT, can be used to boost the performance of parallel graph algorithms. Instead of trying to reduce latency for single-memory access, the MTA-2 tolerates it by ensuring that the processor has other work to do while waiting for a memory request to be satisfied by supporting many concurrent threads in a single processor and switching between them in a single clock cycle. When a thread issues memory request, the processor immediately switches to another ready-to-execute thread. MTA-2 supports fast and dynamic thread creation and destruction allowing the program to dynamically determine the number of threads based on the data to be processed. Support of virtualization of threads allows adaptive parallelism and dynamic load balancing. MTA-2 also supports word-level locking of the data items, which decreases access contention and minimizes impact on the execution of other threads.

Drawbacks of massively multithreaded machines include higher price and much slower clock rate than mainstream systems, and the difficulty in porting to other architectures. Hendrickson and Berry [12], using the massively multithreaded architectures, extended a small subset of the Boost Graph Library (BGL) into the Multithreaded Graph Library (MTGL). This library, which is their ongoing current work, retains the BGL's look and feel, yet encapsulates the use of nonstandard features of massively multithreaded architectures. Madduri et al. [16] also discussed architectural features of the massively multithreaded Cray XMT system and present implementation details and optimization of betweenness centrality computations. They further showed how the parallel algorithm for betweenness centrality can be modified so that locking is not necessary

to avoid concurrent access to a memory unit, which they call lock-free parallel algorithm.

### Distributed Streaming Algorithms

Data streaming is a useful approach to deal with memory constraints, in which processors keep a “sketch” of the data, while making one or more passes through the entire data (or stream), which is assumed to be too big to store. Streaming is usually sequential, and Google's MapReduce [6] and Apache's Hadoop [1] provide a generic programming framework for distributed streaming. MapReduce/Hadoop provide a transparent implementation which takes care of issues like data distribution, synchronization, load balancing, and processor failures. It can, thus, greatly simplify computation over large-scale data sets, and has been proven to be a useful abstraction for solving many problems, especially those related to web applications.

We briefly describe the MapReduce framework here. In the map step, the master node takes a task, partitions it into smaller *independent subtasks* and passes them down to the worker nodes. The worker nodes may recursively do the same, or process the subtask and pass it back to the master node. The master node then collects the solutions to each subtask that it had assigned and processes these solutions to obtain the final solution for the original task. More formally, the input to the MapReduce system is represented as a set of (key, value) pairs, which can be defined in a completely general manner. There are two functions: *Map* and *Reduce*, which need to be specified by the user. A Map function processes a key/value pair  $(k_1, v_1)$  and produces a list of zero or more intermediate key/value pairs  $(k_2, v_2)$ . Once the Map function completes processing of the input key/value pairs, the system groups the intermediate pairs by each key  $k_2$  and makes a list of all values associated with  $k_2$ . These lists are then provided as input to the Reduce function. The Reduce function is then applied independently to each key/value-list pair to obtain a list of final values. The whole sequence can be repeated as needed.

Programs written in this functional style are automatically parallelized and executed on a large cluster of machines, insulating users from all the run-time issues. The MapReduce framework is quite powerful, and can be used to efficiently compute any symmetric

order-invariant function [8], a large subclass of problems that can be solved by streaming algorithms. This class includes many stream statistics that arise in web applications. Kang et al. [13] use this approach to estimate the diameter of a massive instance of the web graph with over 10 billion edges. The (key, value) pairs in their algorithm capture adjacencies of nodes and estimates of the number of nodes within a  $d$ -neighborhood of a node; the algorithm is run in multiple stages in order to keep the keys and associated values simple and small, and also uses efficient sketches for representing set unions. The uniqueness of a MapReduce framework lies in its ability to hide the underlying computing architecture from the end-user. Thus, the user can compute using a MapReduce framework on parallel clusters as well as loosely coupled machines in a cloud or over volunteer computing resources comprised of independent, heterogeneous machines.

### Dynamical Processes on Social Networks

So far, much of the discussion has focused on computational considerations related to social network structure. But many more interesting questions arise when one studies dynamical processes on these networks; in fact, it is fair to say that social networks exist to serve the function of one or more dynamical processes on them. While researchers have worked extensively on structural properties of social networks, the literature on the study of dynamical processes on social networks is relatively sparse. When it does exist, it is usually in the context of very small networks or stylized and regular networks. Unfortunately, many of these results do not scale or apply to large and realistic social networks. In [3], the authors discuss new algorithms and their implementations for modeling certain classes of dynamical processes on large social networks. In general, the problem is *hard* computationally; it becomes even harder when the social network structure, the dynamical processes and individual node behavior co-evolve. For certain class of dynamical processes that capture a number of interesting social, economic, and behavioral theories of collective behavior, it is possible to develop fast parallel algorithms. Intuitively, such dynamical processes can be expressed as SDSs with a certain symmetry condition imposed on local transition functions.

## Conclusions

As our society is becoming more connected, there is an increasing need to develop innovative computational tools to synthesize, analyze, and reason about large and progressively realistic social networks. Applications of social networks for analyzing real-world problems will require the study of multi-network, multi-theory systems – systems composed of multiple networks among agents whose behavior is governed by multiple behaviors. Advances in computing and information are also giving rise to new classes of social networks. Two examples of these networks are: (1) botnet networks in which individual nodes are software agents (bots) and (2) wireless-social networks in which social networks between individuals are being supported by wireless devices that allow them to communicate and interact anytime and anywhere. As the society becomes more connected, these applications require support for real-time individualized decision making over progressively larger evolving social networks. This same technology will form the basis of new modeling and data processing environments. These environments will allow us to leverage the next generation computing and communication resources, including cloud computing, massively parallel peta-scale machines and pervasive wireless sensor networks. The dynamic models will generate new synthetic data sets that cannot be created in any other way (e.g., direct measurement). This will enable social scientists to investigate entirely new research questions about functioning societal infrastructures and the individuals interacting with these systems. Together, these advances also allow scientists, policy makers, educators, planners, and emergency responders unprecedented opportunities for multi-perspective reasoning.

## Acknowledgments

This work has been partially supported by NSF Grant CNS-0626964, SES-0729441, CNS-0831633, and OCI-0904844, and DTRA Grant HDTRA1-0901-0017 and HDTRA1-07-C-0113.

## Bibliography

- Apache hadoop. Code and documentation are available at <http://developer.yahoo.com/hadoop/>
- Barabasi A, Albert R (1999) Emergence of scaling in random networks. *Science* 286:509–512

3. Barrett C, Bisset K, Marathe A, Marathe M (2009) An integrated modeling environment to study the co-evolution of networks, individual behavior and epidemics. *AI Magazine*, 2009
4. Barrett C, Eubank S, Marathe M (2005) Modeling and simulation of large biological, information and socio-technical systems: an interaction based approach interactive computation. In: Goldin D, Smolka S, Wegner P (2005) *The new paradigm*, Springer, Berlin, Heidelberg, New York
5. Baur M, Brandes U, Lerner J, Wagner D (2009), Group-level analysis and visualization of social networks. In: Lerner J, Wagner D, Zweig K (eds) *Algorithmics of large and complex networks*, vol 5515, LNCS. Springer, Berlin, Heidelberg, New York, pp 330–358
6. Dean J, Ghemawat S (2004) Mapreduce: Simplified data processing on large clusters. In: Proceedings of the sixth symposium on operating system design and implementation (OSDI), San Francisco, December 2004
7. Eubank S, Guclu H, Anil Kumar VS, Marathe MV, Srinivasan A, Toroczkai Z, Wang N (2004) Modelling disease outbreaks in realistic urban social networks. *Nature* 429(6998):180–184
8. Feldman J, Muthukrishnan S, Sidiropoulos A, Stein C, Svitkina Z (2008) On distributing symmetric streaming computations. In: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms (SODA), San Francisco, pp 710–719
9. Freeman L (2004) The development of social network analysis: a study in the sociology of science. Empirical Press, Vancouver
10. Granovetter M (1978) Threshold models of collective behavior. *Am J Sociolgy* 83(6):1420–1443
11. Grimmett G (1999) *Percolation*. Springer, New York
12. Hendrickson B, Berry J (2008) Graph analysis with high-performance computing. *Comput Sci Eng* 10:14–19
13. Kang U, Tsourakakis CE, Appel AP, Faloutsos C, Leskovec J (2008) Hadi: fast diameter estimation and mining in massive graphs with hadoop. Technical Report CMU-ML-08-117, Carnegie Mellon University
14. Kempe D, Kleinberg JM, Tardos E (2003) Maximizing the spread of influence through a social network. In: SIGKDD '03, Washington
15. Lumsdaine A, Gregor D, Hendrickson B, Berry J (2007) Challenges in parallel graph processing. *Parallel Process Lett* 17:5–20
16. Madduri K, Ediger D, Jiang K, Bader DA, Chavarra-Miranda DG (2009) A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In: Proceedings of the 3rd Workshop on Multi-threaded Architectures and Applications (MTAAP), Miami, May 2009
17. Mortveit HS, Reidys CM (2000) Discrete sequential dynamical systems. *Discrete Math* 226:281–295
18. National Research Council of the National Academies (2005) *Network Science*. The National Academies Press, Washington, DC
19. Newman M (2003) The structure and function of complex networks. *SIAM Rev* 45:167–256
20. Newman MEJ (2004) Detecting community structure in networks. *European Phy J B*, 38:321–330
21. Special issue on complex systems and networks, *Science*, 24 July 2009, 325 (5939):357–504
22. Travers J, Milgram S (1969) An experimental study of the small world problem. *Sociometry* 32(4):425–443

## Software Autotuning

### ► Autotuning

## Software Distributed Shared Memory

SANDHYA DWARKADAS

University of Rochester, Rochester, NY, USA

### Synonyms

[Implementations of shared memory in software](#); [Shared virtual memory](#); [Virtual shared memory](#)

### Definition

Software distributed shared memory (SDSM) refers to the implementation of shared memory in software on systems that do not provide hardware support for data coherence and consistency across nodes (and the memory therein).

### Discussion

#### Introduction

Executing an application in parallel requires the coordination of computation and the communication of data to respect dependences among tasks. Historically, multiprocessor machines provide either a shared memory or a message passing model for data communication in hardware (with models such as partitioned global address spaces that fall in between the two extremes). In the message passing model, communication and coordination across processes is achieved via explicit messages. In the shared memory model, communication and coordination is achieved by directly reading and writing memory that is accessible by multiple processes and mapped into their address space. Either model can be emulated in software if the hardware does not

directly implement it. This entry focuses on implementations of shared memory in software on machines that do not support shared memory in hardware across all nodes.

In the shared memory model, data communication is implicitly performed when shared data is accessed. Programmers (or compilers) must still ensure, however, that shared data users synchronize (coordinate) with each other in order to respect program dependencies. The message passing model typically requires explicit communication management by the programmer or compiler – software must specify *what* data to communicate, with *whom* to communicate the data, and *when* the communication must be performed (typically on both the sender and the receiver). Coordination is often implicit in the data communication. While parallel programming has proven inherently more difficult than sequential programming, shared memory is considered conceptually easier as a parallel programming model than message passing. In contrast to message passing, the shared memory model hides the need for explicit data communication from the user, thereby avoiding the need to answer the “*what, when, and whom*” questions.

In terms of implementation, typical shared memory-based hardware requires support for coherence – ensuring that any modifications made by a processor propagate to all copies of the data — and consistency – ensuring a well-defined ordering in terms of when and in what order modifications are visible to other processors. Such support and the need for access to the physically shared memory impacts the scalability of the design. In contrast, message-based hardware is typically easier to build and to scale to large numbers of processors.

Software distributed shared memory (SDSM) attempts to combine the conceptual appeal of shared memory with the scalability of distributed message-based communication by allowing shared memory programs to run on message-based machines. SDSM systems target the knee of the price-performance curve by using commodity components as the nodes in a network of machines. SDSM provides processes with the illusion of shared address spaces on machines that do not physically share memory. Kai Li’s [27, 28] pioneering work in the mid-1980s was the first to consider

implementing SDSM on a network of workstations. Since then, there has been a tremendous amount of work exploring the state space of possible implementations and programming models, targeted at reducing the software overheads required and hiding the larger communication overheads of message-based hardware.

## Implementation Issues

In order to implement coherence, accesses to shared data must be detected and controlled. Implementing coherence in software can typically be accomplished either using virtual memory hardware [8, 23, 28], using instrumentation [34], or using language-level hooks [3, 18]. The former two approaches assume that memory is a linear untyped array of bytes. The latter approach takes advantage of object- and type-specific information to optimize the protocol. The trade-offs are discussed below.

## Implementations Using Virtual Memory

The virtual memory mechanisms available on general-purpose processors may be used to implement coherence [8, 23, 28]. The virtual memory framework combines protection with address translation at a granularity of a page, which is typically on the order of a few (e.g., 4) KBytes on today’s machines. Hardware protection modes can be exercised to ensure that read or write accesses to pages that are shared are trapped if protection is violated. Software handlers installed by the runtime can then determine the necessary actions required to maintain coherence. The advantage of the virtual memory approach is that there is no overhead for data that is private or already cached. The disadvantage is the large sharing granularity, which can result in data being falsely (i.e., different processors reading and writing to independent locations that are colocated on the same page) shared, and the large cost of handling a miss.

## Implementations Using Instrumentation

An alternative approach is to instrument the program in order to monitor every load and store operation to shared data. The advantage of this approach is that coherence can be maintained at any desired granularity [35], resulting in lower miss penalties. The disadvantage is that overhead is incurred regardless of whether

data is actually shared, resulting in higher hit latency. Optimizations that identify and instrument accesses to only shared data and that hide the instrumentation behind existing program computation [34] help reduce this overhead. Alternatively, hardware support, such as ECC (error-correcting code) bits in memory [35], can be leveraged to eliminate the software instrumentation by using the ECC bits to raise an exception when the data is not valid. However, complications do arise if these exceptions are not precise.

### Language-Level Implementations

Language-level programming systems (e.g., [3, 18]) provide the opportunity to use objects and types to improve the efficiency of recognizing accesses to shared data. Coherence is usually maintained at the granularity of an object, providing application developers with control over how data is managed. The DOSA system [18] shows how a handle-based implementation (an implementation in which all object references are redirected through a handle) enables efficient fine- and coarse-grain data sharing. Handle-based implementations work transparently when used in conjunction with safe languages (ones in which no pointer arithmetic is allowed). The handles make it easy to relocate objects in memory, making it possible to use virtual memory techniques for access and modification detection and control. They also interact well with the garbage collection algorithms used in these type-safe languages, allowing good performance for these systems. The redirection allows the implementation to avoid false sharing for fine-grained access patterns.

### Single- Versus Multiple-Writer Protocols

Traditional hardware-based cache coherence usually maintains the coherence invariant of allowing only a single writer for each coherence granularity unit of data. This approach has the advantage of simplicity, works well under the intuitive assumption that two processes will not intentionally write to the same location concurrently without some form of coordination, and makes it easy to ensure write serialization. However, coherence granularities are often larger than a single word. Even with hardware coherence, it is possible for a cache line to bounce (“ping-pong”) between caches because processes modify logically disjoint regions of

the same cache line. This problem is further exacerbated by the larger coherence granularities of software coherence, particularly when using a virtual memory implementation.

In order to combat the resulting performance penalties caused by such false sharing, multiple-writer protocols have been proposed [8]. In these protocols, the single writer state invariant of traditional hardware coherence no longer holds. At any given time, multiple processes may be writing to the same coherence unit. Write serialization is ensured by making sure that modifications (determined by keeping track of exactly what locations in the coherence unit were written) are propagated to all valid copies or that these copies are invalidated. Modified addresses can be determined as in the previous section, either by using instrumentation or virtual memory techniques. In the latter case, a *twin* or copy of the virtual memory page is made prior to modification, which is subsequently used to compare to the modified page in order to create a *diff* or an encoding of only the data that has changed on the page. Multiple-writer protocols work particularly well when combined with a more relaxed memory consistency model that allows coherence actions to be delayed to program synchronization points (elaborated on in the next section).

### Memory Models

In order to help hide long communication latencies, SDSM systems typically use some form of relaxed consistency model, which enable aggregation of coherence traffic. In release consistency [16], ordinary accesses to shared data are distinguished from synchronization. Synchronization is further split into acquires and releases. An acquire roughly corresponds to a request for access to data, such as a lock operation. A release roughly corresponds to making such data available, such as an unlock operation. SDSM systems leverage the release consistency model to delay coherence actions on writes until the point of a release [8]. This enables both aggregation of coherence messages and a reduction in the false sharing penalty. The former is beneficial because of the high per message costs on typical network-based platforms. The latter is a result of data “ping-ponging” between sharers due to the fact that logically differentiated data resides in the same coherence

unit. By delaying coherence actions to the release point, the number of such “ping-pongs” is reduced.

Release consistency mandates, however, that before the release is visible to *any* other process, all ordinary data accesses made prior to the release are visible at *all* other processes. This results in extra messages between processes at the time of a release. The TreadMarks system [2] proposed the use of a lazy release consistency model [22] by making the observation that for data race free [1] programs, such ordinary data accesses need only be visible at the time of an acquire. Communication can thus be further aggregated and limited to between an acquirer and a releaser.

While lazy release consistency reduces the number of coherence messages used to the minimum possible, it comes at a cost in terms of implementation complexity. Since data accesses are not made visible everywhere at the time of a release, the runtime system must keep track of the ordering of such coherence events and transmit this information when a process eventually performs an acquire. TreadMarks accomplishes this via the use of vector timestamps. The execution of each process is divided into intervals delineated by synchronization operations (either an acquire or a release). Each interval is assigned a monotonically increasing number. Intervals of different processes are partially ordered: Intervals on a single process are totally ordered by program order and intervals on different processes are ordered by acquire–release causality. This partial order is what is captured by assigning a vector timestamp to each interval. Write notices for all data written in an interval are associated with its vector timestamp. This information is piggybacked on release messages to ensure coherence. Each process must then ensure that write notices from all intervals that precede its current interval are applied prior to execution of the current interval.

On system area networks such as Infiniband [21], where the latency and CPU occupancy of communication is an order of magnitude lower than on traditional networks, it can sometimes be beneficial to overlap communication with computation. Cashmere [25] leverages this observation in a moderately lazy release consistent implementation on the Memory Channel [16] network. Write notices are pushed to all processes at the time of a release. They are only applied at the time of the next acquire, thereby avoiding the need to interrupt the remote process. Another

advantage of this implementation is that it removes the need to maintain vector timestamps along with the associated metadata management complexity.

An alternative consistency model, *entry consistency*, was defined by Bershad and Zekauskas [5]. All shared data are required to be explicitly associated with some synchronization (lock) variable. On a lock acquisition, only the shared data associated with the lock is guaranteed to be made coherent. This model has some implementation and efficiency advantages. However, the cost is a change to the programming model, requiring explicit association of shared data with synchronization and additional synchronization beyond what is usually required for data race free programs. Scope consistency [20] provides a model similar to entry consistency, but helps eliminate the burden of explicit binding by implicitly associating data with the locks under which they are modified.

## Data and Metadata Location

In order to get an up-to-date copy of data, the runtime system must determine the location of the latest version/s. Most implementations fall into two categories – ones in which the information is distributed across all nodes at the time of synchronization, and ones in which each coherence and metadata unit has a *home* that keeps track of where the latest version resides, similar to a hardware directory-based protocol. The former implies that every process knows where the latest version is or has the latest version of the data propagated directly to them. The latter implies a level of indirection (through the home) in order to locate the latest version of the data or owner of the metadata. Variations of the latter protocol include making sure that the home node is kept up to date so that any process requesting a copy can retrieve it directly from the home, and allowing migration of the home to (one of) the most active user/s.

S

## Leveraging Hardware Coherence

As multiprocessor desktop machines become more common, and with the advent of multicore chips, a cluster of multicore multiprocessor machines is a fairly common computing platform. In implementing shared memory on these platforms, the challenge is to take advantage of hardware-based sharing whenever possible and ensure that software overhead is incurred *only* when actively sharing data across nodes in a cluster.

SoftFLASH [12] is a kernel-level implementation of a two-level coherence protocol on a cluster of symmetric multiprocessors (SMPs). SoftFLASH implements coherence using virtual memory techniques, which implies that coherence is implemented by changing read/write permissions in the process's page table. These permissions are normally cached in the translation lookaside buffer (TLB) in each of the processors of a single node. Since the TLB is not typically hardware coherent, in order to ensure that all threads/processes on a single node have appropriate levels of permission to access shared data, the TLBs in each processor within a node must be examined and flushed if necessary. This process, called TLB shutdown, is usually accomplished with costly inter-processor interrupts.

Cashmere-2L [36] avoids the need for these expensive TLB shutdowns through a combination of a relaxed memory model and the use of a novel two-way diffing technique. Leveraging the observation that in a data race free model, accesses by different processes between two synchronization points will be to different memory locations, Cashmere-2L avoids the need to interrupt other processes during a software coherence operation. Updates to a page are applied through a reverse diffing process that identifies and updates only the bytes that have changed, allowing concurrent processes to continue accessing and modifying unrelated data. Invalidations are applied individually by each process and once again leverage the use of diffing in order to avoid potential conflicts with concurrent writers on the node.

Shasta [33] uses instrumentation to implement a finer granularity coherence protocol across SMP nodes. The lack of atomicity of coherence state checks with respect to the actual load or store (the two actions consist of multiple instructions) results in protocol race conditions that require extra overhead when data is shared by processes within an SMP node. A naive solution would involve sufficient synchronization to avoid the race. Shasta avoids this costly synchronization through the selective use of explicit messages. Since protocol metadata is visible to all processes, per-process state tables are used to determine processes that are actively sharing data. Messages are sent only to these processes and overhead incurred only with active sharing.

## Leveraging Additional Hardware Support

Systems such as Ivy, TreadMarks, and Munin are implemented under the assumption that memory on remote machines is not directly accessible and must be read/accessed by requesting service from a handler on the remote machine. This is typically accomplished by sending a message over the network, which can be detected and received at the responding end either by periodically polling or by using interrupts. On traditional local area networks, interrupts are typically expensive because of the need for operating system intervention. However, the advantage is that overhead is incurred only when communication is required. Polling (periodically checking the status of the network interface), on the other hand, is relatively cheap (on the order of a few tens of processor cycles), but because the runtime system has no knowledge of when communication will occur, the frequency of checks can result in a significant overhead that is incurred regardless of whether or not data is actively shared. TreadMarks attempted to minimize the number of messages using a lazy protocol on the assumption that the per message costs (both network and processor occupancy as well as the cost to interrupt a remote processor in order to elicit a response) are at least two orders of magnitude higher than memory read and write latencies.

High-performance network technologies, such as those that conform to the Infiniband [21] standard (as well as research prototypes such as Princeton's Shrimp [7] and earlier commercial offerings such as DEC's Memory Channel, Myrinet, and Quadrics QsNet), provide low latency and high bandwidth communication. These networks achieve low latency by virtualizing the network interface. Issues of protection are separated from actual data communication. The former involves the operating system and is performed once at setup time. The latter is performed at user level, thereby achieving lower latency. The majority of these networks allow the possibility of direct access (reads and/or writes) to remote memory, changing the equation in terms of the trade-off in the number of messages used and the eagerness and laziness of the protocol. Protocols such as Cashmere and HLRC [6] leverage such direct access to perform protocol actions eagerly without the need to interrupt the remote processor.

## Sharing in the Wide Area

As “computing in the cloud,” i.e., taking advantage of geographically distributed compute resources, becomes more ubiquitous, the ability to seamlessly share information across the wide area will improve programmability. Several projects [3, 10, 14, 26, 31, 37] have examined techniques to allow data sharing in the wide area. Most enforce a strongly object-oriented programming model. Specifically, InterWeave [9] supports both language and machine heterogeneity, using an intermediate format to provide persistent data storage and to communicate seamlessly across heterogeneous machines. InterWeave leverages existing hardware and network characteristics (including support for coherence in hardware) whenever possible. The memory model is further relaxed to incorporate application-specific tolerances for delays in consistency. Writes are, however, serialized using a centralized approach (per object) for coordination.

## Compiler and Language-Level Support

Early work in incorporating compiler support with SDSMs [11] examined techniques by which data communication could be aggregated or eliminated entirely by understanding the specific access patterns of the application. In particular, the shortcomings of a generalized runtime system for shared memory relative to a program written for message passing is that in order to minimize the volume of data communicated, data communication is often separated from synchronization and data is fetched on demand at the granularity of the coherence unit. Compile-time analysis can identify data that will be accessed and inform the runtime system so that the data can be explicitly prefetched. Similarly, by using appropriate directives to the runtime to identify when entire coherence units are being written without being read, coherence communication can be eliminated entirely. Subsequently, several efforts have examined the use of SDSM as a back end for programming models such as OpenMP [17, 29]. They discuss the trade-offs between using a threaded (with a default of sharing the entire address space) versus a process model (with a default of private address spaces with shared data being specifically identified). Their work shows that naive implementations can perform poorly, and that identification of shared data is important to

the feasibility and scalability of using programming environments such as OpenMP on SMP clusters.

## Future Directions

Despite intense research and significant advances in the development of SDSM systems in the 1980s and 1990s, they remain in limited use due to their trade of scalability for the platform transparency they achieve. As multicore platforms become more ubiquitous, and as the number of cores increases, the scalability of pure hardware-based coherence is also in question, and SDSM systems may see a bigger role. Several researchers continue to explore the possibility of combining hardware and software coherence, in terms of hardware assists for a software-based coherence implementation [13] and in terms of alternating between automatic hardware coherence and software-managed incoherence based on application or compiler knowledge [24]. There is also renewed interest in the use of SDSM techniques in order to support heterogeneous platforms composed of a combination of general-purpose CPUs and accelerators [4, 32].

## Related Entries

- ▶ Cache Coherence
- ▶ Distributed-Memory Multiprocessor
- ▶ Linda
- ▶ Memory Models
- ▶ Network of Workstations
- ▶ POSIX Threads (Pthreads)
- ▶ Processes, Tasks, and Threads
- ▶ Shared-Memory Multiprocessors
- ▶ SPMD Computational Model
- ▶ Synchronization

S

## Bibliographic Notes and Further Reading

Protic et al. [30] published a compendium of works in the area of distributed shared memory circa 1994. Iftode and Singh [19] wrote an excellent survey article that encompasses both network interface advances and the incorporation of multiprocessor nodes.

## Acknowledgment

This work was supported in part by NSF grants CCF-1016902, CCF-0702505, CNS-0834451, and CNS-0509270. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the granting agencies.

## Bibliography

1. Adve S, Hill M (1990) Weak ordering: a new definition. In: Proceedings of the 17th annual international symposium on computer architecture, May 1990. ACM, New York, pp 2–14
2. Amza C, Cox A, Dwarkadas S, Keleher P, Lu H, Rajamony R, Zwaenepoel W (1996) Tread-marks: shared memory computing on networks of workstations. *IEEE Comput* 29(2):18–28
3. Bal H, Kaashoek M, Tanenbaum A (1992) Orca: a language for parallel programming of distributed systems. *IEEE Trans Softw Eng* 18(3):190–205
4. Becciu M, Cadambi S, Chakradhar S (2010) Enabling legacy applications on heterogeneous platforms. Poster paper, 2nd USENIX workshop on hot topics in parallelism (HOTPAR), Berkley, June 2010
5. Bershad B, Zekauskas M (1991) Midway: shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, Sept 1991
6. Bilas A, Jiang D, Singh JP (2001) Accelerating shared virtual memory via general-purpose network interface support. *ACM Trans Comput Syst* 19:1–35
7. Blumrich M, Li K, Alpert R, Dubnicki C, Felten E, Sandberg J (1994) Virtual memory mapped network interface for the SHRIMP multicomputer. In: Proceedings of the 21st annual international symposium on computer architecture. ACM, New York, pp 142–153
8. Carter J, Bennett J, Zwaenepoel W (1991) Implementation and performance of Munin. In: Proceedings of the 13th ACM symposium on operating systems principles, ACM Press, New York, pp 152–164
9. Chen D, Tang C, Chen X, Dwarkadas S, Scott ML (2001) Beyond S-DSM: shared state for distributed systems. Technical report 744, University of Rochester, Mar 2001
10. Chen D, Tang C, Chen X, Dwarkadas S, Scott ML (2002) Multi-level shared state for distributed systems. In: International conference on parallel processing, Aug 2002, Vancouver
11. Dwarkadas S, Cox A, Zwaenepoel W (1996) An integrated compile-time/run-time software distributed shared memory system. In: Proceedings of the 7th symposium on architectural support for programming languages and operating systems, pp 186–197, Oct 1996
12. Erlichson A, Nuckolls N, Chesson G, Hennessy J (1996) Soft-FLASH: analyzing the performance of clustered distributed virtual shared memory. In: Proceedings of the 7th symposium on architectural support for programming languages and operating systems, Oct 1996. ACM Press, New York, pp 210–220
13. Fensch C, Cintra M (2008) An OS-based alternative to full hardware coherence on tiled CMPs. In: Proceedings of the fourteenth international symposium on high-performance computer architecture symposium, February 2008, Phoenix
14. Foster I, Kesselman C (1997) Globus: a metacomputing infrastructure toolkit. *Int J Supercomputer Appl* 11(2):115–128
15. Gharachorloo K, Lenoski D, Laudon J, Gibbons P, Gupta A, Hennessy J (1990) Memory consistency and event ordering in scalable shared-memory multiprocessors. In: Proceedings of the 17th annual international symposium on computer architecture, May 1990. ACM, New York, pp 15–26
16. Gillett R (1996) Memory channel: an optimized cluster interconnect. *IEEE Micro* 16(2):12–18
17. Hu Y, Lu H, Cox AL, Zwaenepoel W (1999) OpenMP for networks of SMPs. In: Proceedings of the 13th international parallel processing symposium (IPPS/SPDP), Apr 1999. IEEE, New York, pp 302–310
18. Hu YC, Yu W, Cox AL, Wallach D, Zwaenepoel W (2003) Runtime support for distributed sharing in sage languages. *ACM Trans Comput Syst* 21(1):1–35
19. Iftode L, Singh JP (1999) Shared virtual memory: progress and challenges. *Proceedings of the IEEE* 87(3):498–507
20. Iftode L, Singh JP, Li K (1996) Scope consistency: a bridge between release consistency and entry consistency. In: ACM symposium on parallelism in algorithms and architectures, June 1996. ACM Press, New York, pp 277–287
21. Association (2010) InfiniBand. <http://www.infinibandta.org>
22. Keleher P, Cox AL, Zwaenepoel W (1992) Lazy release consistency for software distributed shared memory. In: Proceedings of the 19th annual international symposium on computer architecture, May 1992. ACM Press, New York, pp 13–21
23. Keleher P, Dwarkadas S, Cox A, Zwaenepoel W (1994) Tread-marks: distributed shared memory on standard workstations and operating systems. In: Proceedings of the 1994 winter Usenix conference, Jan 1994. USENIX Association, Berkeley, pp 115–131
24. Kelm JH, Johnson DR, Tuohy W, Lumetta SS, Patel SJ (2010) Cohesion: a hybrid memory model for accelerators. In: Proceedings of the international symposium on computer architecture (ISCA), June 2010, St Malo
25. Kontothanassis L, Hunt G, Stets R, Hardavellas N, Cierniak M, Parthasarathy S, Meira W, Dwarkadas S, Scott M (1997) VM-based shared memory on low-latency, remote-memory-access networks. In: 24th international symposium on computer architecture, June 1997. ACM Press, New York, pp 157–169
26. Lewis M, Grimshaw A (1996) The core legion object model. In: Proceedings of the 5th high performance distributed computing conference, Aug 1996, Syracuse
27. Li K (1986) Shared virtual memory on loosely coupled multiprocessors. Ph.D. thesis, Yale University
28. Li K, Hudak P (1989) Memory coherence in shared virtual memory systems. *ACM Trans Comput Syst* 7(4):321–359
29. Min S-J, Basumallik A, Eigenmann R (2003) Optimizing openmp programs on software distributed shared memory systems. *Int J Parallel Prog* 31:225–249
30. Protic J, Tomasevic M, Milutinovic V (1998) Distributed shared memory: concepts and systems. IEEE Computer Society Press, Piscataway, p 365

31. Rogerson D (1997) Inside COM. Microsoft Press, Redmond
32. Saha B, Zhou X, Chen H, Gao Y, Yan S, Rajagopalan M, Fang J, Zhang P, Ronen R, Mendelson (2009) A Programming Model for a Heterogeneous x86 Platform. In Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation, June 2009
33. Scales D, Gharachorloo K, Aggarwal A (1998) Fine-grain software distributed shared memory on smp clusters. In: Proceedings of the fourth international symposium on high-performance computer architecture symposium, Feb 1998. ACM, New York, pp 125–136
34. Scales D, Gharachorloo K, Thekkath C (1996) Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In: Proceedings of the 7th symposium on architectural support for programming languages and operating systems, Oct 1996, pp 174–185
35. Schoinas I, Falsafi B, Lebeck AR, Reinhardt SK, Larus JR, Wood DA (1994) Fine-grain access control for distributed shared memory. In: Proceedings of the 6th symposium on architectural support for programming languages and operating systems, Oct 1994. ACM, New York, pp 297–306
36. Stets R, Dwarkadas S, Hardavellas N, Hunt G, Kontothanassis L, Parthasarathy S, Scott M (1997) Cashmere-2L: software coherent shared memory on a clustered remote-write network. In: Proceedings of the 16th ACM symposium on operating systems principles, Oct 1997. ACM, New York, pp 170–183
37. van Steen M, Homburg P, Tanenbaum AS (1999) Globe: a wide-area distributed system. IEEE Concurr 7(1):70–78

## Sorting

LAXMIKANT V. KALE<sup>1</sup>, EDGAR SOLOMONIK<sup>2</sup>

<sup>1</sup>University of Illinois at Urbana-Champaign, Urbana, IL, USA

<sup>2</sup>University of California at Berkeley, Berkeley, CA, USA

### Definition

Parallel sorting is a process that given  $n$  keys distributed over  $p$  processors (numbered 0 through  $p - 1$ ), migrates the keys so that all keys on processor  $k$ , for  $k \in [0, p - 2]$ , are sorted locally and are smaller than or equal to all keys on processor  $k + 1$ .

### Discussion

#### Introduction

Parallel sorting algorithms have been studied in a variety of contexts. Early studies in parallel sorting addressed the theoretical problem of sorting  $n$  keys

distributed over  $n$  processors, using fixed interconnection networks. Modern parallel sorting algorithms focus on the scenario where  $n$  is much larger than the number of processors,  $p$ . However, even contemporary algorithms need to satisfy an array of use-cases and architectures, so various parallel sorting techniques have been analyzed for GPU-based sorting, shared memory sorting, distributed memory sorting, and external memory sorting. On most if not all of these architectures, parallel sorting is typically dominated by communication, in particular, the movement of data values associated with the keys.

Parallel sorting has a wide breadth of practical applications. Some scientific applications in the field of high-performance computing (e.g., ChaNGa) perform sorting each iteration, placing high demand on good scalability and adaptivity of parallel sorting algorithms. Parallel sorting is also utilized in the commercial field for processing of numeric as well as nonnumeric data (i.e., parallel database queries). Moreover, Integer Sort is one of the NAS parallel benchmarks.

### Parallel Sorting Algorithms

There are a few important parallel sorting algorithms that dominate multiple use-cases and serve as building blocks for more specialized sorting algorithms. The algorithms will be described for the distributed memory paradigm, though most are also applicable to other architectures and models.

#### Parallel Quicksort

Parallelization of Quicksort can be done in a variety of ways. A simplified, recursive version is described below. A more thorough analysis can be found in [1] or [2].

1. A processor broadcasts a pivot element to all  $p$  processors.
2. Each processor then splits its keys into two sections (smaller keys and larger keys) according to the pivot.
3. Two prefix sums calculate the total number of smaller keys and larger keys on the first  $k$  processors, for  $k \in [0, p - 1]$ . If,  $small_k$  and  $large_k$  are, respectively, the total numbers of smaller and larger keys on processors 0 through  $k$ , then, after this operation, processor  $k$  knows  $small_{k-1}$  and  $large_{k-1}$  as well as  $small_k$  and  $large_k$ .
4. Processor  $p - 1$  knows the total sums,  $small_{p-1}$  and  $large_{p-1}$ . It can therefore divide the set of processors

in proportion with  $small_{p-1}$  and  $large_{p-1}$ , thus determining two sets of processors ( $P_s$  and  $P_l$ ) that should be given the smaller and larger keys. This processor should also broadcast the average number of keys these two sets of processors should receive ( $small_{avg}$  and  $large_{avg}$ ).

5. Each processor  $k$  can now decide where its keys need to be sent. For example, the smaller keys should go to processor  $\left\lfloor \frac{small_{k-1}}{small_{avg}} \right\rfloor$  through processor  $\left\lfloor \frac{small_k}{small_{avg}} \right\rfloor$ .
6. After the communication, it is sufficient for the Parallel Quicksort procedure to recurse on the two processor sets ( $P_s$  and  $P_l$ ) until all data values are on the correct processors and can be sorted locally.

This algorithm generally achieves good load balance by determining the correct portions of processors that should receive smaller and larger keys. However, the necessity of moving half the data for every iteration is costly on large distributed systems. In the average case, Parallel Quicksort necessitates  $\Theta(n \log p)$  data movement. Moreover, both the quality of load balance achieved for small processor sets and the total number of recursive levels are dependent on pivot selection. Efficient implementations of Parallel Quicksort typically use more complex pivoting techniques and multiple pivots [2]. Nevertheless, Parallel Quicksort is easy to implement and can achieve good performance on some shared memory and smaller distributed systems. Additionally, the number of messages sent by each processor in step 5 is constant – typically no more than four. Therefore, Parallel Quicksort has a relatively small message latency cost of  $\Theta(p \log p)$  messages.

### Bitonic Sort

Introduced in 1968 by Batcher [3], Bitonic Sort is one of the oldest parallel sorting algorithms. This algorithm is based on the sorting of *bitonic sequences*. A *bitonic sequence* is a sequence  $S$  or any cyclic shift of  $S$ , such that  $S = S_1S_2$  where  $S_1$  is monotonically nondecreasing and  $S_2$  is monotonically nonincreasing. Further, any unsorted sequence can be treated as a series of bitonic subsequences of length two. A *bitonic merge* turns a bitonic subsequence into a fully sorted subsequence. Bitonic Sort works by application of a series of bitonic merges until the entire sequence is sorted. Applying bitonic merges on a series of bitonic subsequences effectively doubles the length of each sorted subsequence and cuts the number of bitonic subsequences in half.

Therefore, for an unsorted sequence of length  $k$ , where  $k$  is a power of two, Bitonic Sort requires  $\log k$  merges.

The main insight of Bitonic Sort is in the bitonic merge operation. A bitonic merge recursively slices a bitonic sequence into two bitonic sequences, with all elements in one sequence larger than all elements in the other. When the bitonic sequence is sliced into pieces of unit length, the entire input is sorted. Given a bitonic sequence of length  $s$ , where  $s$  is a power of two, every slice operation compares and swaps each element  $k$ , where  $k \in [0, s/2)$  with element  $k + s/2$ . These swaps result in two bitonic sequences of length  $s/2$ , with the elements in one being larger than the elements of the other. Thus, if  $s$  is a power of two, a bitonic merge requires  $\log s$  slices, which amounts to  $\log s$  comparison operations on every element.

In the case of  $n = p$  on a hypercube network, a bitonic merge requires  $\Theta(\log n)$  swaps, one swap in each hypercube dimension. The algorithm requires  $\log n$  merges on such a network, for a composed running time of  $\Theta(\log^2 n)$ . Adaptive Bitonic Sorting [4] avoids the redundant comparisons in Bitonic Sort and achieves a runtime complexity of  $\Theta(\log n)$ . Further, Bitonic Sort is also asymptotically optimal on mesh connected networks [5].

As proposed by Blelloch [6], Bitonic Sort can be extended for the case of  $n \geq p$ , by introducing virtual hypercube dimensions within each processor. Alternatively, a more efficient sequential sorting algorithm can be used for the sequential sorting work. Though historically significant and elegant in nature, Bitonic Sort is not widely used on modern supercomputers, since the algorithm requires data to be migrated through the network  $\Theta(\log^2 p)$  times or, in the case of Adaptive Bitonic Sorting,  $\Theta(\log p)$  times. Nevertheless, Bitonic Sort has been used in a wide variety of applications, ranging from network router hardware to sorting on GPUs [7–9]. A more thorough analysis and justification of correctness of this algorithm can be found in the Bitonic Sort article.

### Parallel Radix Sort

Radix Sort is a counting-based sorting algorithm that relies on the bitwise representation of keys. An  $r$ -bit radix looks at  $r$  bits of each key at a time and permutes the keys to move to one of the  $2^r$  buckets, where the  $r$ -bit value of each key corresponds to its destination bucket. If each key has  $b$  bits, by looking at the  $r$  least-significant bits first, Radix Sort can sort the entire

dataset in  $\lceil \frac{b}{r} \rceil$  permutations. Therefore, this algorithm has a complexity of  $\Theta(\frac{b}{r}n)$ . Notably, this complexity is linear with respect to  $n$ , a feature that cannot be matched by comparison-based sorting algorithms, which must do at least  $\Theta(n \log n)$  comparison operations. However, Radix Sort is inherently cache-inefficient since each key may need to go to any one of the  $2^r$  buckets for each iteration, independent of which bucket it resided in the previous iteration [10]. A further limitation on Radix Sort is its reliance on the bitwise representation of keys, which is satisfied for integers and requires a simple transformation for floats, but is not necessarily possible or simple for strings and other data types.

Parallel Radix Sort is implemented by assigning a subset of the buckets to each processor [6]. Thus, each of  $\lceil \frac{b}{r} \rceil$  permutations would result in a step of all-to-all communication. Load balancing is also achieved relatively easily by computing a histogram of the number of keys headed for each bucket at the beginning of each step. These histograms can be computed using local data on each processor first, then summed up using a reduction. Given a summed-up histogram, a single processor can then adaptively decide the set of buckets to assign to each processor and broadcast this information to all other processors. Each processor then sends all of its keys to the processor that owns the appropriate bucket for each key, using all-to-all communication.

A good way to improve the efficiency of local histogram computation in Radix Sort is to use an auxiliary set of low-bit counters [11]. Given a large  $r$ , it is unlikely that an array of  $2^r$  32-bit counters can fit into the L1 cache. However, by using an array of  $2^r$  8-bit counters and incrementing the 32-bit counter array only when the 8-bit counters are about to overflow, cache performance can be significantly improved.

The simplicity of Radix Sort as well as its relatively good all-around performance and scalability have made it a popular parallel sorting algorithm in a variety of contexts. However, Radix Sort still suffers from cache-efficiency problems and requires multiple steps of all-to-all communication, which can be an extremely costly operation on a large enough system.

## Sample Sort

Sample Sorting is a *splitter-based* method which performs data partitioning by collecting a sample of the entire dataset [12]. *Splitter-based* parallel sorting algorithms determine the destinations for each key by

determining a range bounded by two *splitters* for each processor. These *splitters* are simply values meant to subdivide the entire key range into  $p$  approximately equal chunks, so that each processor can be assigned a roughly even-sized chunk. After the splitters have been determined and broadcasted to all processors, a single all-to-all communication step suffices in giving each processor the correct data.

Parallel Sorting by Regular Sampling is a Sample Sorting algorithm introduced by Shi and Schaeffer [13]. This algorithm determines the correct splitters by collecting a sample of data from each processor. Sorting by Regular Sampling uses a regular sample of size  $p - 1$  and generally operates as follows:

1. Sort local data on each processor.
2. Collect sample of size  $p - 1$  on each processor with the  $k$ th element of each sample as element  $\frac{n}{p} \times \frac{k+1}{p}$  of the local data.
3. Merge the  $p$  samples to form a combined sorted sample of size  $p \times (p - 1)$ .
4. Define  $p - 1$  splitters with the  $k$ th splitter as element  $p \times (k + \frac{1}{2})$  of the sorted sample.
5. Broadcast splitters to all processors.
6. On each processor, subdivide the local keys into  $p$  chunks (numbered 0 through  $p - 1$ ) according to the splitters. Send each chunk to equivalently numbered processor.
7. Merge the incoming data on each processor.

Collecting a regular sample of size  $p - 1$  from each processor has been proven to guarantee no more than  $\frac{2n}{p}$  elements on any processor [14] and shown to achieve almost perfect load balance for most practical distributions. The algorithm has also been shown to be asymptotically optimal as long as  $n \geq p^3$ .

Regular Sample Sort is easy to implement, insensitive to key distribution, and optimal in terms of the data movement required (there is a single all-to-all step, so each key gets moved only once). The algorithm has been shown to perform very well for  $n \gg p$  and has been the parallel sorting algorithm of choice for many modern applications. One important issue with the traditional Sorting by Regular Sampling technique is the requirement of a combined sample of size  $\Theta(p^2)$ . For a small-enough  $p$  this is not a major cost; however, for high-performance computing applications running on thousands of processors, this cost begins to overwhelm

the running time of the sorting algorithm and the  $n \geq p^3$  assumption crumbles.

Sorting by Random Sampling [6] is a parallel sorting technique that can potentially alleviate some of the drawbacks of Parallel Sorting by Regular Sampling. Instead of selecting an evenly distributed sample of size  $p - 1$  from each processor, random samples of size  $s$  are collected from the initial local datasets to form a combined sample of size  $s \times p$ . The  $s$  parameter has to be carefully chosen, but sometimes sufficient load balance can be achieved for  $s < p$ . Additionally, Sorting by Random Sampling allows for better potential overlap between computation and communication since the sample can be collected before the local sorting is done.

Another interesting variation of Sample Sort was introduced by Helman et al. [15]. Instead of collecting a sample, this sorting procedure first permutes the elements randomly with a randomized data transpose, then simply selects the splitters on one processor. To execute the transpose, each processor randomly assigns each of its local keys to one of the  $p$  buckets, then sends the  $j$ th bucket to the  $j$ th processor. With high probability, this permutation guarantees that any processor's elements will be representative of the entire key set. Thus, this algorithm avoids the cost of collecting and analyzing a large sample on a single processor. One processor still needs to select and broadcast splitters but this is a relatively cheap operation. The main disadvantage of this technique is the extra all-to-all communication round, which is very expensive on a large system. Additionally, as  $p$  scales to  $n/p$ , the load balance achieved by the algorithm deteriorates.

### Histogram Sort

Histogram Sort [16] is another splitter-based method for parallel sorting. Like Sample Sort, Histogram Sort determines a set of  $p - 1$  splitters to divide the keys into  $p$  evenly sized sections. However, it achieves this task by taking an iterative guessing approach rather than simply collecting one big sample. Each set of splitter-guesses, called the *probe*, is matched up to the data then adjusted, until *satisfactory values* for all splitters have been determined. A *satisfactory value* for the  $k$ th splitter needs to divide the data so that approximately  $\frac{k+1}{p} \times n$  keys are smaller than the splitter value. Typically, a threshold range is established for each splitter so that the splitter-guesses can converge quicker. The

$k$ th splitter must divide the data within the range of keys,  $(\frac{nk}{p} - \frac{nT}{p}, \frac{nk}{p} + \frac{nT}{p})$ , where  $T$  is the given threshold. A basic implementation of Histogram Sort operates as follows:

1. Sort local data on each processor.
2. Define a probe of  $p - 1$  splitter-guesses distributed evenly over the key data range.
3. Broadcast the probe to all processors.
4. Produce local histograms by determining how much of each processor's local data fits between each key range defined by the splitter-guesses.
5. Sum up the histograms from each processor using a reduction to form a complete histogram.
6. Analyze the complete histogram on a single processor, determining any splitter values satisfied by a splitter-guess, and bounding any unsatisfied splitter values by the closest splitter-guesses.
7. If any splitters have not been satisfied, produce a new probe and go back to step 3.
8. Broadcast splitters to all processors.
9. On each processor, subdivide the local keys into  $p$  chunks (numbered 0 through  $p - 1$ ) according to the splitters. Send each chunk to equivalently numbered processor.
10. Merge the incoming data on each processor.

This iterative technique can refine the splitter-guesses to an arbitrarily narrow threshold range. Quick convergence can be guaranteed by defining each new probe to contain a guess in the middle of the bounded range for each unsatisfied splitter.

Like any splitter-based sort, Histogram Sort is optimal in terms of the data movement required. However, unlike all previously described sorting algorithms, the running time of Histogram Sort depends on the distribution of the data through the data range. The iterative approach employed by this sorting procedure guarantees desired level of load balance which Sample Sort and Radix Sort cannot. Additionally, the probing technique is flexible and does not require that local data be sorted immediately [17]. This advantage allows an excellent opportunity for the exploitation of communication and computation overlap. However, Histogram Sort is generally more difficult to implement than common alternatives such as Radix Sort or Sample Sort.

## Architectures and Theoretical Models

Parallelism is exhibited by a variety of computer architectures. Shared memory multiprocessors, distributed systems, supercomputers, sorting networks, and GPUs are all fundamentally different parallel computing constructions. As such, parallel sorting algorithms need to be designed and tuned for each of these architectures specifically.

## Sorting Networks and Early Theoretical Models

Traditional parallel sorting targeted the problem of sorting  $n$  numbers on  $n$  processors using a fixed interconnection network. Bitonic Sort [3] was an early success as it provided a  $\Theta(\log^2 n)$ -depth sorting network. The bitonic sorting network also yielded a practical algorithm for sorting  $n$  keys in parallel using  $n$  processors in  $\Theta(\log^2 n)$  time on network topologies such as the hypercube and shuffle-exchange. In 1983, Ajtai et al. [18], introduced an  $\Theta(\log n)$ -depth sorting network capable of sorting  $n$  keys in  $\Theta(\log n)$  time using  $\Theta(n \log n)$  comparators. However, this construction was shown to lead to less-efficient networks than Bitonic Sort for reasonable values of  $n$ . Leighton [19] introduced the Column Sort algorithm. He showed that, based on Column Sort, for any sorting network with  $\Theta(n \log n)$  comparators and  $\Theta(\log n)$  depth, one can construct a corresponding constant-degree network of  $n$  processors that can sort in  $\Theta(\log n)$  time.

Much work has targeted the complexity of parallel sorting on the less-restrictive PRAM model where each processor can access the memory of all other processors in constant time. In 1986, Cole [20] introduced an efficient and practical parallel sorting algorithm with versions for the CREW (concurrent read only) and EREW (no concurrent access) PRAM models. This algorithm was based on a simple tree-based Mergesort, but was elegantly pipelined to achieve a  $\Theta(\log n)$  complexity for sorting  $n$  keys using  $n$  processors. Cole's merge sort used a  $\Theta(\log n)$  time merging algorithm, which naturally leads to a  $\Theta(\log^2 n)$  sorting algorithm. However, his sorting algorithm used results from lower levels of the merge tree to partially precompute the merging done in higher levels of the merge tree. Thus, the sorting algorithm was designed so that at every node of the merge tree only a constant amount of work needed to be done, yielding a  $\Theta(\log n)$  overall sorting complexity.

Sorting networks have also been extensively studied for the VLSI model of computation. The VLSI model focuses on area-time complexity, that is, the area of the chip on which the network is constructed and the running time. A good analysis of lower bounds for the complexity of such VLSI sorters can be found in [21].

These theoretical methods have been studied intensively in literature and have yielded many elegant parallel sorting algorithms. However, the theoretical machine models they were designed for are no longer representative or directly useful for current parallel computer architectures. Nevertheless, these studies provide valuable groundwork for modern and future parallel sorting algorithms. Moreover, sorting networks may prove to be useful for emerging architectures such as GPUs and chip multicores.

## GPU-Based Sorting

Early GPU-based sorting algorithms utilized a limited graphics API which, among other restrictions, did not allow scatter operations and made Bitonic Sort the dominant choice. GPUMeraSort [7] is an early efficient hybrid algorithm that uses Radix Sort and Bitonic Sort. GPUMeraSort was designed for GPU-based external sorting, but is also general to in-memory GPU-based sorting. A weakness of the GPUMeraSort algorithm is its  $\Theta(n \log^2 n)$  running time, typical of parallel sorting algorithms based on Bitonic Sort. GPU-ABiSort [8] improved over GPUMeraSort by using Adaptive Bitonic Sorting [4], lowering the theoretical complexity to  $\Theta(n \log n)$  and often demonstrating a lower practical running time.

Newer GPUs, assisted by the CUDA software environment, allow for efficient scan primitives and a much broader set of parallel sorting algorithms. Efficient versions of Radix Sort and a parallel Mergesort are presented by Satish et al. [9]. Newer GPU-based sorting algorithms also exploit instruction-level parallelism by performing steps such as merging using custom vector operations. An array of various other GPU-based sorting algorithms, which are not detailed here, can be found in literature.

Current results on modern GPUs suggest that Radix Sort typically performs best, particularly when the key size is small [9]. Radix Sort is well fit for GPU execution since keys can be processed independently and synchronization is almost purely in the form of prefix sums,

which can be executed with high efficiency on GPUs. Radix Sort also requires few or no low-level branches, unlike comparison-based sorting algorithms. Finally, the cache inefficiency of Radix Sort has been less costly on GPUs since most GPUs have no cache. However, newer GPUs, such as the NVIDIA GPUs of compute capability 2.0, already have small caches and are rapidly evolving. It is hard to predict whether Radix Sort or a different sorting algorithm will prove most efficient on emerging GPU and accelerator architectures.

### Shared Memory Sorting

Parallel merging techniques have commonly been used to produce simple shared memory parallel sorting algorithms. Francis and Mathieson [22] present a  $k$ -processor merge that allows for all processors to participate in a parallel merge tree, with two processors participating in each merge during the first merging stage and all processors participating in the final merge at the head of the tree. Good performance is achieved by subdividing each of the two arrays of size  $a$  and  $b$  being merged into  $k$  sections, where  $k$  is the number of processors participating in the merge. The subdivisions are selected so that the  $i$ th processor merges the  $i$ th section of each of the two arrays and produces elements  $\frac{i}{k}(a+b)$  through  $\frac{i+1}{k}(a+b)$  of the merged array.

Merge-based algorithms, such as the one detailed above, as well as parallel versions of Quicksort and Mergesort, are predominant on contemporary shared memory multiprocessor architectures. However, with the advent of increased parallelism in chip architectures, techniques such as sampling and histogramming may become more viable.

### Distributed Memory Sorting

Parallel Sorting algorithms for distributed memory architectures are typically used in the high-performance computing field, and often require good scaling on modern supercomputers. In the 1990s Radix Sort and Bitonic Sort were widely used. However, as architectures evolved, these sorting techniques proved insufficient. Modern machines have thousands of cores, so, to achieve good scaling interprocessor communication needs to be minimized. Therefore, splitter-based algorithms such as Sample Sort and Histogram Sort are now more commonly used for distributed memory sorting.

Splitter-based parallel sorting algorithms have minimal communication since they only move data once.

### Future Directions

Despite the extraordinarily large amount of literature on parallel sorting, the demand for optimized parallel sorting algorithms continues to motivate novel algorithms and more research on the topic. Moreover, the continuously changing and, more recently, diverging nature of parallel architectures has made parallel sorting an evolving problem.

High-performance computer architectures are rapidly growing in size, creating a premium on parallel sorting algorithms that have minimal communication and good overlap between computation and communication. Since parallel sorting necessitates communication between all processors, the creation and optimization of topology-aware all-to-all personalized communication strategies is extremely valuable for many splitter-based parallel sorting algorithms. As previously mentioned, algorithms similar to Sample Sort and Histogram Sort are the most viable candidates for this field due to the minimal nature of the communication they need to perform.

Shared memory sorting algorithms are beginning to face a challenge that most modern sequential algorithms have to endure. The demand for good cache efficiency is now a key constraint for all algorithms due to the increasing relative cost of memory accesses. Parallel sorting is being studied under the cache-oblivious model [23] in an attempt to reevaluate previously established algorithms and introduce better ones.

Currently, parallel sorting using accelerators, such as GPUs, is probably the most active of all parallel sorting research areas due to the rapid changes and advances happening in the accelerator architecture field. Little can be said about which algorithms will dominate this field in the future, due to the influx of novel GPU-based sorting algorithms and newer accelerators in recent years.

The wide use of sorting in computer science along with the popularization of parallel architectures and parallel programming necessitates the implementation of parallel sorting libraries. Such libraries can be difficult to standardize, however, especially

for the efficiency-sensitive field of high-performance computing. Nevertheless, these libraries are quickly emerging, especially under the shared memory computing paradigm.

## Related Entries

- [Algorithm Engineering](#)
- [All-to-All](#)
- [Bitonic Sort](#)
- [Bitonic Sorting, Adaptive](#)
- [Collective Communication](#)
- [Data Mining](#)
- [NAS Parallel Benchmarks](#)
- [PRAM \(Parallel Random Access Machines\)](#)

## Bibliographic Notes and Further Reading

The literature on parallel sorting is very large and this entry is forced to cite only a select few. The sources cited are a mixture of the largest impact publications and the most modern publications. A few of the sources also give useful information on multiple parallel sorting algorithms. Blelloch et al. [6] provide an in-depth experimental and theoretical comparative analysis of a few of the most important distributed memory parallel sorting algorithms. Satish et al. [9] provide good analysis of a few of the most modern GPU-sorting algorithms. Vitter [24] presents a good analysis of external memory parallel sorting.

## Bibliography

1. Kumar V, Grama A, Gupta A, Karypis G (1994) Introduction to parallel computing: design and analysis of algorithms. Benjamin-Cummings, Redwood City, CA
2. Sanders P, Hansch T (1997) Efficient massively parallel quicksort. In: IRREGULAR '97: Proceedings of the 4th International Symposium on Solving Irregularly Structured Problems in Parallel, pp 13–24. Springer-Verlag, London, UK
3. Batcher K (1968) Sorting networks and their application. Proc. SICC, AFIPS 32:307–314
4. Bilardi G, Nicolau A (1986) Adaptive bitonic sorting: an optimal parallel algorithm for shared memory machines. Technical report, Ithaca, NY
5. Thompson CD, Kung HT (1977) Sorting on a mesh-connected parallel computer. Commun ACM 20(4):263–271
6. Blelloch G et al. (1991) A comparison of sorting algorithms for the Connection Machine CM-2. In: Proceedings of the Symposium on Parallel Algorithms and Architectures, July 1991
7. Govindaraju N, Gray J, Kumar R, Manocha D (2006) Gputerasort: high performance graphics co-processor sorting for large database management. In: SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data, pp 325–336. ACM, New York
8. Greb A, Zachmann G (2006) Gpu-abisort: optimal parallel sorting on stream architectures. In: Parallel and Distributed Processing Symposium, 2006. IPDPS 2006, 20th International, pp 45–54, April 2006
9. Satish N, Harris M, Garland M (2009) Designing efficient sorting algorithms for manycore gpus. Parallel and Distributed Processing Symposium, International, Rome, pp 1–10
10. LaMarca A, Ladner RE (1997) The influence of caches on the performance of sorting. In: SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms, pp 370–379. Society for Industrial and Applied Mathematics, Philadelphia, PA
11. Thearling K, Smith S (1992) An improved supercomputer sorting benchmark. In: Proceedings of the Supercomputing, November 1992.
12. Huang JS, Chow YC (1983) Parallel sorting and data partitioning by sampling. In: Proceedings of the Seventh International Computer Software and Applications Conference, November 1983
13. Shi H, Schaeffer J (1992) Parallel sorting by regular sampling. J Parallel Distrib Comput 14:361–372
14. Li X, Lu P, Schaeffer J, Shillington J, Wong PS, Shi H (1993) On the versatility of parallel sorting by regular sampling. Parallel Comput 19(10):1079–1103
15. Helman DR, Bader DA, JáJá J (1998) A randomized parallel sorting algorithm with an experimental study. J Parallel Distrib Comput 52(1):1–23
16. Kale LV, Krishnan S (1993) A comparison based parallel sorting algorithm. In: Proceedings of the 22nd International Conference on Parallel Processing, pp 196–200, St. Charles, IL, August 1993
17. Solomonik E, Kale LV (2010) Highly scalable parallel sorting. In: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS). Urbana, IL, April 2010
18. Ajtai M, Komlós J, Szemerédi E (1983) Sorting in  $c \log n$  parallel steps. Combinatorica 3(1):1–19
19. Leighton T (1984) Tight bounds on the complexity of parallel sorting. In: STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing, pp 71–80. ACM, New York
20. Cole R (1986) Parallel merge sort. In: SFCS '86: Proceedings of the 27th Annual Symposium on Foundations of Computer Science, pp 511–516. IEEE Computer Society, Washington, DC
21. Bilardi G, Preparata FP (1986) Area-time lower-bound techniques with applications to sorting. Algorithmica 1(1):65–91
22. Francis RS, Mathieson ID (1988) A benchmark parallel sort for shared memory multiprocessors. Comput IEEE Trans 37(12):1619–1626

23. Blelloch GE, Gibbons PB, Simhadri HV (2009) Brief announcement: low depth cache-oblivious sorting. In: SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures, pp 121–123. ACM, New York
24. Vitter JS (2001) External memory algorithms and data structures: dealing with massive data. ACM Comput Surv 33(2):209–271

## Space-Filling Curves

MICHAEL BADER<sup>1</sup>, HANS-JOACHIM BUNGARTZ<sup>2</sup>,  
MIRIAM MEHL<sup>2</sup>

<sup>1</sup>Universität Stuttgart, Stuttgart, Germany

<sup>2</sup>Technische Universität München, Garching, Germany

### Synonyms

FASS (space-filling, self-avoiding, simple, and self-similar)-curves

### Definition

A space-filling curve is a continuous and surjective mapping from a 1D parameter interval, say  $[0, 1]$ , onto a higher-dimensional domain, say the unit square in 2D or the unit cube in 3D. Although this, at first glance, seems to be of a purely mathematical interest, space-filling curves and their recursive construction process have obtained a broad impact on scientific computing in general and on the parallelization of numerical algorithms for spatially discretized problems in particular.

### Discussion

#### Introduction

Space-filling curves (SFC) were presented at the end of the nineteenth century – first by Peano (1890) and Hilbert (1891), and later by Moore, Lebesgue, Sierpinski, and others. The idea that some curves (i.e., something actually one-dimensional) may completely cover an area or a volume sounds somewhat strange and formerly caused mathematicians to call them “topological monsters.” The construction of all SFC follows basically the same principle: start with a *generator* indicating an order of traversal through the similar first-level substructures of the initial domain (the unit square, unit cube, etc.), and produce the next iterates by successively subdividing the domain in the same way as well

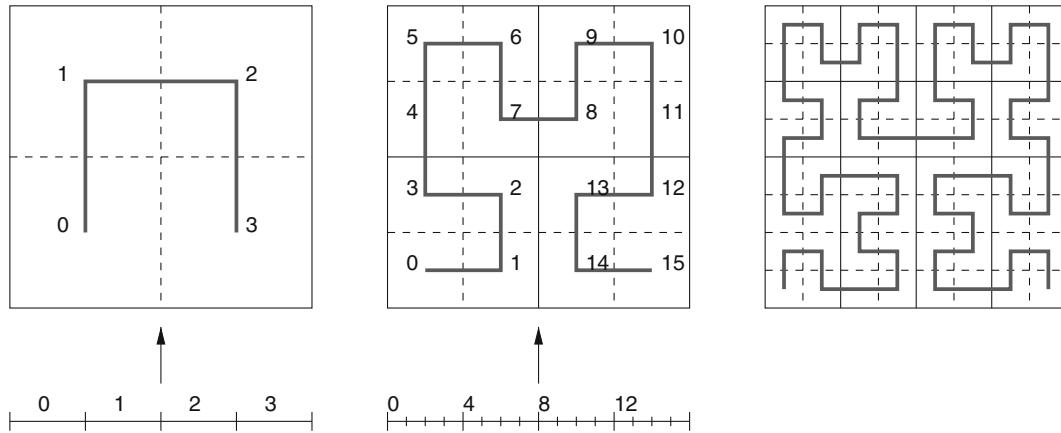
as placing and connecting shrunked, rotated, or reflected versions of the generator in the next-level subdomains. This has to happen in an appropriate way, ensuring the two properties *neighborhood* (neighboring subintervals are mapped to neighboring subdomains) and *inclusion* (subintervals of an interval are mapped to subdomains of the interval’s image). If all is done properly, it can be proven that the limit of this recursive process in fact defines a curve that completely fills the target domain and, hence, results in an SFC.

While the SFC itself as the asymptotical result of a continuous limit process is more a playground of mathematics, the iterative or recursive construction or, to be precise, the underlying mapping can be used for sequentializing higher-dimensional domains and data. Roughly speaking, these higher-dimensional data (elements of a finite element mesh, particles in a molecular dynamics simulation, stars in an astrophysics simulation, pixels in an image, voxels in a geometric model, or even entries in a data base, e.g.) now appear as pearls on a thread. Thus, via the locality properties of the SFC mapping, clusters of data can be easily identified (interacting finite elements, neighboring stars, similar data base entries, etc.). This helps for the efficient processing of tasks such as answering data base queries, defining hardware-aware traversal strategies through adaptively refined meshes or heterogeneous particle sets, and, of course, subdividing the data across cores or processors in the sense of (static or dynamic) load distribution in parallel computing. The main idea for the latter is that the linear (1D) arrangement of the data via the mapping basically reduces the load distribution problem to the sorting of indices.

#### Construction

Space-filling curves are typically constructed via an iterative or recursive process. The source interval and the target domain are recursively substructured into smaller intervals and domains. From each level of recursion to the next, a mapping between subintervals and subdomains is constructed, where the child intervals of a subinterval are typically mapped to the children of the image of the parent interval. The SFC is then defined as the image of the limit of these mappings.

Figure 1 illustrates this recursive construction process for the 2D Hilbert curve. From each level to the next, the subintervals are split into four congruent



**Space-Filling Curves.** Fig. 1 The first three iterations of the Hilbert curve

subintervals. Likewise, the square subdomains are split into four subsquares. In the  $n$ th iteration, an interval  $[i \cdot 4^{-n}, (i + 1) \cdot 4^{-n}]$  is mapped to the  $i$ th subsquare, as indicated in the figure. The curves in Fig. 1 connect the subsquares of the  $n$ th level according to their source intervals and are called *iterations* of the Hilbert curve. For the limit  $n \rightarrow \infty$ , the iterations shall, in an intuitive sense, converge to the Hilbert curve.

More formal: for any given parameter  $t \in [0, 1]$ , there exists a sequence of nested intervals  $[i_n \cdot 4^{-n}, (i_n + 1) \cdot 4^{-n}]$  that all contain  $t$ . The corresponding subsquares converge to a point  $h(t) \in [0, 1]^2$ . The image of the mapping  $h$  defined by that construction is called the Hilbert curve.  $h$  shall be called the Hilbert mapping.

### Computation of Mappings

Figure 1 shows that the  $n$ th Hilbert iteration consists of the connection of four  $(n - 1)$ th Hilbert iterations, which are scaled, rotated, and translated appropriately. For  $n \rightarrow \infty$ , this turns into a fix-point argument: the Hilbert curve consists of the connection of four suitably scaled, rotated, and translated Hilbert curves. The respective transformations shall be given by operations  $H_q$ , where  $q \in \{0, 1, 2, 3\}$  determines the relative position of the transformed Hilbert curve. This leads to the following recursive equation:

$$h(0_4.q_1q_2q_3q_4 \dots) = H_{q_1} \circ h(0_4.q_2q_3q_4 \dots). \quad (1)$$

If the parameter  $t$  is given as a quarternary fraction, i.e.,  $t = 0_4.q_1q_2q_3 \dots = \sum_n q_n \frac{1}{4^n}$ , then the interval numbers  $i_n$  and the relative position of subintervals within their parent can be obtained from the quarternary digits  $q_n$ .

For finite quarternary fractions, successive application of Eq. 1 leads to the following formula to compute  $h$ :

$$h(0_4.q_1q_2 \dots q_n) = H_{q_1} \circ H_{q_2} \circ \dots \circ H_{q_n} \circ h(0). \quad (2)$$

For the Hilbert curve, the operators  $H_q$  are defined as:

$$\begin{aligned} H_0 &:= \begin{pmatrix} x \\ y \end{pmatrix} \rightarrow \begin{pmatrix} \frac{1}{2}y \\ \frac{1}{2}x \end{pmatrix} \\ H_1 &:= \begin{pmatrix} x \\ y \end{pmatrix} \rightarrow \begin{pmatrix} \frac{1}{2}x \\ \frac{1}{2}y + \frac{1}{2} \end{pmatrix} \\ H_2 &:= \begin{pmatrix} x \\ y \end{pmatrix} \rightarrow \begin{pmatrix} \frac{1}{2}x + \frac{1}{2} \\ \frac{1}{2}y + \frac{1}{2} \end{pmatrix} \\ H_3 &:= \begin{pmatrix} x \\ y \end{pmatrix} \rightarrow \begin{pmatrix} -\frac{1}{2}y + 1 \\ -\frac{1}{2}x + \frac{1}{2} \end{pmatrix} \end{aligned}$$

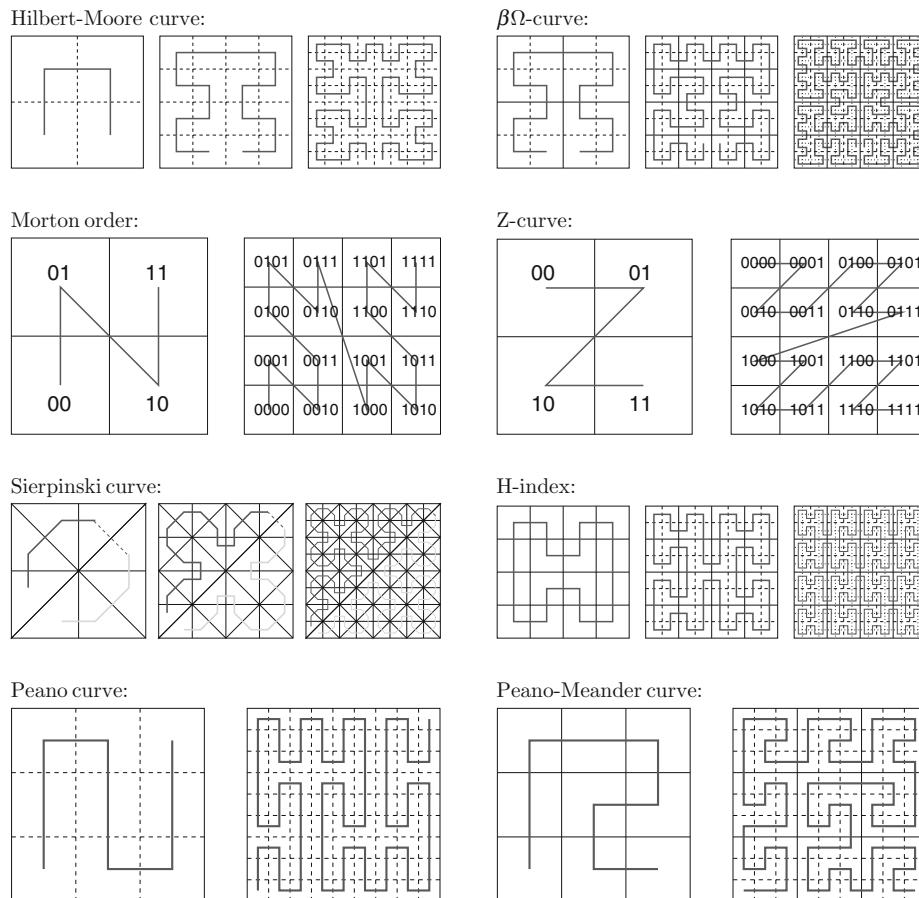
Equations 1 and 2 may be easily turned into an algorithm to compute the image point  $h(t)$  from any given parameter  $t$ .

Inverting this process leads to algorithms that find a parameter  $t$  that is mapped to a given point  $p = h(t)$ . However, note that the Hilbert mapping  $h$  is not bijective; hence, an inverse mapping  $h^{-1}$  does not exist. Still, it is possible to construct mappings  $\bar{h}^{-1}$  that return a uniquely defined parameter  $t = \bar{h}^{-1}(p)$  with  $p = h(t)$ . Note that for practical applications, only the discrete orders induced by  $h$  are of interest. These orders are usually bijective – for example, the relation between

subsquares and subintervals during the construction of the Hilbert mapping is a bijective one. Bijectivity is only lost with  $n \rightarrow \infty$ .

### Examples of Space-Filling Curves

**Figure 2** illustrates the construction of different SFC. All of them are constructed in a similar way as the Hilbert curve and can be computed via an analogous approach. The Hilbert-Moore curve combines four regular, scaled-down Hilbert curves to a closed curve. The  $\beta\Omega$ -curve is a Hilbert-like curve that uses nonuniform refinement patterns throughout the iterations. Similar to the Hilbert-Moore curve, it is a closed curve. Morton order and the Z-curve result from a bit-interleaving code for 2D grids. They lead to discontinuous mappings from the unit interval to the unit square. However, the Lebesgue curve uses the same construction, but maps the Cantor set to the unit square in order to obtain continuity.



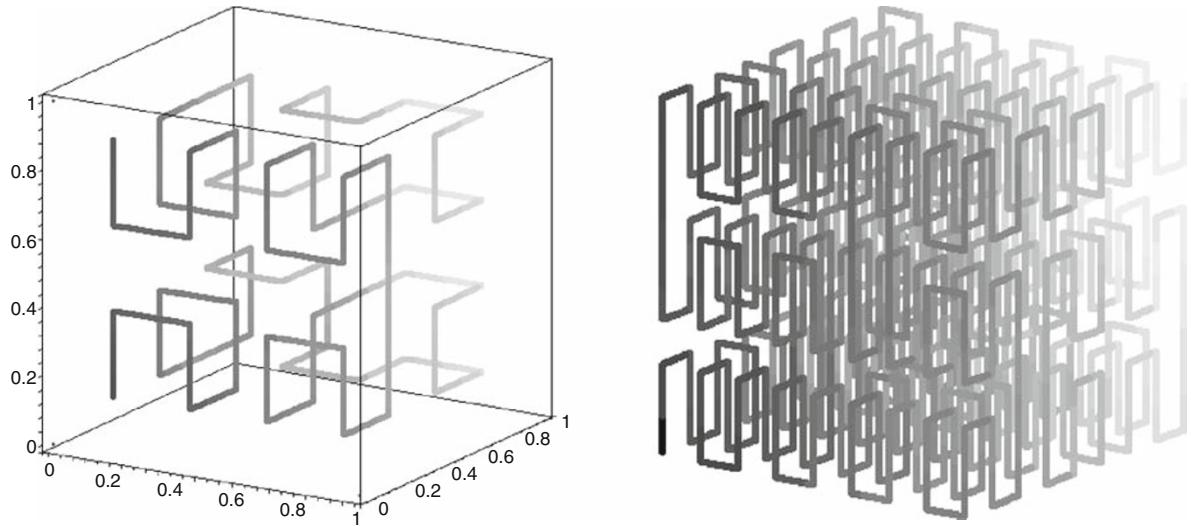
**Space-Filling Curves. Fig. 2** Several examples for the construction of space-filling curves on the unit square

The Sierpinski curve is a curve that is generated via recursive substructuring of triangles. The combination of two triangle-filling Sierpinski curves leads to a curve that fills the unit square. The H-index follows a construction compatible to that for the Sierpinski curve, but generates a discrete order on Cartesian grids. Infinite refinement of the H-index, then, leads to the Sierpinski curve. The Peano curve, finally, as well as its variant, the Peano-Meander curve, are square-filling curves that are based on a recursive  $3 \times 3$ -refinement of the unit square.

**Figure 3** provides snapshots of the construction processes of 3D Hilbert and Peano curves.

### Locality Properties of Space-Filling Curves

The recursive construction of SFC leads to locality properties that can be exploited for efficient load distribution and load balancing. The Hilbert curve, for example,



**Space-Filling Curves. Fig. 3** Second iterations of three-dimensional Hilbert and Peano curves

and curves that follow a similar construction, can be shown to be Hölder continuous, i.e., for two parameters  $t_0$  and  $t_1$ , the distance of the images  $h(t_0)$  and  $h(t_1)$  is bounded by

$$\|h(t_0) - h(t_1)\|_2 \leq C |t_0 - t_1|^{1/d}, \quad (3)$$

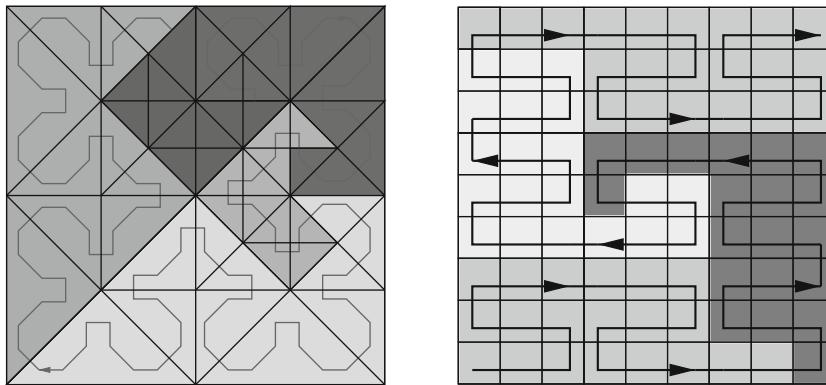
where  $d$  denotes the dimension. As  $|t_0 - t_1|$  is also equal to the area covered by the space-filling curve segment defined by the parameter interval  $[t_0, t_1]$  (i.e.,  $h$  is parameterized by area or volume), Eq. 3 gives a relation between the area covered by a curve segment and the distance between the end points of the curve. It is thus a measure for the compactness of partitions defined by curve segments. For  $d$ -dimensional objects such as spheres or cubes, the volume typically grows with the  $d$ th power of the extent (diameter, e.g.) of the object. Hence, an exponent of  $d^{-1}$  is asymptotically optimal, and the constant  $C$  characterizes the compactness of an SFC.

### High Performance Computing and Load Balancing with Space-Filling Curves

The ability of recursively substructuring and linearizing high-dimensional domains is mainly responsible for the revival of SFC – or their arrival in computational applications – starting in the 1960s of the last century. As some milestones, the Z-order (Morton-order) curve has been proposed for file sequencing in geodetic data

bases in 1966 by G.M. Morton [15]; *quadtrees* (nothing else than 2D Lebesgue SFC) were introduced in image processing [9, 19]; *octrees*, their 3D counterparts, marked the crucial idea for breakthroughs in the complexity of particle simulations, the *Barnes Hut* [3] as well as the *Fast Multipole* algorithms [11]. Octrees were also successfully used in CAD [10, 12], as well as for organizational tasks such as grid generation [20, 21] or steering volume-oriented simulations [16], and the portfolio of SFC of practical relevance is still widening.

The locality properties of SFC yield a high locality of data access and, therewith, a high efficiency of cache-usage for various kinds of applications. This will be a crucial aspect for future computing architectures, in particular multi-core architectures, where the memory bottleneck will become even more severe than already for today's high-performance computers. For example, the Peano curve is used for highly efficient block-structured matrix–matrix products [2], and it can serve as a general paradigm for PDE frameworks, where all the steps of geometry representation, adaptive grid generation, grid traversal, data structure design, and parallelization follow an SFC-based strategy [8, 13]. While all the SFC discussed so far refer to structured Cartesian grids or subdomain structures, the Sierpinski curve is defined on a triangular master domain and has been recently used to benefit from SFC characteristics also in the context of managing, traversing, and distributing triangular finite element meshes [1].



**Space-Filling Curves.** Fig. 4 Partitioning of a triangulation according to the Sierpinski curve and a Cartesian grid according to the Peano curve

Hilbert or Hilbert-Peano curves were probably the first SFC to see a broad application for load distribution and load balancing [6, 17]. Meanwhile, SFC-based strategies have become a well-established tool frequently outperforming alternatives such as graph partitioning. The classical SFC-based load-balancing approach is to queue up grid elements like pearls on a thread and to cut this queue into pieces with equal workload [4, 6, 7, 17, 18, 22, 24]. Fig. 4 shows examples for the partitioning of a triangulation according to the Sierpinski curve and a Cartesian grid according to the Peano curve. This reduces the load balancing problem to a problem of sorting data according to their positions on a space-filling curve. The locality properties of SFC yield connected partitions with quasi-minimal surfaces that is quasi-minimal communication costs. However, the constants involved in the quasi-optimality statement can be rather large [24].

More recent approaches combine the space-filling curve ordering with a tree-based domain decomposition [5, 14, 23]. This approach has the advantage that already the domain decomposition itself as well as dynamical rebalancing can easily be done fully parallel and that it fits in a natural way with highly efficient multilevel numerical methods such as multigrid solvers.

Summarizing, it is obvious that although load balancing is the most attractive application of SFC in the context of parallel computing, the scope of SFC has become much wider with their parallelization characteristics often just being one side effect.

## Related Entries

- [Domain Decomposition](#)
- [Hierarchical data format](#)

## Bibliographic Notes and Further Reading

1. Sagan H (1994) Space-filling curves, Springer, New York.
2. Bader M Space-filling curves – an introduction with applications in scientific computing, Texts in Computational Science and Engineering, Springer, submitted.

## Bibliography

1. Bader M, Schraufstetter S, Vigh CA, Behrens J (2008) Memory efficient adaptive mesh generation and implementation of multigrid algorithms using Sierpinski curves. *Int J Comput Sci Eng* 4(1):12–21
2. Bader M, Zenger Ch (2006) Cache oblivious matrix multiplication using an element ordering based on a Peano curve. *Linear Algebra Appl* 417(2–3):301–313
3. Barnes J, Hut P (1986) A hierarchical  $O(n \log n)$  force-calculation algorithm. *Nature* 324:446–449
4. Brázdová V, Bowler DR (2008) Automatic data distribution and load balancing with space-filling curves: implementation in conquest. *J Phy Condens Matt* 20
5. Brenk M, Bungartz H-J, Mehl M, Muntean IL, Neckel T, Weinzierl T (2008) Numerical simulation of particle transport in a drift ratchet. *SIAM J Sci Comput* 30(6):2777–2798
6. Griebel M, Zumbusch GW (1999) Parallel multigrid in an adaptive PDE solver based on hashing and space-filling curves. *Parallel Comput* 25:827–843

7. Günther F, Krahne A, Langlotz M, Mehl M, Pögl M, Zenger Ch (2004) On the parallelization of a cache-optimal iterative solver for PDES based on hierarchical data structures and space-filling curves. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users Group Meeting Budapest, Hungary, September 19–22, 2004. Proceedings, vol 3241 of Lecture Notes in Computer Science. Springer, Heidelberg
8. Günther F, Mehl M, Pögl M, Zenger C (2006) A cache-aware algorithm for PDEs on hierarchical data structures based on space-filling curves. *SIAM J Sci Comput* 28(5):1634–1650
9. Hunter GM, Steiglitz K (1979) Operations on images using quad trees. *IEEE Trans Pattern Anal Machine Intell PAMI-1(2)*: 145–154
10. Jackins C, Tanimoto SL (1980) Octrees and their use in representing three-dimensional objects. *Comp Graph Image Process* 14(31):249–270
11. Rokhlin V, Greengard L (1987) A fast algorithms for particle simulations. *J Comput Phys* 73:325–348
12. Meagher D (1980) Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3d objects by computer. Technical Report, IPL-TR-80-111
13. Mehl M, Weinzierl T, Zenger C (2006) A cache-oblivious self-adaptive full multigrid method. *Numer Linear Algebr* 13(2–3):275–291
14. Mitchell WF (2007) A refinement-tree based partitioning method for dynamic load balancing with adaptively refined grids. *J Parallel Distrib Comput* 67(4):417–429
15. Morton GM (1966) A computer oriented geodetic data base and a new technique in file sequencing. Technical Report, IBM Ltd., Ottawa, Ontario
16. Mundani R-P, Bungartz H-J, Niggli A, Rank E (2006) Embedding, organisation, and control of simulation processes in an octree-based cscw framework. In: Proceedings of the 11th International Conference on Computing in Civil and Building Engineering, Montreal, pp 3208–3215
17. Patra A, Oden JT (1995) Problem decomposition for adaptive hp finite element methods. *Comput Syst Eng* 6(2):97–109
18. Roberts S, Klyanasundaram S, Cardew-Hall M, Clarke W (1998) A key based parallel adaptive refinement technique for finite element methods. In: Proceedings of the Computational Techniques and Applications: CTAC '97, Singapore, pp 577–584
19. Samet H (1980) Region representation: quadtrees from binary arrays. *Comput Graph Image Process* 13(1):88–93
20. Saxena M, Finnigan PM, Graichen CM, Hathaway AF, Parthasarathy VN (1995) Octree-based automatic mesh generation for non-manifold domains. *Eng Comput* 11(1): 1–14
21. Schroeder WJ, Shephard MS (1988) A combined octree/delaunay method for fully automatic 3-d mesh generation. *Int J Numer Methods Eng* 26(1):37–55
22. Sundar H, Sampath RS, Biros G (2008) Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. *SIAM J Sci Comput* 30(5):2675–2708
23. Weinzierl T (2009) A framework for parallel PDE solvers on multiscale adaptive Cartesian grids. Verlag Dr. Hut
24. Zumbusch GW (2001) On the quality of space-filling curve induced partitions. *Z Angew Math Mech* 81:25–28

## SPAI (SParse Approximate Inverse)

THOMAS HUCKLE, MATOUS SEDLACEK  
Technische Universität München, Garching, Germany

### Synonyms

[Sparse approximate inverse matrix](#)

### Definition

For a given sparse matrix  $A$  a sparse matrix  $M \approx A^{-1}$  is computed by minimizing  $\|AM - I\|_F$  in the Frobenius norm over all matrices with a certain sparsity pattern. In the SPAI algorithm the pattern of  $M$  is updated dynamically to improve the approximation until a certain stopping criterion is reached.

### Discussion

#### Introduction

For applying an iterative solution method like the conjugate gradient method (CG), GMRES, BiCGStab, QMR, or similar algorithms, to a system of linear equations  $Ax = b$  with sparse matrix  $A$ , it is often crucial to include an efficient preconditioner. Here, the original problem  $Ax = b$  is replaced by the preconditioned system  $MAx = Mb$  or  $Ax = A(My) = b$ . In a parallel environment a preconditioner should satisfy the following conditions:

- $M$  can be computed efficiently in parallel.
- $Mc$  can be computed efficiently in parallel for any given vector  $c$ .
- The iterative solver applied on  $MAx = b$  or  $MAx = Mb$  converges much faster than for  $Ax = b$  (e.g., it holds  $cond(MA) \ll cond(A)$ ).

The first two conditions can be easily satisfied by using a sparse matrix  $M$  as approximation to  $A^{-1}$ . Note, that the inverse of a sparse  $A$  is nearly dense, but in many

cases the entries of  $A^{-1}$  are rapidly decaying, so most of the entries are very small [11].

Benson and Frederickson [4] were the first to propose a sparse approximate inverse preconditioner in a static way by computing

$$\min_M \|AM - I\|_F \quad (1)$$

for a prescribed a priori chosen sparsity pattern for  $M$ . The computation of  $M$  can be split into  $n$  independent subproblems  $\min_{M_k} \|AM_k - e_k\|_2$ ,  $k = 1, \dots, n$  with  $M_k$  the columns of  $M$  and  $e_k$  the  $k$ -th column of the identity matrix  $I$ . In view of the sparsity of these Least Squares (LS) problems, each subproblem is related to a small matrix  $\hat{A}_k := A(I_k, J_k)$  with index set  $J_k$  which is given by the allowed pattern for  $M_k$  and the so-called shadow  $I_k$  of  $J_k$ , that is, the indices of nonzero rows in  $A(:, J_k)$ . These  $n$  small LS problems can be solved independently, for example, based on QR decompositions of the matrices  $\hat{A}_k$  by using the Householder method or the modified Gram-Schmidt algorithm.

## The SPAI Algorithm

The SPAI algorithm is an additional feature in this Frobenius norm minimization that introduces different strategies for choosing new profitable indices in  $M_k$  that lead to an improved approximation. Assume that, by solving (1) for a given index set  $J$ , an optimal solution  $M_k(J_k)$  has been already determined resulting in the sparse vector  $M_k$  with residual  $r_k$ . Dynamically there will be defined new entries in  $M_k$ . Therefore, (1) has to be solved for this enlarged index set  $\tilde{J}_k$  such that a reduction in the norm of the new residual  $\tilde{r}_k = A(\tilde{I}_k, \tilde{J}_k)M_k(\tilde{J}_k) - e_k(\tilde{I}_k)$  is achieved.

Following Cosgrove, Griewank, Díaz [10], and Grote, Huckle [13], one possible new index  $j \in J_{new}$  out of a given set of possible new indices  $J_{new}$  is tested to improve  $M_k$ . Therefore, the reduced 1D problem

$$\min_{\lambda_j} \|A(M_k + \lambda_j e_j) - e_k\| = \min_{\lambda_j} \|\lambda_j A_j + r_k\| \quad (2)$$

has to be considered. The solution of this problem is given by

$$\lambda_j = -\frac{r_k^T A e_j}{\|A e_j\|^2}$$

which leads to an improved squared residual norm

$$\rho_j^2 = \|r_k\|^2 - \frac{(r_k^T A e_j)^2}{\|A e_j\|^2}.$$

Obviously, for improving  $M_k$  one has to consider only indices  $j$  in rows of  $A$  that are related to the nonzero entries in the old residual  $r_k$ ; otherwise they do not lead to a reduction in the residual norm. Thus, the column indices  $j$  have to be determined that satisfy  $r_k^T A e_j \neq 0$  with the old residual  $r_k$ . Let the index set of nonzero entries in  $r_k$  be denoted by  $L$ . Furthermore, let  $\tilde{J}_i$  denote the set of new indices that are related to the nonzero elements in the  $i$ -th row of  $A$ , and let  $J_{new} = \cup_{i \in L} \tilde{J}_i$  denote the set of all possible new indices that can lead to a reduction of the residual norm. Then, one or more indices  $J_c$  are chosen as a subset of  $J_{new}$  that corresponds to a large reduction in  $r_k$ . For this enlarged index set  $J_k \cup J_c$  the QR decomposition of the related LS submatrix has to be updated and solved for the new column  $M_k$ .

Inside SPAI there are different parameters that steer the computation of the preconditioner  $M$ :

- How many entries are added in one step
- How many steps of adding new entries are allowed
- Start pattern
- Maximum allowed pattern
- What residual  $\|r_k\|$  should be reached
- How to solve the LS problems

## Modifications of SPAI

A different and more expensive way to determine a new profitable index  $j$  with  $\tilde{J}_k := J_k \cup \{j\}$  considers the more accurate problem

$$\min_{M_k(\tilde{J}_k)} \|A(:, \tilde{J}_k)M_k(\tilde{J}_k) - e_k\|$$

introduced by Gould and Scott [12]. For  $\tilde{J}_k$  the optimal reduction of the residual is determined for the full minimization problem instead of the 1D minimization in SPAI.

Chow [9] showed ways to prescribe an efficient static pattern a priori and developed the software package PARASAILS.

Holland, Shaw, and Wathen [17] have generalized this ansatz allowing a sparse target matrix on the right side in the form  $\min_M \|AM - B\|_F$ . This approach is

useful in connection with some kind of two-level preconditioning: First compute a standard sparse preconditioner  $B$  for  $A$  and then improve this preconditioner by an additional Frobenius norm minimization with target  $B$ . From the algorithmic point of view the minimization with target matrix  $B$  instead of  $I$  introduces no additional difficulties. Only the pattern of  $M$  should be chosen more carefully with respect to  $A$  and  $B$ .

Zhang [23] introduced an iterative form of SPAI where in each step a thin  $M$  is derived starting with  $\min_{M_1} \|AM_1 - I\|_F$ . In the second step the sparse matrix  $AM_1$  is used and  $\min_{M_2} \|(AM_1)M_2 - I\|_F$  is solved, and so on. The advantage is, that because of the very sparse patterns in  $M$ , the Least Squares problems are very cheap.

Chan and Tang [8] applied SPAI not to the original matrix but first used a Wavelet transform  $W$  and computed the sparse approximate inverse preconditioner for  $WAW^T$  that is assumed to be more diagonal dominant.

Yeremin, Kolotilina, Nikishin, and Kaporin [19, 20] introduced factorized sparse approximate inverses of the form  $A^{-1} \approx LU$ . Huckle generalized the factorized preconditioners adding new entries dynamically like in SPAI [14].

Grote and Barnard [2] developed a software package for SPAI and also introduced a block version of SPAI.

Huckle and Kallischko [15] generalized SPAI and the target approach. They combined SPAI with the probing method [7] in the form

$$\min_M (\|AM - I\|_F^2 + \rho^2 \|e^T AM - e^T\|^2)$$

for probing vectors  $e$  on which the preconditioner should be especially improved. Furthermore, they developed a software package for MSPAI.

## Properties and Applications

Advantages of SPAI:

- Good parallel scalability.
- SPAI allows modifications like factorized approximation or including probing conditions to improve the preconditioner relative to certain subspaces, for example, as smoother in Multigrid or for regularization [16].
- It is especially efficient for preconditioning dense problems (see Benzi [1] et al.).

Disadvantages of SPAI:

- SPAI is sequentially more expensive, especially for denser patterns of  $M$ .
- Sometimes it shows poor approximation of  $A^{-1}$  and slow convergence as preconditioner.

## Related Entries

- Preconditioners for Sparse Iterative Methods

## Bibliographic Notes and Further Reading

### Books

1. Axelsson O (1996) Iterative solution methods. Cambridge University Press, Cambridge
2. Saad Y (2003) Iterative methods for sparse linear systems. SIAM Philadelphia, PA
3. Bruaset AM (1995) A survey of preconditioned iterative methods. Longman Scientific & Technical, Harlow, Essex
4. Chen K (2005) Matrix preconditioning techniques and applications. Cambridge University Press, Cambridge

### Software

1. Chow E. Parasails, <https://computation.llnl.gov/casc/parasails/parasails.html>
2. Barnard S, Bröker O, Grote M, Hagemann M. SPAI and Block SPAI, <http://www.computational.unibas.ch/software/spai>
3. Huckle T, Kallischko A, Sedlacek M. MSPAI, <http://www5.in.tum.de/wiki/index.php/MSPAI>

## Bibliography

1. Alleon G, Benzi M, Giraud L (1997) Sparse approximate inverse preconditioning for dense linear systems arising in computational electromagnetics. Numer Algorith 16(1):1–15
2. Barnard S, Grote M (1999) A block version of the SPAI preconditioner. Proceedings of the 9th SIAM conference on Parallel Processing for Scientific Computing, San Antonio, TX
3. Barnard ST, Clay RL (1997) A portable MPI implementation of the SPAI preconditioner in ISIS++. In: Heath M, et al (eds) Proceedings of the eighth SIAM conference on parallel processing for scientific computing, Philadelphia, PA

4. Benson MW, Frederickson PO (1982) Iterative solution of large sparse linear systems arising in certain multidimensional approximation problems. *Utilitas Math* 22:127–140
5. Bröker O, Grote M, Mayer C, Reusken A (2001) Robust parallel smoothing for multigrid via sparse approximate inverses. *SIAM J Scient Comput* 23(4):1396–1417
6. Bröker O, Grote M (2002) Sparse approximate inverse smoothers for geometric and algebraic multigrid. *Appl Num Math* 41(1): 61–80
7. Chan TFC, Mathew TP (1992) The interface probing technique in domain decomposition. *SIAM J Matrix Anal Appl* 13(1): 212–238
8. Chan TF, Tang WP, Wan WL (1997) Wavelet sparse approximate inverse preconditioners. *BIT* 37(3):644–660
9. Chow E (2000) A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM J Sci Comput* 21(5):1804–1822
10. Cosgrove JDF, D’iaz JC, Griewank A (1992) Approximate inverse preconditionings for sparse linear systems. *Int J Comput Math* 44:91–110
11. Demko S, Moss WF, Smith PW (1984) Decay rates of inverses of band matrices. *Math Comp* 43:491–499
12. Gould NIM, Scott JA (1995) On approximate-inverse preconditioners. Technical Report RAL-TR-95-026, Rutherford Appleton Laboratory, Oxfordshire, England
13. Grote MJ, Huckle T (1997) Parallel preconditioning with sparse approximate inverses. *SIAM J Sci Comput* 18(3):838–853
14. Huckle T (2003) Factorized sparse approximate inverses for preconditioning. *J Supercomput* 25:109–117
15. Huckle T, Kallischko A (2007) Frobenius norm minimization and probing for preconditioning. *Int J Comp Math* 84(8):1225–1248
16. Huckle T, Sedlacek M (2010) Smoothing and regularization with modified sparse approximate inverses. *Journal of Electrical and Computer Engineering – Special Issue on Iterative Signal Processing in Communications*, Appearing (2010)
17. Holland RM, Shaw GJ, Wathen AJ (2005) Sparse approximate inverses and target matrices. *SIAM J Sci Comput* 26(3):1000–1011
18. Kaporin IE (1994) New convergence results and preconditioning strategies for the conjugate gradient method. *Numer Linear Algebra Appl* 1:179–210
19. Kolotilina LY, Yeremin AY (1993) Factorized sparse approximate inverse preconditionings I: Theory. *SIAM J Matrix Anal Appl* 14(1):45–58
20. Kolotilina LY, Yeremin AY (1995) Factorized sparse approximate inverse preconditionings II: Solution of 3D FE systems on massively parallel computers. *Inter J High Speed Comput* 7(2): 191–215
21. Tang W-P (1999) Toward an effective sparse approximate inverse preconditioner. *SIAM J Matrix Anal Appl* 20(4):970–986
22. Tang WP, Wan WL (2000) Sparse approximate inverse smoother for multigrid. *SIAM J Matrix Anal Appl* 21(4):1236–1252
23. Zhang J (2002) A sparse approximate inverse technique for parallel preconditioning of general sparse matrices. *Appl Math Comput* 130(1):63–85

## Spanning Tree, Minimum Weight

DAVID A. BADER<sup>1</sup>, GUOJING CONG<sup>2</sup>

<sup>1</sup>Georgia Institute of Technology, Atlanta, GA, USA

<sup>2</sup>IBM, Yorktown Heights, NY, USA

### Definition

Given an undirected connected graph  $G$  with  $n$  vertices and  $m$  edges, the minimum-weight spanning tree (MST) problem consists in finding a spanning tree with the minimum sum of edge weights. A single graph can have multiple MSTs. If the graph is not connected, then it has a minimum spanning forest (MSF) that is a union of minimum spanning trees for its connected components. MST is one of the most studied combinatorial problems with practical applications in VLSI layout, wireless communication, and distributed networks, recent problems in biology and medicine such as cancer detection, medical imaging, and proteomics.

With regard to any MST of graph  $G$ , two properties hold: *Cycle property*: the heaviest edge (edge with the maximum weight) in any cycle of  $G$  does not appear in the MST. *Cut property*: if the weight of an edge  $e$  of any cut  $C$  of  $G$  is smaller than the weights of other edges of  $C$ , then this edge belongs to all MSTs of the graph.

When all edges of  $G$  are of unique weights, the MST is unique. When the edge weights are not unique, they can be made unique by numbering the edges and break ties using the edge number.

### Sequential Algorithms

Three classical sequential algorithms, Prim, Kruskal, and Borůvka, are known for MST. Each algorithm grows a forest in stages, adding at each stage one or more tree edges, whose membership in the MST is guaranteed by the cut property. They differ in how the tree edges are chosen and the order that they are added.

Prim starts with one vertex and takes a greedy approach in growing the tree. In each step, it always maintains a connected tree by choosing the edge of the smallest weight that connects the current tree to a vertex that is outside the tree.

Kruskal starts with isolated vertices. As the algorithm progresses, multiple trees may appear, and eventually they merge into one.

Borůvka selects for each vertex the incident edge of the smallest weight as a tree edge. It then compacts the graph by contracting each connected component into a super-vertex. Note that finding the tree edges can be done in parallel for each vertex. Borůvka's algorithm lends itself naturally to parallelization.

These algorithms can easily be made to run in  $O(m \log n)$  time. Better complexities can be achieved with Prim's algorithm if Fibonacci heap is used instead of binary heap.

Graham and Hell [14] gave a good introduction for the history of MST algorithms up to 1985. More complex algorithms with better asymptotic run times have since been proposed. For example, Gabow et al. designed an algorithm that runs in almost linear time, i.e.,  $O(m \log \beta(m, n))$ , where  $\beta(m, n) = \min\{i | \log^{(i)} n \leq m/n\}$ . Karger, Klein, and Tarjan presented a randomized linear-time algorithm to find minimum spanning trees. This algorithm uses the random sampling technique together with a linear time verification algorithm (e.g., King's verification algorithm). Pettie and Ramachandran presented an optimal MST algorithm that runs in  $O(T^*(m, n))$ , where  $T^*$  is the minimum number of edge-weight comparisons needed to determine the solution.

Moret and Shapiro [20] presented a comprehensive experimental study on the performance MST algorithms. Prim's algorithm with binary heap is found in general to be a fast solution. Katriel et al. [19] have developed a pipelined algorithm that uses the cycle property and provide an experimental evaluation on the special-purpose NEC SX-5 vector computer.

Cache-oblivious MST is presented by Arge et al. [2]. Their algorithm is based on a cache-oblivious priority queue that supports *insertion*, *deletion*, and *deletemin* operations in  $O\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$  amortized memory transfers, where  $M$  and  $B$  are the memory and block transfer sizes. The cache-oblivious implementation of MST runs in  $O(\text{sort}(m) \log \log(n/c) + c)$  memory transfers ( $c = m/B$ ). The MST algorithm combines two phases: Borůvka and Prim.

The memory access behavior of some MST algorithms are characterized in [11].

## Parallel Algorithms

Fast theoretical parallel MST algorithms also exist in the literature. Pettie and Ramachandran [21] designed a randomized, time-work optimal MST algorithm for the EREW PRAM, and using EREW to QSM and QSM to BSP emulations from [13], mapped the performance onto QSM and BSP models. Cole et. al. [7, 8] and Poon and Ramachandran [22] earlier had randomized linear-work algorithms on CRCW and EREW PRAM. Chong et al. [5] gave a deterministic EREW PRAM algorithm that runs in logarithmic time with a linear number of processors. On the BSP model, Adler et al. [1] presented a communication-optimal MST algorithm.

Most parallel and some fast sequential MST algorithms employ the Borůvka iteration. Three steps characterize a Borůvka iteration: *find-min*, *connect-components*, and *compact-graph*.

1. *find-min*: for each vertex  $v$  label the incident edge with the smallest weight to be in the MST.
2. *connect-components*: identify connected components of the induced graph with edges found in Step 1.
3. *compact-graph*: compact each connected component into a single supervertex, remove self-loops and multiple edges; and relabel the vertices for consistency.

The Borůvka algorithm iterates until no new tree edges can be found. After each Borůvka iteration, the number of vertices in the graph is reduced at least by half. Some algorithms invoke several rounds of the Borůvka iterations to reduce the input size (e.g., [1]) and/or to increase the edge density (e.g., [18]).

## Implementation of Parallel Borůvka

Many of the fast theoretic MST algorithms are considered impractical for input of realistic size because they are too complicated and have large constant factors hidden in the asymptotic complexity. Complex MST algorithms are hard to implement and usually do not achieve good parallel speedups on current architectures. Most existing implementations are based on the Borůvka algorithm.

For a Borůvka iteration, *Find-min* and *connect-components* are simple and straightforward to implement. The *compact-graph* step performs bookkeeping

that is often left as a trivial exercise to the reader. JáJá [16] describes a compact-graph algorithm for dense inputs. For sparse graphs, though, the compact-graph step often is the most expensive step in the Borůvka iteration. Implementations and data structures for parallel Borůvka on shared-memory machines are described in [3].

### Edge List Representation

This implementation of Borůvka's algorithm (designated **Bor-EL**) uses the edge list representation of graphs, with each edge  $(u, v)$  appearing twice in the list for both directions  $(u, v)$  and  $(v, u)$ . An elegant implementation of the compact-graph step sorts the edge list (using an efficient parallel sample sort [15]) with the supervertex of the first endpoint as the primary key, the supervertex of the second endpoint as the secondary key, and the edge weight as the tertiary key. When sorting completes, all of the self-loops and multiple edges between two supervertices appear in consecutive locations and can be merged efficiently using parallel prefix-sums.

### Adjacency List Representation

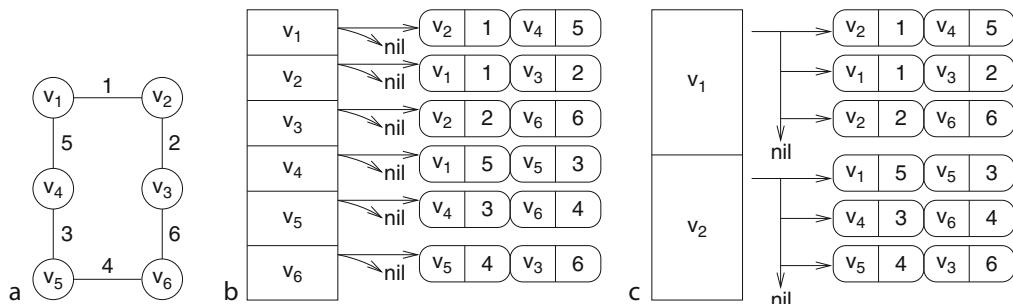
With the adjacency list representation, each entry of an index array of vertices points to a list of its incident edges. The compact-graph step first sorts the vertex array according to the supervertex label, then concurrently sorts each vertex's adjacency list using the supervertex of the other endpoint of the edge as the key. After sorting, the set of vertices with the same supervertex label are contiguous in the array, and can be merged efficiently. This approach is designated as **Bor-AL**.

Both **Bor-EL** and **Bor-AL** achieve the same goal that self-loops and multiple edges are moved to consecutive locations to be merged. **Bor-EL** uses one call to sample sort, while **Bor-AL** calls a smaller parallel sort and then a number of concurrent sequential sorts.

### Flexible Adjacency List Representation

The flexible adjacency list augments the traditional adjacency list representation by allowing each vertex to hold multiple adjacency lists instead of just a single one; in fact, it is a linked list of adjacency lists. During initialization, each vertex points to only one adjacency list. After the connect-components step, each vertex appends its adjacency list to its supervertex's adjacency list by sorting together the vertices that are labeled with the same supervertex. The compact-graph step is simplified, allowing each supervertex to have self-loops and multiple edges inside its adjacency list. Thus, the compact-graph step now uses a smaller parallel sort plus several pointer operations instead of costly sortings and memory copies, while the find-min step gets the added responsibility of filtering out the self-loops and multiple edges. This approach is designated as **Bor-FAL**.

Figure 1 illustrates the use of the flexible adjacency list for a 6-vertex input graph. After one Borůvka iteration, vertices 1, 2, and 3 form one supervertex, and vertices 4, 5, and 6 form a second supervertex. Vertex labels 1 and 4 represent the supervertices and receive the adjacency lists of vertices 2 and 3, and vertices 5 and 6, respectively. Vertices 1 and 4 are relabeled as 1 and 2. Note that most of the original data structure is kept intact. Instead of relabeling vertices in the adjacency list, a separate lookup table is maintained that holds the



**Spanning Tree, Minimum Weight. Fig. 1** Example of flexible adjacency list representation. (a) Input graph. (b) Initialized flexible adjacency list. (c) Flexible adjacency list after one iteration

supervertex label for each vertex. The find-min step uses this table to filter out self-loops and multiple edges.

## Analysis of Implementations

Helman and Jájá's SMP complexity model [15] provides a reasonable framework for the realistic analysis that favors cache-friendly algorithms by penalizing noncontiguous memory accesses. Under this model, there are two parts to an algorithm's complexity:  $M_E$ , the memory access complexity and,  $T_C$ , the computation complexity. The  $M_E$  term is the number of noncontiguous memory accesses, and the  $T_C$  term is the running time. The  $M_E$  term recognizes the effect that memory accesses have over an algorithm's performance. Parameters of the model includes the problem size  $n$  and the number of processors  $p$ .

For a sparse graph  $G$  with  $n$  vertices and  $m$  edges, as the algorithm iterates, the number of vertices decreases by at least half in each iteration, so there are at most  $\log n$  iterations for all of the Borůvka variants.

Hence, the complexity of **Bor-EL** is given as, where  $c$  and  $z$  are constants related to cache size and sampling ratio [15].

$$\begin{aligned} T(n, p) &= \langle M_E ; T_C \rangle \\ &= \left\langle \left( \frac{8m + n + n \log n}{p} + \frac{4mc \log(2m/p)}{p \log z} \right) \log n ; \right. \\ &\quad \left. O\left(\frac{m}{p} \log m \log n\right) \right\rangle. \end{aligned}$$

As in each iteration these **Bor-AL** and **Bor-EL** compute similar results in different ways, it suffices to compare the complexity of the first iteration. For **Bor-AL**, the complexity of the first iteration is

$$\begin{aligned} T(n, p) &= \langle M_E; T_C \rangle \\ &= \left\langle \left( \frac{8n + 5m + n \log n}{p} \right. \right. \\ &\quad \left. \left. + \frac{2nc \log(n/p) + 2mc \log(m/n)}{p \log z} \right) ; \right. \\ &\quad \left. O\left(\frac{n}{p} \log m + \frac{m}{p} \log(m/n)\right) \right\rangle. \end{aligned}$$

While for **Bor-EL**, the complexity of the first iteration is

$$\begin{aligned} T(n, p) &= \langle M_E; T_C \rangle \\ &= \left\langle \left( \frac{8m + n + n \log n}{p} + \frac{4mc \log(2m/p)}{p \log z} \right); \right. \\ &\quad \left. O\left(\frac{m}{p} \log m\right) \right\rangle. \end{aligned}$$

**Bor-AL** is a faster algorithm than **Bor-EL**, as expected, since the input for **Bor-AL** is “bucketed” into adjacency lists, versus **Bor-EL** that is an unordered list of edges, and sorting each bucket first in **Bor-AL** saves unnecessary comparisons between edges that have no vertices in common. The complexity of **Bor-EL** can be considered to be an upper bound of **Bor-AL**.

In **Bor-FAL**,  $n$  reduces at least by half while  $m$  stays the same. Compact-graph first sorts the  $n$  vertices, then assigns  $O(n)$  pointers to append each vertex's adjacency list to its supervertex's. For each processor, sorting takes  $O\left(\frac{n}{p} \log n\right)$  time, and assigning pointers takes  $O(n/p)$  time assuming each processor gets to assign roughly the same amount of pointers. Updating the lookup table costs each processor  $O(n/p)$  time. With **Bor-FAL**, to find the smallest weight edge for the supervertices, all the  $m$  edges will be checked, with each processor covering  $O(m/p)$  edges. The aggregate running time is  $T_C(n, p)_{fm} = O(m \log n/p)$ , and the memory access complexity is  $M_E(n, p)_{fm} = m/p$ . For the finding connected component step, each processor takes  $T_{cc} = O\left(n \log \frac{n}{p}\right)$  time, and  $M_E(n, p)_{cc} \leq 2n \log n$ . The complexity for the whole Borůvka's algorithm is

$$\begin{aligned} T(n, p) &= T(n, p)_{fm} + T(n, p)_{cc} + T(n, p)_{cg} \\ &\leq \left\langle \frac{8n + 2n \log n + m \log n}{p} + \frac{4cn \log(n/p)}{p \log z} ; \right. \\ &\quad \left. O\left(\frac{m+n}{p} \log n\right) \right\rangle \end{aligned}$$

## A Hybrid Parallel MST Algorithm

An MST algorithm has been proposed in [3] that marries Prim's algorithm (known as an efficient sequential algorithm for MST) with that of the naturally parallel Borůvka approach. In this algorithm, essentially each processor simultaneously runs Prim's algorithm from different starting vertices. A tree is said to be *growing* when there exists a lightweight edge that connects the tree to a vertex not yet in another tree, and *mature* otherwise. When all of the vertices have been incorporated into mature subtrees, the algorithm contracts each subtree into a supervertex and call the approach recursively until only one supervertex remains. When the problem size is small enough, one processor solves the remaining problem using the best sequential MST algorithm.

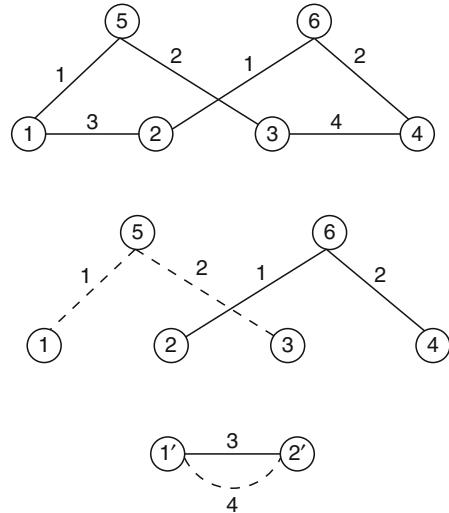
This parallel MST algorithm possesses an interesting feature: when run on one processor the algorithm behaves as Prim's, and on  $n$  processors becomes Boruvka's, and runs as a hybrid combination for  $1 < p < n$ , where  $p$  is the number of processors. Each of  $p$  processors in the algorithm finds for its starting vertex the smallest-weight edge, contracts that edge, and then finds the smallest-weight edge again for the contracted supervertex. It does not find all the smallest-weight edges for all vertices, synchronize, and then compact as in the parallel Boruvka's algorithm. The algorithm adapts for any number  $p$  of processors in a practical way for SMPs, where  $p$  is often much less than  $n$ , rather than in parallel implementations of Boruvka's approach that appear as PRAM emulations with  $p$  coarse-grained processors that emulate  $n$  virtual processors.

### Implementation with Fine-Grained Locks

Most parallel graph algorithms are designed without locks. Indeed it is hard to measure contention for these algorithms. Yet proper use of locks can simplify implementation and improve performance. Cong and Bader [10] presented an implementation of Boruvka's algorithm (**Bor-spinlock**) that uses locks and avoids modifying the input data structure. In **Bor-spinlock**, the *compact-graph* step is completely eliminated.

The main idea is that instead of compacting connected components, for each vertex there is now an associated label *supervertex* showing to which supervertex it belongs. In each iteration, all the vertices are partitioned as evenly as possible among the processors. Processor  $p$  finds the adjacent edge with smallest weight for a supervertex  $v'$ . As the graph is not compacted, the adjacent edges for  $v'$  are scattered among the adjacent edges of all vertices that share the same supervertex  $v'$ , and different processors may work on these edges simultaneously. Now the problem is that these processors need to synchronize properly in order to find the edge with the minimum weight. Figure 2 illustrates the specific problem for the MST case.

On the top in Fig. 2 is an input graph with six vertices. Suppose there are two processors  $P_1$  and  $P_2$ . Vertices 1, 2, and 3 are partitioned on to processor  $P_1$ , and vertices 4, 5, and 6 are partitioned on to processor  $P_2$ . It takes two iterations for Boruvka's algorithm to find the MST. In the first iteration, the *find-min* step of **Bor-spinlock** labels  $< 1, 5 >$ ,  $< 5, 3 >$ ,  $< 2, 6 >$ , and  $< 6, 4 >$ ,



**Spanning Tree, Minimum Weight.** Fig. 2 Example of the race condition between two processors when Boruvka's algorithm is used to solve the MST problem

to be in the MST. *Connected-components* finds vertices 1, 3, and 5 in one component, and vertices 2, 4, and 6 in another component. The MST edges and components are shown in the middle of Fig. 2. Vertices connected by dashed lines are in one component, and vertices connected by solid lines are in the other component. At this time, vertices 1, 3, and 5 belong to supervertex 1', and vertices 2, 4, and 6 belong to supervertex 2'. In the second iteration, processor  $P_1$  again inspects vertices 1, 2, and 3, and processor  $P_2$  inspects vertices 4, 5, and 6. Previous MST edges  $< 1, 5 >$ ,  $< 5, 3 >$ ,  $< 2, 6 >$ , and  $< 6, 4 >$  are found to be edges inside supervertices and are ignored. On the bottom of Fig. 2 are the two supervertices with two edges between them. Edges  $< 1, 2 >$  and  $< 3, 4 >$  are found by  $P_1$  to be the edges between supervertices 1' and 2', edge  $< 3, 4 >$  is found by  $P_2$  to be the edge between the two supervertices. For supervertex 2',  $P_1$  tries to label  $< 1, 2 >$  as the MST edge, while  $P_2$  tries to label  $< 3, 4 >$ . This is a race condition between the two processors, and locks are used in **Bor-spinlock** to ensure correctness.

In addition to locks and barriers, recent development in transactional memory provides a new mechanism for synchronization among processors. Kang and Bader implemented minimum spanning forest algorithms with transactional memory [17]. Their

implementation achieved good scalability on some current architectures.

## Implementation on Distributed-Memory Machines

Partitioned global address space (PGAS) languages such as UPC and X10 [4, 23] have been proposed recently that present a shared-memory abstraction to the programmer for distributed-memory machines. They allow the programmer to control the data layout and work assignment for the processors. Mapping shared-memory graph algorithms onto distributed-memory machines is straightforward with PGAS languages.

Figure 3 shows both the SMP implementation and UPC implementation of parallel Boruvka. The two implementations are also almost identical. The differences are shown in underscore. Performance wise, straightforward PGAS implementation for irregular graph algorithms does not usually achieve high performance due to the aggregate startup cost of many small messages. Cong, Almasi, and Saraswat presented

their study in optimizing the UPC implementation of graph algorithm in [9]. They apply communication coalescing together with other techniques for improving the performance. The idea is to merge the small messages to/from a processor into a single, large message. As all operations in each step of a typical PRAM algorithm are parallel, reads and writes can be scheduled in an order such that communication coalescing is possible. After communication coalescing, these data can be accessed in one communication round where one processor sends at most one message to another processor.

## Experimental Results

Chung and Condon [6] implement parallel Boruvka's algorithm on the TMC CM-5. On a 16-processor machine, for geometric, structured graphs with 32,000 vertices and average degree 9 and graphs with fewer vertices but higher average degree, their code achieves a relative parallel speedup of about 4, on 16-processors, over the sequential Boruvka's algorithm, which was already 2–3 times slower than their sequential Kruskal

```

grafted = 0;
ppc_forall(i=0;i<m; i++)
{
 i = E1[1].v1; w = E1[1].w;
 j = E1[1].v2;
 i = D[i]; j = D[j];
 if (i!=j){
 upc_lock(lock_array[i]);
 if (Min[i] > w) {
 Min[i] = w;
 Min_ind[i] = j;
 grafted = 1;
 }
 upc_unlock(lock_array[i]);
 }
 upc_barrier;
 grafted = all_reduce_i(grafted , UPC_MAX);
 if (grafted ==0) break;

 upc_forall(i=0;i<n; i++)
 if (Min_ind[i]!= -1)
 D[i]=Min_ind[i];
 upc_barrier;

 upc_forall(i=0; i<n; i++)
 while(D[i]!=D[D[i]]) D[i]=D[D[i]];
}

```

```

grafted = 0;
pardo(1,0,m,1)
{
 i = E1[1].v1; w = E1[1].w;
 j = E1[1].v2;
 i = D[i]; j = D[j];
 if (i!=j){
 pthread_lock(lock_array[i]);
 if (Min[i]>w) {
 Min[i] = w;
 Min_ind[i]=j;
 grafted=1;
 }
 pthread_unlock(lock_array[i]);
 }
 node_barrier();
 grafted = node_Reduce_i(grafted , MAX, TH);
 if (grafted==0) break;

 pardo(i,0,n,1)
 if (Min_ind[i]!= -1)
 D[i]=Min_ind[i];
 node_Barrier();

 pardo(i,0,n,1)
 while(D[i]!=D[D[i]]) D[i]=D[D[i]];
}

```

**Spanning Tree, Minimum Weight. Fig. 3** UPC implementation and SMP implementation of MST: the main loop bodies

algorithm. Dehne and Götz [12] studied practical parallel algorithms for MST using the BSP model. They implement a dense Borůvka parallel algorithm, on a 16-processor Parsytec CC-48, that works well for sufficiently dense input graphs. Using a fixed-sized input graph with 1,000 vertices and 400,000 edges, their code achieves a maximum speedup of 6.1 using 16 processors for a random dense graph. Their algorithm is not suitable for the more challenging sparse graphs.

Bader and Cong presented their studies of parallel MST on symmetric multiprocessors (SMPs) in [3, 10]. Their implementation achieved for the first time good parallel speedups over a wide range of inputs on SMPs. Their Experimental results show that for **Bor-EL** and **Bor-AL** the compact-graph step dominates the running time. **Bor-EL** takes much more time than **Bor-AL**, and only gets worse when the graphs get denser. In contrast the execution time of compact-graph step of **Bor-FAL** is greatly reduced: in the experimental section with a random graph of 1M vertices and 10M edges, it is over 50 times faster than **Bor-EL**, and over 7 times faster than **Bor-AL**. Actually the execution time of the compact-graph step of **Bor-FAL** is almost the same for the three input graphs because it only depends on the number of vertices. As predicted, the execution time of the find-min step of **Bor-FAL** increases. And the connect-components step only takes a small fraction of the execution time for all approaches.

Cong, Almasi and Saraswat presented a UPC implementation of distributed MST in [9]. For input graphs with billions of edges, the distributed implementation achieved significant speedups over the SMP implementation and the best sequential implementation.

## Bibliography

1. Adler M, Dittrich W, Juurlink B, Kutylowski M, Rieping I (1998) Communication-optimal parallel minimum spanning tree algorithms (extended abstract). In: SPAA '98: proceedings of the tenth annual ACM symposium on parallel algorithms and architectures, Puerto Vallarta, Mexico. ACM, New York, pp 27–36
2. Arge L, Bender MA, Demaine ED, Holland-Minkley B, Munro JI (2002) Cache-oblivious priority queue and graph algorithm applications. In: Proceedings of the 34th annual ACM symposium on theory of computing, Montreal, Canada. ACM, New York, pp 268–276
3. Bader DA, Cong G (2006) Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *J Parallel Distrib Comput* 66:1366–1378
4. Charles P, Donawa C, Ebcioğlu K, Grothoff C, Kielstra A, Van Praun C, Saraswat V, Sarkar V (2005) X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of the 2005 ACM SIGPLAN conference on object-oriented programming systems, languages and applications (OOPSLA), San Diego, CA, pp 519–538
5. Chong KW, Han Y, Lam TW (2001) Concurrent threads and optimal parallel minimum spanning tree algorithm. *J ACM* 48: 297–323
6. Chung S, Condon A (1996) Parallel implementation of Borůvka's minimum spanning tree algorithm. In: Proceedings of the 10th international parallel processing symposium (IPPS'96), Honolulu, Hawaii, pp 302–315
7. Cole R, Klein PN, Tarjan RE (1996) Finding minimum spanning forests in logarithmic time and linear work using random sampling. In: Proceedings of the 8th annual symposium parallel algorithms and architectures (SPAA-96), Newport, RI. ACM, New York, pp 243–250
8. Cole R, Klein PN, Tarjan RE (1994) A linear-work parallel algorithm for finding minimum spanning trees. In: Proceedings of the 6th annual ACM symposium on parallel algorithms and architectures, Cape May, NJ, ACM, New York, pp 11–15
9. Cong G, Almasi G, Saraswat V (2010) Fast PGAS implementation of distributed graph algorithms. In: Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis (SC '10), IEEE Computer Society, Washington, DC, pp 1–11
10. Cong G, Bader DA (2004) Lock-free parallel algorithms: an experimental study. In: Proceeding of the 33rd international conference on high-performance computing (HiPC 2004), Bangalore, India
11. Cong G, Sbaraglia S (2006) A study of the locality behavior of minimum spanning tree algorithms. In: The 13th international conference on high performance computing (HiPC 2006), Bangalore, India. IEEE Computer Society, pp 583–594
12. Dehne F, Götz S (1998) Practical parallel algorithms for minimum spanning trees. In: Proceedings of the seventeenth symposium on reliable distributed systems, West Lafayette, IN. IEEE Computer Society, pp 366–371
13. Gibbons PB, Matias Y, Ramachandran V (1997) Can shared-memory model serve as a bridging model for parallel computation? In: Proceedings 9th annual symposium parallel algorithms and architectures (SPAA-97), Newport, RI, ACM, New York pp 72–83
14. Graham RL, Hell P (1985) On the history of the minimum spanning tree problem. *IEEE Ann History Comput* 7(1):43–57
15. Helman DR, JáJá J (1999) Designing practical efficient algorithms for symmetric multiprocessors. In: Algorithm engineering and experimentation (ALENEX'99), Baltimore, MD, Lecture notes in computer science, vol 1619. Springer-Verlag, Heidelberg, pp 37–56
16. JáJá J (1992) An Introduction to parallel algorithms. Addison-Wesley, New York
17. Kang S, Bader DA (2009) An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. In: Proceedings of the 14th ACM SIGPLAN symposium on principles and practice of parallel programming (PPoPP), Raleigh, NC

18. Karger DR, Klein PN, Tarjan RE (1995) A randomized linear-time algorithm to find minimum spanning trees. *J ACM* 42(2):321–328
19. Katriel I, Sanders P, Träff JL (2003) A practical minimum spanning tree algorithm using the cycle property. In: 11th Annual European symposium on algorithms (ESA 2003), Budapest, Hungary, Lecture notes in computer science, vol 2832. Springer-Verlag, Heidelberg, pp 679–690
20. Moret BME, Shapiro HD (1994) An empirical assessment of algorithms for constructing a minimal spanning tree. In: DIMACS monographs in discrete mathematics and theoretical computer science: computational support for discrete mathematics vol 15, American Mathematical Society, Providence, RI, pp 99–117
21. Pettie S, Ramachandran V (2002) A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM J Comput* 31(6):1879–1895
22. Poon CK, Ramachandran V (1997) A randomized linear work EREW PRAM algorithm to find a minimum spanning forest. In: Proceedings of the 8th international symposium algorithms and computation (ISAAC'97), Lecture notes in computer science, vol 1350. Springer-Verlag, Heidelberg, pp 212–222
23. Carlson WW, Draper JM, Culler DE, Yelick K, Brooks E, Warren K (1999) Introduction to UPC and Language Specification. CCS-TR-99-157. IDA/CCS, Bowie, Maryland

## Sparse Approximate Inverse Matrix

► [SPAI \(SParse approximate inverse\)](#)

## Sparse Direct Methods

ANSHUL GUPTA

IBM T.J. Watson Research Center, Yorktown Heights, NY, USA

### Synonyms

[Gaussian elimination](#); [Linear equations solvers](#); [Sparse gaussian elimination](#)

### Definition

Direct methods for solving linear systems of the form  $Ax = b$  are based on computing  $A = LU$ , where  $L$  and  $U$  are lower and upper triangular, respectively. Computing the triangular factors of the coefficient matrix  $A$  is also known as *LU decomposition*. Following the factorization, the original system is trivially solved by solving

the triangular systems  $Ly = b$  and  $Ux = y$ . If  $A$  is symmetric, then a factorization of the form  $A = LL^T$  or  $A = LDL^T$  is computed via *Cholesky factorization*, where  $L$  is a lower triangular matrix (unit lower triangular in the case of  $A = LDL^T$  factorization) and  $D$  is a diagonal matrix. One set of common formulations of LU decomposition and Cholesky factorization for dense matrices are shown in Figs. 1 and 2, respectively. Note that other mathematically equivalent formulations are possible by rearranging the loops in these algorithms. These algorithms must be adapted for sparse matrices, in which a large fraction of entries are zero. For example, if  $A[j, i]$  in the division step is zero, then this operation need not be performed. Similarly, the update steps can be avoided if either  $A[j, i]$  or  $A[i, k]$  ( $A[k, i]$  if  $A$  is symmetric) is zero.

When  $A$  is sparse, the triangular factors  $L$  and  $U$  typically have nonzero entries in many more locations than  $A$  does. This phenomenon is known as *fill-in*, and results in a superlinear growth in the memory and time requirements of a direct method to solve a sparse system with respect to the size of the system. Despite a high memory requirement, direct methods are often used in many real applications due to their generality and robustness. In applications requiring solutions with respect to several right-hand side vectors and the same coefficient matrix, direct methods are often the solvers of choice because the one-time cost of factorization can be amortized over several inexpensive triangular solves.

### Discussion

The direct solution of a sparse linear system typically involves four phases. The two computational phases, *factorization* and *triangular solutions* have already been mentioned. The number of nonzeros in the factors and sometimes their numerical properties are functions of the initial permutation of the rows and columns of the coefficient matrix. In many parallel formulations of sparse factorization, this permutation can also have an effect on load balance. The first step in the direct solution of a sparse linear system, therefore, is to apply heuristics to compute a desirable permutation the matrix. This step is known as *ordering*. A sparse matrix can be viewed as the adjacency matrix of a graph. Ordering heuristics typically use the graph view of the matrix

```

1. begin LU_Decompon (A, n)
2. for $i = 1, n$
3. for $j = i + 1, n$
4. $A[j, i] = A[j, i]/A[i, i]$; /* division step, computes column i of L */
5. end for
6. for $k = i + 1, n$
7. for $j = i + 1, n$
8. $A[j, k] = A[j, k] - A[j, i] \times A[i, k]$; /* update step*/
9. end for
10. end for
11. end for
12. end LU_Decompon

```

**Sparse Direct Methods.** Fig. 1 A simple column-based algorithm for LU decomposition of an  $n \times n$  dense matrix  $A$ . The algorithm overwrites  $A$  by  $L$  and  $U$  such that  $A = LU$ , where  $L$  is unit lower triangular and  $U$  is upper triangular. The diagonal entries after factorization belong to  $U$ ; the unit diagonal of  $L$  is not explicitly stored

```

1. begin Cholesky (A, n)
2. for $i = 1, n$
3. $A[i, i] = \sqrt{A[i, i]}$;
4. for $j = i + 1, n$
5. $A[j, i] = A[j, i]/A[i, i]$; /* division step, computes column i of L */
6. end for
7. for $k = i + 1, n$
8. for $j = k, n$
9. $A[j, k] = A[j, k] - A[j, i] \times A[k, i]$; /* update step*/
10. end for
11. end for
12. end for
13. end Cholesky

```

**Sparse Direct Methods.** Fig. 2 A simple column-based algorithm for Cholesky factorization of an  $n \times n$  dense symmetric positive definite matrix  $A$ . The lower triangular part of  $A$  is overwritten by  $L$ , such that  $A = LL^T$

and label the vertices in a particular order that is equivalent to computing a permutation of the coefficient matrix with desirable properties. In the second phase, known as *symbolic factorization*, the nonzero pattern of the factors is computed. Knowing the nonzero pattern of the factors before actually computing them is useful for several reasons. The memory requirements of numerical factorization can be predicted during symbolic factorization. With the number and locations of nonzeros known before hand, a significant amount of indirect addressing can be avoided during numerical factorization, thus boosting performance. In a parallel implementation, symbolic factorization helps in the distribution of data and computation among processing units. The ordering and symbolic factorization phases are also referred to as preprocessing or analysis steps.

Of the four phases, numerical factorization typically consumes the most memory and time. Many applications involve factoring several matrices with different numerical values but the same sparsity structure. In such cases, some or all of the results of the ordering and symbolic factorization steps can be reused. This is also advantageous for parallel sparse solvers because parallel ordering and symbolic factorization are typically less scalable. Amortization of the cost of these steps over several factorization steps helps maintain the overall scalability of the solver close to that of numerical factorization. The parallelization of the triangular solves is highly dependent on the parallelization of the numerical factorization phase. The parallel formulation of numerical factorization dictates how the factors are distributed among parallel tasks. The subsequent triangular solution steps must use a parallelization scheme

that works on this data distribution, particularly in a distributed-memory parallel environment. Given its prominent role in the parallel direct solution of sparse linear system, the numerical factorization phase is the primary focus of this entry.

The algorithms used for preprocessing and factoring a sparse coefficient matrix depend on the properties of the matrix, such as symmetry, diagonal dominance, positive definiteness, etc. However, there are common elements in most sparse factorization algorithms. Two of these, namely, *task graphs* and *supernodes*, are key to the discussion of parallel sparse matrix factorization of all types for both practical and pedagogical reasons.

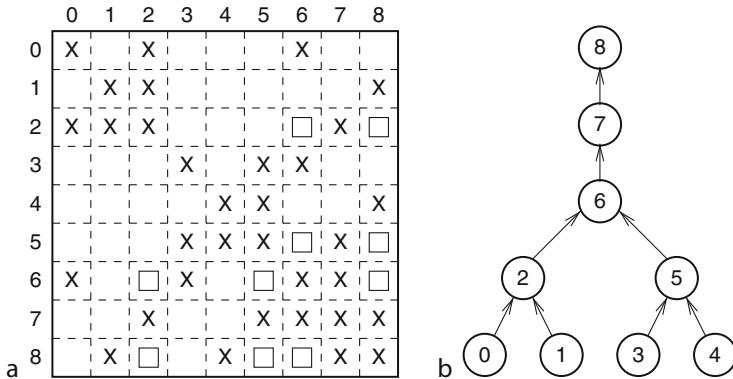
### Task Graph Model of Sparse Factorization

A parallel computation is usually the most efficient when running at the maximum possible level of granularity that ensures a good load balance among all the processors. Dense matrix factorization is computationally rich and requires  $O(n^3)$  operations for factoring an  $n \times n$  matrix. Sparse factorization involves a much smaller overall number of operations per row or column of the matrix than its dense counterpart. The sparsity results in additional challenges, as well as additional opportunities to extract parallelism. The challenges are centered around finding ways of orchestrating the unstructured computations in a load-balanced fashion and of containing the overheads of interaction between parallel tasks in the face of a relatively small number of operations per row or column of the matrix. The added opportunity for parallelism results from the fact that, unlike the dense algorithms of Figs. 1 and 2, the columns of the factors in the sparse case do not need to be computed one after the other. Note that in the algorithms shown in Figs. 1 and 2, row and column  $i$  are updated by rows and columns  $1 \dots i-1$ . In the sparse case, column  $i$  is updated by a column  $j < i$  only if  $U[j, i] \neq 0$ , and a row  $i$  is updated by a row  $j < i$  only if  $L[i, j] \neq 0$ . Therefore, as the sparse factorization begins, the division step can proceed in parallel for all columns  $i$  for which  $A[i, j] = 0$  and  $A[j, i] = 0$  for all  $j < i$ . Similarly, at any stage in the factorization process, there could be large pool of columns that are ready of the division step. Any unfactored column  $i$  would belong to this pool iff all columns  $j < i$  with a nonzero entry in row

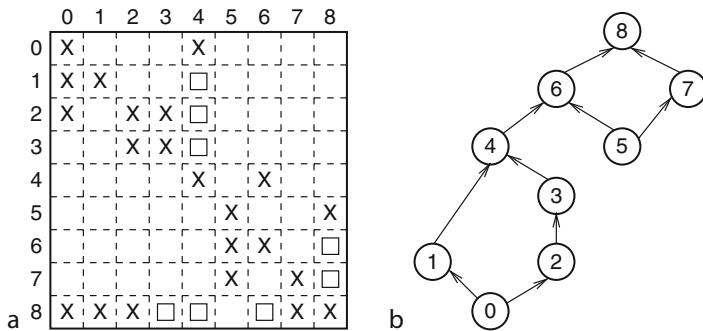
$i$  of  $L$  and all rows  $j < i$  with a nonzero entry in column  $i$  of  $U$  have been factored.

A task dependency graph is an excellent tool for capturing parallelism and the various dependencies in sparse matrix factorization. It is a directed acyclic graph (DAG) whose vertices denote tasks and the edges specify the dependencies among the tasks. A task is associated with each row and column (column only in the symmetric case) of the sparse matrix to be factored. The vertex  $i$  of the task graph denotes the task responsible for computing column  $i$  of  $L$  and row  $i$  of  $U$ . A task is ready for execution if and only if all tasks with incoming edges to it have completed. Task graphs are often explicitly constructed during symbolic factorization to guide the numerical factorization phase. This permits the numerical factorization to avoid expensive searches in order to determine which tasks are ready for execution at any given stage of the parallel factorization process. The task graphs corresponding to matrices with a symmetric structure are trees and are known as *elimination trees* in the sparse matrix literature.

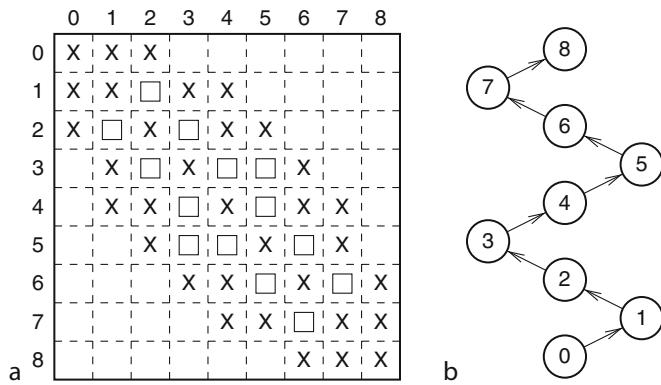
Figure 3 shows the elimination tree for a structurally symmetric sparse matrix and Fig. 4 shows the task DAG for a structurally unsymmetric matrix. Once a task graph is constructed, then parallel factorization (and even parallel triangular solution) can be viewed as the problem of scheduling the tasks onto parallel processes or threads. Static scheduling is generally preferred in a distributed-memory environment and dynamic scheduling in a shared-memory environment. The shape of the task graph is a function of the initial permutation of rows and columns of the sparse matrix, and is therefore determined by the outcome of the ordering phase. Figure 5 shows the elimination tree corresponding to the same matrix as in Fig. 3a, but with a different initial permutation. The structure of the task DAG usually affects how effectively it can be scheduled for parallel factorization. For example, it may be intuitively recognizable to readers that the elimination tree in Fig. 3 is more amenable to parallel scheduling than the tree corresponding to a different permutation of the same matrix in Fig. 5. Figure 6 illustrates that the matrices in Figs. 3 and 5 have the same underlying graph. The only difference is in the labeling of the vertices of the graph, which results in a different permutation of the rows and columns of the matrix, different amount



**Sparse Direct Methods.** Fig. 3 A structurally symmetric sparse matrix and its elimination tree. An X indicates a nonzero entry in the original matrix and a box denotes a fill-in



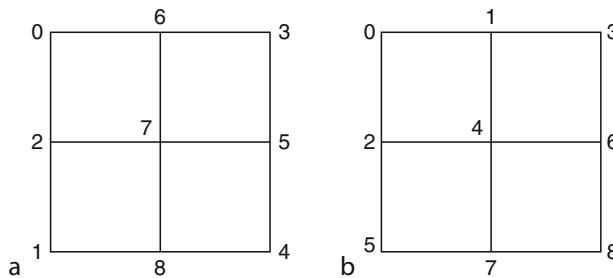
**Sparse Direct Methods.** Fig. 4 An unsymmetric sparse matrix and the corresponding task DAG. An X indicates a nonzero entry in the original matrix and a box denotes a fill-in



**Sparse Direct Methods.** Fig. 5 A permutation of the sparse matrix of Fig. 3a and its elimination tree

of fill-in, and different shapes of task graphs. In general, long and skinny task graphs result in limited parallelism and a long critical path. Short and broad task graphs have a high degree of parallelism and shorter critical paths.

Since the shape, and hence the amenability to efficient parallel scheduling of the task graph is sensitive to ordering, heuristics that result in balanced and broad task graphs are preferred for parallel factorization. The best known ordering heuristic in this class is called



**Sparse Direct Methods.** Fig. 6 An illustration of the duality between graph vertex labeling and row/column permutation of a structurally symmetric sparse matrix. Grid (a), with vertices labeled based on nested dissection, is the adjacency graph of the matrix in Fig. 3a and grid (b) is the adjacency graph of the matrix in Fig. 5a

*nested dissection.* Nested dissection is based on recursively computing balanced bisections of a graph by finding small vertex separators. The vertices in the two disconnected partitions of the graph are labeled before the vertices of the separator. The same heuristic is applied recursively for labeling the vertices of each partition. The ordering in Fig. 3 is actually based on nested dissection. Note that the vertex set 6, 7, 8 forms a separator, dividing the graph into two disconnected components, 0, 1, 2 and 3, 4, 5. Within the two components, vertices 2 and 5 are the separators, and hence have the highest label in their respective partitions.

## Supernodes

In sparse matrix terminology, a set of consecutive rows or columns that have the same nonzero structure is loosely referred to as a supernode. The notion of supernodes is crucial to efficient implementation of sparse factorization for a large class of sparse matrices arising in real applications.

Coefficient matrices in many applications have natural supernodes. In graph terms, there are sets of vertices with identical adjacency structures. Graphs like these can be compressed by having one supervertex represent the whole set that has the same adjacency structure. When most vertices of a graph belong to supernodes and the supernodes are of roughly the same size (in terms of the number of vertices in them) with an average size of, say,  $m$ , then it can be shown that the compressed graph has  $O(m)$  fewer vertices and  $O(m^2)$  fewer edges than in the original graph. It can also be shown that an ordering of original graph can be derived from an ordering of the compressed graph,

while preserving the properties of the ordering, by simply labeling the vertices of the original graph consecutively in the order of the supernodes of the compressed graph. Thus, the space and the time requirements of ordering can be dramatically reduced. This is particularly useful for parallel sparse solvers because parallel ordering heuristics often yield orderings of lower quality than their serial counterparts. For matrices with highly compressible graphs, it is possible to compute the ordering in serial with only a small impact on the overall scalability of the entire solver because ordering is performed on a graph with  $O(m^2)$  fewer edges.

While the natural supernodes in the coefficient matrix, if any, can be useful during ordering, it is the presence of supernodes in the factors that have the biggest impact on the performance of the factorization and triangular solution steps. Although there can be multiple ways of defining supernodes in matrices with an unsymmetric structure, the most useful form involves groups of indices with identical nonzero pattern in the corresponding columns of  $L$  and rows of  $U$ . Even if there are no supernodes in the original matrix, supernodes in the factors are almost inevitable for matrices in most real applications. This is due to fill-in. Examples of supernodes in factors include indices 6–8 in Fig. 3a, indices 2–3 and 7–8 in Fig. 4a, and indices 4–5 and 6–8 in Fig. 5. Some practitioners prefer to artificially increase the size (i.e., the number of member rows and columns) of supernodes by padding the rows and columns that have only slightly different nonzero patterns, so that they can be merged into the same supernode. The supernodes in the factors are typically detected and recorded as they emerge during symbolic

factorization. In the remainder of this chapter, the term supernode refers to a supernode in the factors.

It can be seen from the algorithms in [Figs. 1](#) and [2](#) that there are two primary computations in a column-based factorization: the division step and the update step. A supernode-based sparse factorization too has the same two basic computation steps, except that these are now matrix operations on row/column blocks corresponding to the various supernodes.

Supernodes impart efficiency to numerical factorization and triangular solves because they permit floating point operations to be performed on dense submatrices instead of individual nonzeros, thus improving memory hierarchy utilization. Since rows and columns in supernodes share the nonzero structure, indirect addressing is minimized because the structure needs to be stored only once for these rows and columns. Supernodes help to increase the granularity of tasks, which is useful for improving computation to overhead ratio in a parallel implementation. The task graph model of sparse matrix factorization was introduced earlier with a task  $k$  defined as the factorization of row and column  $k$  of the matrix. With supernodes, a task can be defined as the factorization of all rows and columns associated with a supernode. Actual task graphs in practical implementations of parallel sparse solvers are almost always supernodal task graphs.

Note that some applications, such as power grid analysis, in which the basis of the linear system is not a finite-element or finite-difference discretization of a physical domain, can give rise to sparse matrices that incur very little fill-in during factorization. The factors of these matrices may have very small supernodes.

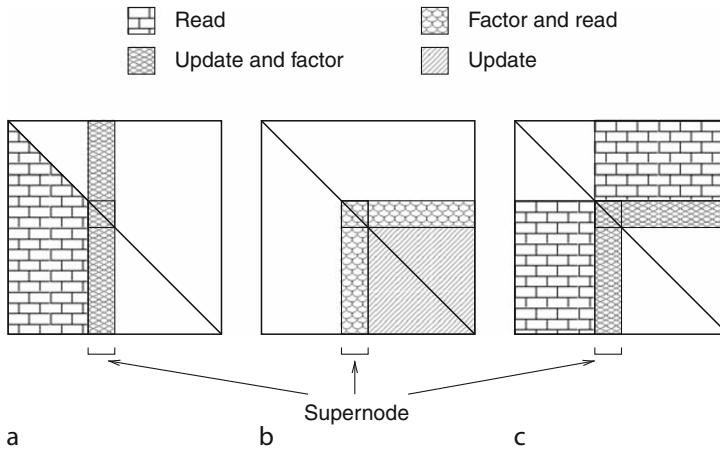
### An Effective Parallelization Strategy

The task graphs for sparse matrix factorization have some typical properties that make scheduling somewhat different from traditional DAG scheduling. Note that the task graphs corresponding to irreducible matrices have a distinct root; that is, one node that has no outgoing edges. This corresponds to the last (rightmost) supernode in the matrix. The number of member rows and columns in supernodes typically increases away from the leaves and toward the root of the task graph. The reason is that a supernode accumulates fill-in from all its predecessors in the task graph. As a result, the portions of the factors that correspond to task graph

nodes with a large number of predecessors tend to get denser. Due to their larger supernodes, the tasks that are relatively close to the root tend to have more work associated with them. On the other hand, the width of the task graph shrinks close to the root. In other words, a typical task graph for sparse matrix factorization tends to have a large number of small independent tasks closer to the leaves, but a small number of large tasks closer to the root. An ideal parallelization strategy that would match the characteristics of the problem is as follows. Starting out, the relatively plentiful independent tasks at or near the leaves would be scheduled to parallel threads or processes. As tasks complete, other tasks become available and would be scheduled similarly. This could continue until there are enough independent tasks to keep all the threads or processes busy. When the number of available parallel tasks becomes smaller than the number of available threads or processes, then the only way to keep the latter busy would be to utilize more than one of them per task. The number of threads or processes working on individual tasks would increase as the number of parallel tasks decreases. Eventually, all threads or processes would work on the root task. The computation corresponding to the root task is equivalent to factoring a dense matrix of the size of the root supernode.

### Sparse Factorization Formulations Based on Task Roles

So far in this entry, the tasks have been defined somewhat ambiguously. There are multiple ways of defining the tasks precisely, which can result in different parallel implementations of sparse matrix factorization. Clearly, a task is associated with a supernode and is responsible for computing that supernode of the factors; that is, performing the computation equivalent to the division steps in the algorithms in [Figs. 1](#) and [2](#). However, a task does not own all the data that is required to compute the final values of its supernode's rows and columns. The data for performing the update steps on a supernode may be contributed by many other supernodes. Based on the tasks' responsibilities, sparse LU factorization has traditionally been classified into three categories, namely, *left-looking*, *right-looking*, and *Crout*. These variations are illustrated in [Fig. 7](#). The traditional left-looking variant uses nonconforming supernodes made up of columns of both  $L$  and  $U$ , which are not very



**Sparse Direct Methods.** Fig. 7 The left-looking (a), right-looking (b), and Crout (c) variations of sparse LU factorization. Different patterns indicate the parts of the matrix that are read, updated, and factored by the task corresponding to a supernode. Blank portions of the matrix are not accessed by this task

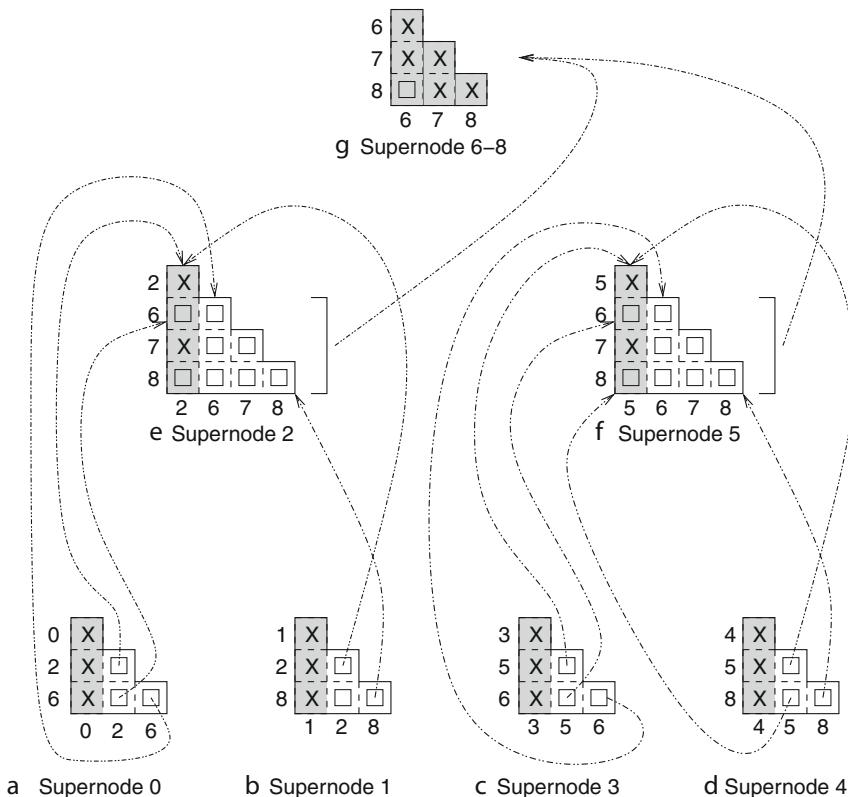
common in practice. In this variant, a task is responsible for gathering all the data required for its own columns from other tasks and for updating and factoring its columns. The left-looking formulation is rarely used in modern high-performance sparse direct solvers. In the right-looking variation of sparse LU, a task factors the supernode that it owns and performs all the updates that use the data from this supernode. In the Crout variation, a task is responsible for updating and factoring the supernode that it owns. Only the right-looking and Crout variants have symmetric counterparts.

A fourth variation, known as the *multifrontal method*, incorporates elements of both right-looking and Crout formulations. In the multifrontal method, the task that owns a supernode computes its own contribution to updating the remainder of the matrix (like the right-looking formulation), but does not actually apply the updates. Each task is responsible for collecting all relevant precomputed updates and applying them to its supernode (like the Crout formulation) before factoring the supernode. The supernode data and its update contribution in the multifrontal method is organized into small dense matrices called *frontal matrices*. Integer arrays maintain a mapping of the local contiguous indices of the frontal matrices to the global indices of the sparse factor matrices. Figure 8 illustrates the complete supernodal multifrontal Cholesky factorization of the symmetric matrix shown in Fig. 3a. Note that, since rows and columns with indices 6–8 form a supernode,

there would be only one task (Fig. 8g) corresponding to these in the supernodal task graph (elimination tree).

When a task is ready for execution, it first constructs its frontal matrix by accumulating contributions from the frontal matrices of its children and from the coefficient matrix. It then factors its supernode, which is the portion of the frontal matrix that is shaded in Fig. 8. After factorization, the unshaded portion (this submatrix of a frontal matrix is called the *update matrix*) is updated based on the update step of the algorithm in Fig. 2. The update matrix is then used by the parent task to construct its frontal matrix.

Note that Fig. 8 illustrates a symmetric multifrontal factorization; hence, the frontal and update matrices are triangular. For general LU decomposition, these matrices would be square or rectangular. In symmetric multifrontal factorization, a child's update matrix in the elimination tree contributes only to its parent's frontal matrix. The task graph for general matrices is usually not a tree, but a DAG, as shown in Fig. 4. Apart from the shape of the frontal and update matrices, unsymmetric pattern multifrontal method differs from its symmetric counterpart in two other ways. First, an update matrix can contribute to more than one frontal matrices. Secondly, the frontal matrices receiving data from an update matrix can belong to the contributing supernode's ancestors (not necessarily parents) in the task graph.



**Sparse Direct Methods.** Fig. 8 Frontal matrices and data movement among them in the supernodal multifrontal Cholesky factorization of the sparse matrix shown in Fig. 3a

The multifrontal method is often the formulation of choice for highly parallel implementations of sparse matrix factorization. This is because of its natural data locality (most of the work of the factorization is performed in the well-contained dense frontal matrices) and the ease of synchronization that it permits. In general, each supernode is updated by multiple other supernodes and it can potentially update many other supernodes during the course of factorization. If implemented naively, all these updates may require excessive locking and synchronization in a shared-memory environment or generate excessive message traffic in a distributed environment. In the multifrontal method, the updates are accumulated and channeled along the paths from the leaves of the task graph to its the root. This gives a manageable structure to the potentially haphazard interaction among the tasks.

Recall that the typical supernodal sparse factorization task graph is such that the size of tasks generally increases and the number of parallel tasks generally

diminishes on the way to the root from the leaves. The multifrontal method is well suited for both task parallelism (close to the leaves) and data parallelism (close to the root). Larger tasks working on large frontal matrices close to the root can readily employ multiple threads or processes to perform parallel dense matrix operations, which not only have well-understood data-parallel algorithms, but also a well-developed software base.

### Pivoting in Parallel Sparse $LDL^T$ and $LU$ Factorization

The discussion in this entry so far has focussed on the scenario in which the rows and columns of the matrix are permuted during the ordering phase and this permutation stays static during numerical factorization. While this assumption is valid for a large class of practical problems, there are applications that would generate matrices that could encounter a zero or a very small entry on the diagonal during the factorization process.

This will cause the division step of the LU decomposition algorithm to fail or to result in numerical instability. For nonsingular matrices, this problem can be solved by interchanging rows and columns of the matrix by a process known as *partial pivoting*. When a small or zero entry is encountered at  $A[i, i]$  before the division step, then row  $i$  is interchanged with another row  $j$  ( $i < j \leq n$ ) such that  $A[j, i]$  (which would occupy the location  $A[i, i]$  after the interchange) is sufficiently greater in magnitude compared to other entries  $A[k, i]$  ( $i < k \leq n, k \neq j$ ). Similarly, instead of row  $i$ , column  $i$  could be exchanged with a suitable column  $j$  ( $i < j \leq n$ ). In symmetric  $LDL^T$  factorization, both row and column  $i$  are interchanged simultaneously with a suitable row–column pair to maintain symmetry.

Until recently, it was believed that due to unpredictable changes in the structure of the factors due to partial pivoting, a priori ordering and symbolic factorization could not be performed, and these steps needed to be combined with numerical factorization. Keeping the analysis and numerical factorization steps separate has substantial performance and parallelization benefits, which would be lost if these steps are combined. Fortunately, modern parallel sparse solvers are able to perform partial pivoting and maintain numerical stability without mixing the analysis and numerical steps. The multifrontal method permits effective implementation of partial pivoting in parallel and keeps its effects as localized as possible.

Before computing a fill-reducing ordering, the rows or columns of the coefficient matrix are permuted such that the absolute value of the product of the magnitude of the diagonal entries is maximized. Special graph matching algorithms are used to compute this permutation. This step ensures that the diagonal entries of the matrix have relatively large magnitudes at the beginning of factorization. It has been observed that once the matrix has been permuted this way, in most cases, very few interchanges are required during the factorization process to keep it numerically stable. As a result, factorization can be performed using the static task graph and the static structures of the supernodes of  $L$  and  $U$  predicted by symbolic factorization. When an interchange is necessary, the resulting changes in the data structures are registered. Since such interchanges are rare, the resulting disruption and the overhead is usually well contained.

The first line of defense against numerical instability is to perform partial pivoting within a frontal matrix. Exchanging rows or columns within a supernode is local, and if all rows and columns of a supernode can be successfully factored by simply altering their order, then nothing outside the supernode is affected. Sometimes, a supernode cannot be factored completely by local interchanges. This can happen when all candidate rows or columns for interchange have indices greater than that of the last row–column pair of the supernode. In this case, a technique known as *delayed pivoting* is employed. The unfactored rows and columns are simply removed from the current supernode and passed onto the parent (or parents) in the task graph. Merged with the parent supernode, these rows and columns have additional candidate rows and columns available for interchange, which increases the chances of their successful factorization. The process of upward migration of unsuccessful pivots continues until they are resolved, which is guaranteed to happen at the root supernode for a nonsingular matrix.

In the multifrontal framework, delayed pivoting simply involves adding extra rows and columns to the frontal matrices of the parents of supernode with failed pivots. The process is straightforward for the supernodes whose tasks are mapped onto individual threads or processes. For the tasks that require data-parallel involvement of multiple threads or processes, the extra rows and columns can be partitioned using the same strategy that is used to partition the original frontal matrix.

## Parallel Solution of Triangular Systems

As mentioned earlier, solving the original system after factoring the coefficient matrix involves solving a sparse lower triangular and a sparse upper triangular system. The task graph constructed for factorization can be used for the triangular solves too. For matrices with an unsymmetric pattern, a subset of edges of the task DAG may be redundant in each of the solve phases, but these redundant edges can be easily marked during symbolic factorization. Just like factorization, the computation for the lower triangular solve phase starts at the leaves of the task graph and proceeds toward the root. In the upper triangular solve phase, computation starts at the root and fans out toward the leaves (in other

words, the direction of the edges in the task graph is effectively reversed).

## Related Entries

- [Dense Linear System Solvers](#)
- [Multifrontal Method](#)
- [Reordering](#)

## Bibliographic Notes and Further Reading

Books by George and Liu [6] and Duff et al. [5] are excellent sources for a background on sparse direct methods. A comprehensive survey by Demmel et al. [4] sums up the developments in parallel sparse direct solvers until the early 1990s. Some remarkable progress was made in the development of parallel algorithms and software for sparse direct methods during a decade starting in the early 1990s. Gupta et al. [9] developed the framework for highly scalable parallel formulations of symmetric sparse factorization based on the multifrontal method (see tutorial by Liu [12] for details), and recently demonstrated scalable performance of an industrial strength implementation of their algorithms on thousands of cores [10]. Demmel et al. [3] developed one of the first scalable algorithms and software for solving unsymmetric sparse systems without partial pivoting. Amestoy et al. [1, 2] developed parallel algorithms and software that incorporated partial pivoting for solving unsymmetric systems with (either natural or forced) symmetric pattern. Hadfield [11] and Gupta [7] laid the theoretical foundation for a general unsymmetric pattern parallel multifrontal algorithm with partial pivoting, with the latter following up with a practical implementation [8].

## Bibliography

1. Amestoy PR, Duff IS, Koster J, L'Excellent JY (2001) A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J Matrix Anal Appl* 23(1):15–41
2. Amestoy PR, Duff IS, L'Excellent JY (2000) Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput Methods Appl Mech Eng* 184:501–520
3. Demmel JW, Gilbert JR, Li XS (1999) An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM J Matrix Anal Appl* 20(4):915–952
4. Demmel JW, Heath MT, van der Vorst HA (1993) Parallel numerical linear algebra. *Acta Numerica* 2:111–197

5. Duff IS, Erisman AM, Reid JK (1990) Direct methods for sparse matrices. Oxford University Press, Oxford, UK
6. George A, Liu JW-H (1981) Computer solution of large sparse positive definite systems. Prentice-Hall, NJ
7. Gupta A (2002) Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices. *SIAM J Matrix Anal Appl* 24(2):529–552
8. Gupta A (2007) A shared- and distributed-memory parallel general sparse direct solver. *Appl Algebra Eng Commun Comput* 18(3):263–277
9. Gupta A, Karypis G, Kumar V (1997) Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Trans Parallel Distrib Syst* 8(5):502–520
10. Gupta A, Koric S, George T (2009) Sparse matrix factorization on massively parallel computers. In: SC09 Proceedings, ACM, Portland, OR, USA
11. Hadfield SM (1992) On the LU factorization of sequences of identically structured sparse matrices within a distributed memory environment. PhD thesis, University of Florida, Gainsville, FL
12. Liu JW-H (1992) The multifrontal method for sparse matrix solution: theory and practice. *SIAM Rev* 34(1):82–109

## Sparse Gaussian Elimination

- [Sparse Direct Methods](#)
- [SuperLU](#)

## Sparse Iterative Methods, Preconditioners for

- [Preconditioners for Sparse Iterative Methods](#)

## SPEC Benchmarks

MATTHIAS MÜLLER<sup>1</sup>, BRIAN WHITNEY<sup>2</sup>  
ROBERT HENSCHEL<sup>3</sup>, KALYAN KUMARAN<sup>4</sup>  
<sup>1</sup>Technische Universität Dresden, Dresden, Germany  
<sup>2</sup>Oracle Corporation, Hillsboro, OR, USA  
<sup>3</sup>Indiana University, Bloomington, IN, USA  
<sup>4</sup>Argonne National Laboratory, Argonne, IL, USA

## Synonyms

[SPEC HPC96](#); [SPEC HPC2002](#); [SPEC MPI2007](#); [SPEC OMP2001](#)

## Definition

Application-based Benchmarks measure the performance of computer systems by running a set of applications with a well defined configuration and workload.

## Discussion

### Introduction

The Standard Performance Evaluation Corporation (SPEC [Product and service names mentioned herein may be the trademarks of their respective owners]) is an organization for creating industry-standard benchmarks to measure various aspects of modern computer system performance. SPEC was founded in 1988. In January 1994, the High-Performance Group of the Standard Performance Evaluation Corporation (SPEC HPG) was founded with the mission to establish, maintain, and endorse a suite of benchmarks representative of real-world, high-performance computing applications. Several efforts joined forces to form SPEC HPG and to initiate a new benchmarking venture that is supported broadly. Founding partners included the member organizations of SPEC, former members of the Perfect Benchmark effort, and representatives of area-specific benchmarking activities. Other benchmarking organizations have joined the SPEC HPG committee since its formation.

SPEC HPG has developed various benchmark suites and its run rules over the last few years. The purpose of those benchmarks and their run rules is to further the cause of fair and objective benchmarking of high-performance computing systems. Results obtained with the benchmark suites are to be reviewed to see whether the individual run rules have been followed. Once they are accepted, the results are published on the SPEC web site (<http://www.spec.org>). All results, including a comprehensive description of the hardware they were produced on, are freely available. SPEC believes that the user community benefits from an objective series of tests which serve as a common reference.

The development of the benchmark suites includes obtaining candidate benchmark codes, putting these codes into the SPEC harness, testing and improving the codes' portability across as many operating systems, compilers, interconnects, runtime libraries as possible, and testing the codes for correctness and scalability.

The codes are put into the SPEC harness. The SPEC harness is a set of tools that allow users of the benchmark suite to easily run the suite, and obtain validated and publishable results. The users then only need to submit the results obtained to SPEC for review and publication on the SPEC web site.

The goals of the run rules of the benchmark suites are to ensure that published results are meaningful, comparable to other results, and reproducible. A result must contain enough information to allow another user to reproduce the result. The performance tuning methods employed when attaining a result should be more than just "prototype" or "experimental" or "research" methods; there must be a certain level of maturity and general applicability in the performance methods employed, e.g., the used compiler optimization techniques should be beneficial for other applications as well and the compiler should be generally available and supported.

Two set of metrics can be measured with the benchmark suite: "Peak" and "Base" metrics. "Peak" metrics may also be referred to as "aggressive compilation," e.g., they may be produced by building each benchmark in the suite with a set of optimizations individually selected for that benchmark, and running them with environment settings individually selected for that benchmark. The optimizations selected must adhere to the set of general benchmark optimization rules. Base optimizations must adhere to a stricter set of rules than the peak optimizations. For example, the "Base" metrics must be produced by building all the benchmarks in the suite with a common set of optimizations and running them with environment settings common to all the benchmarks in the suite.

### SPEC HPC96

The efforts of SPEC HPG began in 1994 when a group from industry and academia came together to try and provide a benchmark suite based upon the principles that had started with SPEC. Two of the more popular benchmarks suites at the time were the NAS Parallel Benchmarks and the PERFECT Club Benchmarks. The group built upon the direction these benchmarks provided to produce their first benchmark, SPEC HPC96.

The benchmark SPEC HPC96 came out originally with two components, SPECseis96 and SPECchem96, with a third component SPECclimate available later.

Each of these benchmarks provided a set of rules, code, and validation which allowed benchmarking across a wide variety of hardware, including parallel platforms.

SPECseis96 is a benchmark application that was originally developed at Atlantic Richfield Corporation (ARCO). This benchmark was designed to test computation that was of interest to the oil and gas industry, in particular, time and depth migrations which are used to locate gas and oil deposits.

SPECchem96 is a benchmark based upon the application GAMESS (General Atomic and Molecular Electronic Structure System). This computational chemistry code was used in the pharmaceutical and chemical industries for drug design and bonding analysis.

SPECclimate is a benchmark based upon the application MM5, the PSU/NCAR limited-area, hydrostatic or non-hydrostatic, sigma-coordinate model designed to simulate or predict mesoscale and regional-scale atmospheric circulation. MM5 was developed by the Pennsylvania State University (Penn State) and the University Corporation for Atmospheric Research (UCAR).

SPEC HPC96 was retired in February 2003, a little after the introduction of SPEC HPC2002 and SPEC OMP2001. The results remain accessible on the SPEC web site, for reference purposes.

## SPEC HPC2002

The benchmark SPEC HPC2002 was a follow-on to SPEC HPC96. The update involved using newer versions of some of the software, as well as additional parallelism models. The use of MM5 was replaced with the application WRF.

The benchmark was suitable for shared and distributed memory machines or clusters of shared memory nodes. SPEC HPC applications have been collected from among the largest, most realistic computational applications that are available for distribution by SPEC. In contrast to SPEC OMP, they were not restricted to any particular programming model or system architecture. Both shared-memory and message passing methods are supported. All codes of the current SPEC HPC2002 suite were available in an MPI and an OpenMP programming model and they included two data set sizes.

The benchmark consisted of three scientific applications:

*SPECenv* (WRF) is based on the WRF weather model, a state-of-the-art, non-hydrostatic mesoscale weather model, see <http://www.wrf-model.org>. The code consists of 25,000 lines of C and 145,000 lines of F90.

*SPECseis* was developed by ARCO beginning in 1995 to gain an accurate measure of performance of computing systems as it relates to the seismic processing industry for procurement of new computing resources. The code is written in F77 and C and has approximately 25,000 lines.

*SPECchem* used to simulate molecules ab initio, at the quantum level, and optimize atomic positions. It is a research interest under the name of GAMESS at the Gordon Research Group of Iowa State University and is of interest to the pharmaceutical industry. It consists of 120,000 lines of F77 and C.

The SPEC HPC2002 suite was retired in June 2007. The results remain accessible on the SPEC web site, for reference purposes.

## SPEC OMP2001

SPEC's benchmark suite that measures performance using applications based on the OpenMP standard for shared-memory parallel processing. Two levels of workload (OMPM2001 and OMPL2001) characterize the performance of medium and large sized systems. Benchmarks running under SPEC OMPM2001 use up to 1.6 GB of memory, whereas the applications of SPEC OMPL2001 require about 6.4 GB in a 16-thread run.

The SPEC OMPM2001 benchmark suite consists of 11 large application programs, which represent the type of software used in scientific technical computing. The applications include modeling and simulation programs from the fields of chemistry, mechanical engineering, climate modeling, and physics. Of the 11 application programs, 8 are written in Fortran and 3 (AMMP, ART, and EQUAKE) are written in C. The benchmarks require a virtual address space of about 1.5 GB in a 1-processor execution. The rationales for this size were to provide data sets fitting in a 32-bit address space.

SPEC OMPL2001 consists of 9 application programs, of which 7 are written in Fortran and 2 (ART and

*EQUAKE*) are written in C. The benchmarks require a virtual address space of about 6.4 GB in a 16-processor run. The rationale for this size were to provide data sets significantly larger than those of the SPEC OMPM benchmarks, with a requirement for a 64-bit address space.

The following is a short description of the application programs of OMP2001:

*APPLU* Solves five coupled non-linear PDEs on a 3-dimensional logically structured grid, using the Symmetric Successive Over-Relaxation implicit time-marching scheme.

*APSI* Lake environmental model, which predicts the concentration of pollutants. It solves the model for the mesoscale and synoptic variations of potential temperature, wind components, and for the mesoscale vertical velocity, pressure, and distribution of pollutants.

*MGRID* Simple multigrid solver, which computes a 3-dimensional potential field.

*SWIM* Weather prediction model, which solves the shallow water equations using a finite difference method.

*FMA3D* Crash simulation program. It simulates the inelastic, transient dynamic response of 3-dimensional solids and structures subjected to impulsively or suddenly applied loads. It uses an explicit finite element method.

*ART* (Adaptive Resonance Theory) neural network, which is used to recognize objects in a thermal image. The objects in the benchmark are a helicopter and an airplane.

*GAFORT* Computes the global maximum fitness using a genetic algorithm. It starts with an initial population and then generates children who go through crossover, jump mutation, and creep mutation with certain probabilities.

*EQUAKE* Is an earthquake-modeling program. It simulates the propagation of elastic seismic waves in large, heterogeneous valleys in order to recover the time history of the ground motion everywhere in the valley due to a specific seismic event. It uses a finite element method on an unstructured mesh.

*WUPWISE* (Wuppertal Wilson Fermion Solver) is a program in the field of lattice gauge theory. Lattice

gauge theory is a discretization of quantum chromodynamics. Quark propagators are computed within a chromodynamic background field. The inhomogeneous lattice-Dirac equation is solved.

*GALGEL* This problem is a particular case of the GAMM (Gesellschaft fuer Angewandte Mathematik und Mechanik) benchmark devoted to numerical analysis of oscillatory instability of convection in low-Prandtl-number fluids. This program is only part of OMPM2001.

*AMMP* (Another Molecular Modeling Program) is a molecular mechanics, dynamics, and modeling program. The benchmark performs a molecular dynamics simulation of a protein-inhibitor complex, which is embedded in water. This program is only part of OMPM2001.

## SPEC MPI2007

SPEC MPI2007 is SPEC's benchmark suite for evaluating MPI-parallel, floating point, compute-intensive performance across a wide range of cluster and SMP hardware. MPI2007 continues the SPEC tradition of giving users the most objective and representative benchmark suite for measuring and comparing high-performance computer systems.

SPEC MPI2007 focuses on performance of compute intensive applications using the Message-Passing Interface (MPI), which means these benchmarks emphasize the performance of the type of computer processor (CPU), the number of computer processors, the MPI Library, the communication interconnect, the memory architecture, the compilers, and the shared file system.

It is important to remember the contribution of all these components. SPEC MPI performance intentionally depends on more than just the processor. MPI2007 is not intended to stress other computer components such as the operating system, graphics, or the I/O system. Table 1 contains the list of codes, together with information on the benchmark set size, programming language, and application area.

*104.milc,142.dmilc* stands for MIMD Lattice Computation and is a quantum chromodynamics (QCD) code for lattice gauge theory with dynamical quarks. Lattice gauge theory involves the study of some of the fundamental constituents of matter, namely

**SPEC Benchmarks. Table 1** List of applications in SPEC MPI2007

| Benchmark     | Suite         | Language  | Application domain                                       |
|---------------|---------------|-----------|----------------------------------------------------------|
| 104.milc      | medium        | C         | Physics: Quantum Chromodynamics (QCD)                    |
| 107.leslie3d  | medium        | Fortran   | Computational Fluid Dynamics (CFD)                       |
| 113.GemsFDTD  | medium        | Fortran   | Computational Electromagnetics (CEM)                     |
| 115.fds4      | medium        | C/Fortran | Computational Fluid Dynamics (CFD)                       |
| 121.pop2      | medium, large | C/Fortran | Ocean Modeling                                           |
| 122.tachyon   | medium, large | C         | Graphics: Parallel Ray Tracing                           |
| 125.RAxML     | large         | C         | DNA Matching                                             |
| 126.lammps    | medium, large | C++       | Molecular Dynamics Simulation                            |
| 127.wrf2      | medium        | C/Fortran | Weather Prediction                                       |
| 128.GAPgeofem | medium, large | C/Fortran | Heat Transfer using Finite Element Methods (FEM)         |
| 129.tera_tf   | medium, large | Fortran   | 3D Eulerian Hydrodynamics                                |
| 130.socorro   | medium        | C/Fortran | Molecular Dynamics using Density-Functional Theory (DFT) |
| 132.zeusmp2   | medium, large | C/Fortran | Physics: Computational Fluid Dynamics (CFD)              |
| 137.lu        | medium, large | Fortran   | Computational Fluid Dynamics (CFD)                       |
| 142.dmilc     | large         | C         | Physics: Quantum Chromodynamics (QCD)                    |
| 143.dleslie   | large         | Fortran   | Computational Fluid Dynamics (CFD)                       |
| 145.lGemsFDTD | large         | Fortran   | Computational Electromagnetics (CEM)                     |
| 147.lwrf2     | large         | C/Fortran | Weather Prediction                                       |

quarks and gluons. In this area of quantum field theory traditional perturbative expansions are not useful and introducing a discrete lattice of space-time points is the method of choice.

*107.leslie3d,143.dleslie* The main purpose of this code is to model chemically reacting (i.e., burning) turbulent flows. Various different physical models are available in this algorithm. For MPI2007, the program has been set up a to solve a test problem which represents a subset of such flows, namely the temporal mixing layer. This type of flow occurs in the mixing regions of all combustors that employ fuel injection (which is nearly all combustors). Also, this sort of mixing layer is a benchmark problem used to understand physics of turbulent mixing.

LESlie3d uses a strongly conservative, finite-volume algorithm with the MacCormack Predictor-Corrector time integration scheme. The accuracy is fourth-order spatially and second-order temporally.

*113.GemsFDTD,145.lGemsFDTD* GemsFDTD solves the Maxwell equations in 3D in the time domain using the finite-difference time-domain (FDTD)

method. The radar cross section (RCS) of a perfectly conducting (PEC) object is computed. GemsFDTD is a subset of the code GemsTD developed in the General ElectroMagnetic Solvers (GEMS) project. The core of the FDTD method are second-order accurate central-difference approximations of the Faraday's and Ampere's laws. These central-differences are employed on a staggered Cartesian grid resulting in an explicit finite-difference method. The FDTD method is also referred to as the Yee scheme. It is the standard time-domain method within computational electrodynamics (CEM).

An incident plane wave is generated using so-called Huygens' surfaces. This means that the computational domain is split into a total field part and a scattered field part, where the scattered field part surrounds the total field part. A time-domain near-to-far-field transformation computes the RCS according to Martin and Pettersson. Fast Fourier transforms (FFT) are employed in the post-processing.

145.lGemsFDTD contains extensive performance improvements as compared with 113.GemsFDTD.

*115.fds4* is a computational fluid dynamics (CFD) model of fire-driven fluid flow. The software solves numerically a form of the Navier-Stokes equations appropriate for low-speed, thermally driven flow with an emphasis on smoke and heat transport from fires. It uses the block64 test case as dataset. This dataset is similar to ones used to simulate fires in the World Trade Center. The author's agency did the investigation of the collapse.

*121.pop2* The Parallel Ocean Program (POP) is a descendant of the Bryan-Cox-Semtner class of ocean models first developed by Kirk Bryan and Michael Cox at the NOAA Geophysical Fluid Dynamics Laboratory in Princeton, NJ, in the late 1960s. POP had its origins in a version of the model developed by Semtner and Chervin. POP is the ocean component of the Community Climate System Model. Time integration of the model is split into two parts. The three-dimensional vertically varying (baroclinic) tendencies are integrated explicitly using a leapfrog scheme. The very fast vertically uniform (barotropic) modes are integrated using an implicit free surface formulation in which a preconditioned conjugate gradient solver is used to solve for the two-dimensional surface pressure.

*122.tachyon* is a ray tracing program. It implements all of the basic geometric primitives such as triangles, planes, spheres, cylinders, etc. Tachyon is nearly embarrassingly parallel. As a result, MPI usage tends to be much lower as compared to other types of MPI applications. The scene to be rendered is partitioned into a fixed number of pieces, which are distributed out by the master process to each processor participating in the computation. Each processor then renders its piece of the scene in parallel, independent of the other processors. Once a processor completes the rendering of its particular piece of the scene, it waits until the other processors have rendered their pieces of the scene, and then transmits its piece back to the master process. The process is repeated until all pieces of the scene have been rendered.

*126.lammps* is a classical molecular dynamics simulation code designed to run efficiently on parallel computers. It was developed at Sandia National Laboratories, a US Department of Energy facility, with

funding from the DOE. LAMMPS divides 3D space into 3D sub-volumes, e.g., a  $P = A \times B \times C$  grid of processors, where P is the total number of processors. It tries to make the sub-volumes as cubic as possible, since the volume of data exchanged is proportional to the surface of the sub-volume.

*127.wrf2,147.l2wrf* is a weather forecasting code based on the Weather Research and Forecasting (WRF) Model, which is a next-generation mesoscale numerical weather prediction system designed to serve both operational forecasting and atmospheric research needs. The code is written in Fortran 90. WRF features multiple dynamical cores, a 3-dimensional variational (3DVAR) data assimilation system, and a software architecture allowing for computational parallelism and system extensibility. Multi-level parallelism support includes distributed memory (MPI), shared memory (OpenMP), and hybrid shared/distributed modes of execution. In the SPEC MPI2007 version of WRF, all OpenMP directives have been switched off. WRF version 2.0.2 is used in the 127.wrf2 benchmark version. WRF version 2.1.2 is used in the benchmark version, 147.l2wrf2.

*128.GAPgeofem* is a GeoFEM-based parallel finite-element code for transient thermal conduction with gap radiation and very heterogeneous material property. GeoFEM is an acronym for Geophysical Finite Element Methods, and it is a software used on the Japanese Earth Simulator system for the modeling of solid earth phenomena such as mantle-core convection, plate tectonics, and seismic wave propagation and their coupled phenomena. A backward Euler implicit time-marching scheme has been adopted. Linear equations are solved by parallel CG (conjugate gradient) iterative solvers with point Jacobi preconditioning.

*129.tera\_tf* is a three dimensional Eulerian hydrodynamics application using a 2nd order Godunov-type scheme and a 3rd order remapping. It uses mostly point-to-point messages, and some reductions use non-blocking messages. The global domain is a cube, with  $N$  cells in each direction, which amounts to a total number of  $N^3$  cells. To set up the problem, one needs to define the number of cells in each direction, and the number of blocks in each direction. Each block corresponds to an MPI task.

*130.socorro* is a modular, object oriented code for performing self-consistent electronic-structure calculations utilizing the Kohn-Sham formulation of density-functional theory. Calculations are performed using a plane wave basis and either norm-conserving pseudopotentials or projector augmented wave functions. Several exchange-correlation functionals are available for use including the local-density approximation (Perdew-Zunger or Perdew-Wang parameterizations of the Ceperley-Alder QMC correlation results) and the generalized-gradient approximation (PW91, PBE, and BLYP). Both Fourier-space and real-space projectors have been implemented, and a variety of methods are available for relaxing atom positions, optimizing cell parameters, and performing molecular dynamics calculations.

*132.zeusmp2* is a computational fluid dynamics code developed at the Laboratory for Computational Astrophysics (NCSA, SDSC, University of Illinois at Urbana-Champaign, UC San Diego) for the simulation of astrophysical phenomena. The program solves the equations of ideal (non-resistive), non-relativistic, hydrodynamics and magnetohydrodynamics, including externally applied gravitational fields and self-gravity. The gas can be adiabatic or isothermal, and the thermal pressure is isotropic. Boundary conditions may be specified as reflecting, periodic, inflow, or outflow.

*132.zeusmp2* is based on ZEUS-MP Version 2, which is a Fortran 90 rewrite of ZEUS-MP under development at UCSD (by John Hayes). It includes new physics models such as flux-limited radiation diffusion (FLD), and multispecies fluid advection is added to the physics set. ZEUS-MP divides the computational space into 3D tiles that are distributed across processors. The physical problem solved in SPEC MPI2007 is a 3D blastwave simulated with the presence of a uniform magnetic field along the x-direction. A Cartesian grid is used and the boundaries are “outflow.”

*137.lu* has a rich ancestry in benchmarking. Its immediate predecessor is the LU benchmark in NPB3.2-MPI, part of the NAS Parallel Benchmark suite. It is sometimes referred to as APPLU (a version of which was 173.applu in CPU2000) or NAS-LU. The NAS-LU code is a simplified compressible

Navier-Stokes equation solver. It does not perform an LU factorization, but instead implements a symmetric successive over-relaxation (SSOR) numerical scheme to solve a regular-sparse, block lower and upper triangular system.

The code computes the solution of five coupled nonlinear PDE's, on a 3-dimensional logically structured grid, using an implicit pseudo-time marching scheme, based on two-factor approximate factorization of the sparse Jacobian matrix. This scheme is functionally equivalent to a nonlinear block SSOR iterative scheme with lexicographic ordering. Spatial discretization of the differential operators are based on a second-order accurate finite volume scheme. It insists on the strict lexicographic ordering during the solution of the regular sparse lower and upper triangular matrices. As a result, the degree of exploitable parallelism during this phase is limited to  $O(N^2)$  as opposed to  $O(N^3)$  in other phases and its spatial distribution is non-homogenous. This fact also creates challenges during the loop re-ordering to enhance the cache locality.

## Related Entries

- ▶ [Benchmarks](#)
- ▶ [HPC Challenge Benchmark](#)
- ▶ [NAS Parallel Benchmarks](#)
- ▶ [Perfect Benchmarks](#)

## Bibliographic Notes and Further Reading

There are numerous efforts to create benchmarks for different purposes. One of the early application benchmarks are the so-called “Perfect Club Benchmarks”<sup>[6]</sup>, an effort that among others initiated the SPEC High Performance Group (HPG) activities<sup>[9]</sup>. The goal of SPEC HPG is to create application benchmarks to measure the performance of High Performance Computers. There are also other efforts that share this goal. Often such a collection is assembled during a procurement process. However, it normally is not used outside this specific process, nor does such a collection claim to be representative for a wider community. Two of the collections that are in wider use are the NAS Parallel Benchmarks<sup>[5]</sup> and the HPC Challenge benchmark<sup>[8]</sup>. The NAS Parallel Benchmarks (NPB) are a small set of programs designed to help evaluate the performance of parallel

supercomputers. The benchmarks, which are derived from computational fluid dynamics (CFD) applications, consist of five kernels and three pseudo-applications. HPCC consists of synthetic kernels measuring different aspects of a parallel system, like CPU, memory subsystem, and interconnect.

Two other benchmarks that consist of applications are the ASCI-benchmarks used in the procurement process of various ASCI-machines. ASCI-Purple [1] consists of nine benchmarks and three stress tests. Its last update was in 2003. The ASCI-Sequoia benchmark [2] is the successor, it consists of 17 applications and synthetic benchmarks, but only seven codes are MPI parallel. A similar collection is the DEISA benchmark suite [7]. It contains 14 codes, but for licensing reasons, three of the benchmarks must be obtained directly from the code authors and placed in the appropriate location within the benchmarking framework.

There are also a number of publications with more details about the SPEC HPG benchmarks. Details of SPEC HPC2002 are available in [10]. Characteristics of the SPEC benchmark suite OMP2001 are described by Saito et al. [14] and Müller et. al [12]. Aslot et al. [3] have presented the benchmark suite. Aslot et al. [4] and Iwashita et al. [11] have described performance characteristics of the benchmark suite. The SPEC High Performance Group published a more detailed description of MPI2007 in [13]. Some performance characteristics are depicted in [15].

## Bibliography

1. ASCI-Purple home page (2003) [https://asc.llnl.gov/computing\\_resources/purple/archive/benchmarks](https://asc.llnl.gov/computing_resources/purple/archive/benchmarks)
2. ASCI-Sequoia home page. <https://asc.llnl.gov/sequoia/benchmarks>.
3. Aslot V, Domeika M, Eigenmann R, Gaertner G, Jones WB, Parady B (2001) SPEComp: a new benchmark suite for measuring parallel computer performance. In: Eigenmann R, Voss MJ (eds) WOMPAT'01: workshop on openmp applications and tools. LNCS, vol 2104. Springer, Heidelberg, pp 1–10
4. Aslot V, Eigenmann R (2001) Performance characteristics of the SPEC OMP2001 benchmarks. In: 3rd European workshop on OpenMP, EWOMP'01, Barcelona, Spain, September 2001
5. Bailey D, Harris T, Saphir W, van der Wijngaart R, Woo A, Yarrow M (1995) The NAS parallel benchmarks 2.0. Technical report NAS-95-020, NASA Ames Research Center, Moffett Field, CA. <http://www.nas.nasa.gov/Software/NPB>
6. Berry M, Chen D, Koss P, Kuck D, Lo S, Pang Y, Pointer, Rolo R, Sameh A, Clementi E, Chin S, Schneider D, Fox G, Messina P, Walker D, Hsiung C, Schwarzmeier J, Lue K, Orszag S, Seidl F, Johnson O, Goodrum R, Martin J (1989) The perfect club benchmarks: effective performance evaluation of supercomputers. *Int J High Perform Comp Appl* 3(3):5–40
7. DEISA benchmark suite home page. <http://www.deisa.eu/science/benchmarking>
8. Dongarra J, Luszczek P (2005) Introduction to the hpcchallenge benchmark suite. ICL technical report ICL-UT-05-01, ICL 2005
9. Eigenmann R, Hassanzadeh S (1996) Benchmarking with real industrial applications: the SPEC high-performance group. *IEEE Comp Sci & Eng* 3(1):18–23
10. Eigenmann R, Gaertner G, Jones W (2002) SPEC HPC2002: the next high-performance computer benchmark. In: Lecture notes in computer science, vol 2327. Springer, Heidelberg, pp 7–10
11. Iwashita H, Yamanaka E, Sueyasu N, van Waveren M, Miura K (2001) The SPEC OMP 2001 benchmark on the Fujitsu PRIMEPOWER system. In: 3rd European workshop on OpenMP, EWOMP'01, Barcelona, Spain, September 2001
12. Müller MS, Kalyanasundaram K, Gaertner G, Jones W, Eigenmann R, Lieberman R, van Waveren M, Whitney B (2004) SPEC HPG benchmarks for high performance systems. *Int J High Perform Comp Netw* 1(4):162–170
13. Müller MS, van Waveren M, Lieberman R, Whitney B, Saito H, Kumaran K, Baron J, Brantley WC, Parrott C, Elken T, Feng H, Ponder C (2010) SPEC MPI2007 – an application benchmark suite for parallel systems using MPI. *Concurrency Computat: Pract Exper* 22(2):191–205
14. Saito H, Gaertner G, Jones W, Eigenmann R, Iwashita H, Lieberman R, van Waveren M, Whitney B (2002) Large system performance of SPEC OMP2001 benchmarks. In: Zima HP, Joe K, Sata M, Seo Y, Shimasaki M (eds) High performance computing, 4th international symposium, ISHPC 2002. Lecture notes in computer science, vol 2327. Springer, Heidelberg, pp 370–379
15. Szebenyi Z, Wylie BJN, Wolf F (2008) SCALASCA parallel performance analyses of SPEC MPI2007 applications. In: Proceedings of the 1st SPEC international performance evaluation workshop (SIPEW). LNCS, vol 5119. Springer, Heidelberg, pp 99–123

## SPEC HPC2002

### ► SPEC Benchmarks

## SPEC HPC96

### ► SPEC Benchmarks

## SPEC MPI2007

### ► SPEC Benchmarks

## SPEC OMP2001

- [SPEC Benchmarks](#)

## Special-Purpose Machines

- [Anton, a Special-Purpose Molecular Simulation Machine](#)
- [GRAPE](#)
- [JANUS FPGA-Based Machine](#)
- [QCD apeNEXT Machines](#)
- [QCDSF and QCDOC Computers](#)

## Speculation

- [Speculative Parallelization of Loops](#)
- [Speculation, Thread-Level](#)
- [Transactional Memory](#)

## Speculation, Thread-Level

JOSEP TORRELLAS  
University of Illinois at Urbana-Champaign, Urbana,  
IL, USA

### Synonyms

[Speculative multithreading \(SM\)](#); [Speculative parallelization](#); [Speculative run-time parallelization](#); [Speculative threading](#); [Speculative thread-level parallelization](#); [Thread-level data speculation \(TLDS\)](#); [Thread level speculation \(TLS\) parallelization](#); [TLS](#)

### Definition

Thread-Level Speculation (TLS) refers to an environment where execution threads operate speculatively, performing potentially unsafe operations, and temporarily buffering the state they generate in a buffer or cache. At a certain point, the operations of a thread are declared to be correct or incorrect. If they are correct, the thread commits, merging the state it generated with the correct state of the program; if they are incorrect,

the thread is squashed and typically restarted from its beginning. The term TLS is most often associated to a scenario where the purpose is to execute a sequential application in parallel. In this case, the compiler or the hardware breaks down the application into speculative threads that execute in parallel. However, strictly speaking, TLS can be applied to any environment where threads are executed speculatively and can be squashed and restarted.

## Discussion

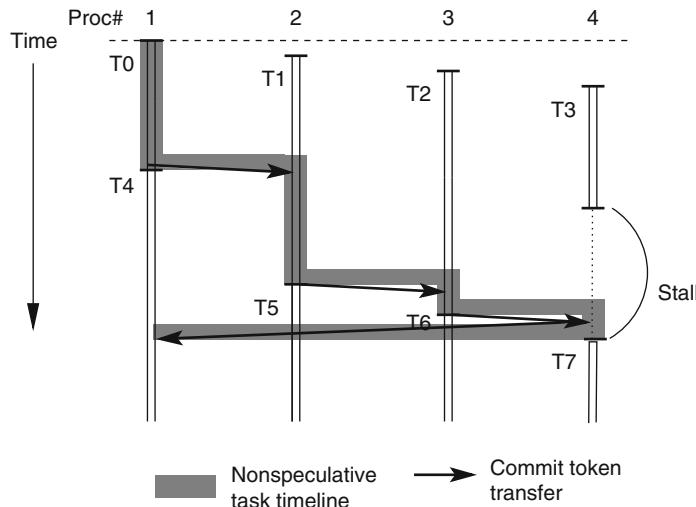
### Basic Concepts in Thread-Level Speculation

In its most common use, Thread-Level Speculation (TLS) consists of extracting units of work (i.e., tasks) from a sequential application and executing them on different threads in parallel, hoping not to violate sequential semantics. The control flow in the sequential code imposes a relative ordering between the tasks, which is expressed in terms of predecessor and successor tasks. The sequential code also induces a data dependence relation on the memory accesses issued by the different tasks that parallel execution cannot violate.

A task is *Speculative* when it may perform or may have performed operations that violate data or control dependences with its predecessor tasks. Otherwise, the task is nonspeculative. The memory accesses issued by speculative tasks are called speculative memory accesses.

When a nonspeculative task finishes execution, it is ready to *Commit*. The role of commit is to inform the rest of the system that the data generated by the task is now part of the safe, nonspeculative program state. Among other operations, committing always involves passing the *Commit Token* to the immediate successor task. This is because maintaining correct sequential semantics in the parallel execution requires that tasks commit in order from predecessor to successor. If a task reaches its end and is still speculative, it cannot commit until it acquires nonspeculative status and all its predecessors have committed.

[Figure 1](#) shows an example of several tasks running on four processors. In this example, when task T3 executing on processor 4 finishes the execution, it cannot commit until its predecessor tasks T0, T1, and T2 also finish and commit. In the meantime, depending on



**Speculation, Thread-Level. Fig. 1** A set of tasks executing on four processors. The figure shows the nonspeculative task timeline and the transfer of the commit token

the hardware support, processor 4 may have to stall or may be able to start executing speculative task T7. The example also shows how the nonspeculative task status changes as tasks finish and commit, and the passing of the commit token.

Memory accesses issued by a speculative task must be handled carefully. Stores generate *Speculative Versions* of data that cannot simply be merged with the nonspeculative state of the program. The reason is that they may be incorrect. Consequently, these versions are stored in a *Speculative Buffer* local to the processor running the task – e.g., the first-level cache. Only when the task becomes nonspeculative are its versions safe.

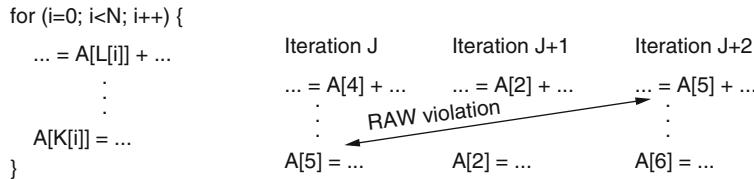
Loads issued by a speculative task try to find the requested datum in the local speculative buffer. If they miss, they fetch the correct version from the memory subsystem, i.e., the closest predecessor version from the speculative buffers of other tasks. If no such version exists, they fetch the datum from memory.

As tasks execute in parallel, the system must identify any violations of cross-task data dependences. Typically, this is done with special hardware or software support that tracks, for each individual task, the data that the task wrote and the data that the task read without first writing it. A data-dependence violation is flagged when a task modifies a datum that has been read earlier by a successor task. At this point, the consumer

task is *squashed* and all the data versions that it has produced are discarded. Then, the task is re-executed.

Figure 2 shows an example of a data-dependence violation. In the example, each iteration of a loop is a task. Each iteration issues two accesses to an array, through an un-analyzable subscripted subscript. At run-time, iteration J writes A[5] after its successor iteration J+2 reads A[5]. This is a Read After Write (RAW) dependence that gets violated due to the parallel execution. Consequently, iteration J+2 is squashed and restarted. Ordinarily, all the successor tasks of iteration J+2 are also squashed at this time because they may have consumed versions generated by the squashed task. While it is possible to selectively squash only tasks that used incorrect data, it would involve extra complexity. Finally, as iteration J+2 re-executes, it will re-read A[5]. However, at this time, the value read will be the version generated by iteration J.

Note that WAR and WAW dependence violations do not need to induce task squashes. The successor task has prematurely written the datum, but the datum remains buffered in its speculative buffer. A subsequent read from a predecessor task (in a WAR violation) will get a correct version, while a subsequent write from a predecessor task (in a WAW violation) will generate a version that will be merged with main memory before the one from the successor task.



**Speculation, Thread-Level. Fig. 2** Example of a data-dependence violation

However, many proposed TLS schemes, to reduce hardware complexity, induce squashes in a variety of situations. For instance, if the system has no support to keep different versions of the same datum in different speculative buffers in the machine, cross-task WAR and WAW dependence violations induce squashes. Moreover, if the system only tracks accesses on a per-line basis, it cannot disambiguate accesses to different words in the same memory line. In this case, false sharing of a cache line by two different processors can appear as a data-dependence violation and also trigger a squash.

Finally, while TLS can be applied to various code structures, it is most often applied to loops. In this case, tasks are typically formed by a set of consecutive iterations.

The rest of this article is organized as follows: First, the article briefly classifies TLS schemes. Then, it describes the two major problems that any TLS scheme has to solve, namely, buffering and managing speculative state, and detecting and handling dependence violations. Next, it describes the initial efforts in TLS, other uses of TLS, and machines that use TLS.

## Classification of Thread-Level Speculation Schemes

There have been many proposals of TLS schemes. They can be broadly classified depending on the emphasis on hardware versus software, and the type of target machine.

The majority of the proposed schemes use hardware support to detect cross-task dependence violations that result in task squashes (e.g., [1, 4, 6, 8, 11, 12, 14, 16, 18, 20, 23, 27, 28, 31, 32, 36]). Typically, this is attained by using the hardware cache coherence protocol, which sends coherence messages between the caches when multiple processors access the same memory line. Among all these hardware-based schemes, the majority rely on a compiler or a software layer to identify and prepare the tasks that should be executed in parallel. Consequently,

there have been several proposals for TLS compilers (e.g., [9, 19, 33, 34]). Very few schemes rely on the hardware to identify the tasks (e.g., [1]).

Several schemes, especially in the early stages of TLS research, proposed software-only approaches to TLS (e.g., [7, 13, 25, 26]). In this case, the compiler typically generates code that causes each task to keep shadow locations and, after the parallel execution, checks if multiple tasks have updated a common location. If they have, the original state is restored.

Most proposed TLS schemes target small shared-memory machines of about two to eight processors (e.g., [14, 18, 27, 29]). It is in this range of parallelism that TLS is most cost effective. Some TLS proposals have focused on smaller machines and have extended a superscalar core with some hardware units that execute threads speculatively [1, 20]. Finally, some TLS proposals have targeted scalable multiprocessors [4, 23, 28]. This is a more challenging environment, given the longer communication latencies involved. It requires applications that have significant parallelism that cannot be analyzed statically by the compiler.

## Buffering and Managing Speculative State

The state produced by speculative tasks is unsafe, since such tasks may be squashed. Therefore, any TLS scheme must be able to identify such state and, when necessary, separate it from the rest of the memory state. For this, TLS systems use structures, such as caches [4, 6, 12, 18, 28], and special buffers [8, 14, 23, 32], or undo logs [7, 11, 36]. This section outlines the challenges in buffering and managing speculative state. A more detailed analysis and a taxonomy is presented by Garzaran et al. [10].

## Multiple Versions of the Same Variable in the System

Every time that a task writes for the first time to a variable, a new version of the variable appears in the

system. Thus, two speculative tasks running on different processors may create two different versions of the same variable [4, 12]. These versions need to be buffered separately, and special actions may need to be taken so that a reader task can find the correct version out of the several coexisting in the system. Such a version will be the version created by the producer task that is the closest predecessor of the reader task.

A task has at most a single version of any given variable, even if it writes to the variable multiple times. The reason is that, on a dependence violation, the whole task is undone. Therefore, there is no need to keep intermediate values of the variable.

### Multiple Speculative Tasks per Processor

When a processor finishes executing a task, the task may still be speculative. If the TLS buffering support is such that the processor can only hold state from a single speculative task, the processor stalls until the task commits. However, to better tolerate task load imbalance, the local buffer may have been designed to buffer state from several speculative tasks, enabling the processor to execute another speculative task. In this case, the state of each task must be tagged with the ID of the task.

### Multiple Versions of the Same Variable in a Single Processor

When a processor buffers state from multiple speculative tasks, it is possible that two such tasks create two versions of the same variable. This occurs in load-imbalanced applications that exhibit private data patterns (i.e., WAW dependences between tasks). In this case, the buffer will have to hold multiple versions of the same variable. Each version will be tagged with a different task ID. This support introduces complication to the buffer or cache. Indeed, on an external request, extra comparisons will need to be done if the cache has two versions of the same variable.

### Merging of Task State

The state produced by speculative tasks is typically merged with main memory at task commit time; however, it can instead be merged as it is being generated. The first approach is called *Architectural Main Memory (AMM)* or *Lazy Version Management*; the second one is called *Future Main Memory (FMM)* or *Eager Version Management*. These schemes differ on whether the main

memory contains only safe data (AMM) or it can also contain speculative data (FMM).

In AMM systems, all speculative versions remain in caches or buffers that are kept separate from the coherent memory state. Only when a task becomes non-speculative can its buffered state be merged with main memory. In a straightforward implementation, when a task commits, all the buffered dirty cache lines are merged with main memory, either by writing back the lines to memory [4] or by requesting ownership for them to obtain coherence with main memory [28].

In FMM systems, versions from speculative tasks are merged with the coherent memory when they are generated. However, to enable recovery from task squashes, when a task generates a speculative version of a variable, the previous version of the variable is saved in a log. Note that, in both approaches, the coherent memory state can temporarily reside in caches, which function in their traditional role of extensions of main memory.

## Detecting and Handling Dependence Violations

### Basic Concepts

The second aspect of TLS involves detecting and handling dependence violations. Most TLS proposals focus on data dependences, rather than control dependences. To detect (cross-task) data-dependence violations, most TLS schemes use the same approach. Specifically, when a speculative task writes a datum, the hardware sets a Speculative Write bit associated with the datum in the cache; when a speculative task reads a datum before it writes to it (an event called *Exposed Read*), the hardware sets an Exposed Read bit. Depending on the TLS scheme supported, these accesses also cause a tag associated with the datum to be set to the ID of the task.

In addition, when a task writes a datum, the cache coherence protocol transaction that sends invalidations to other caches checks these bits. If a successor task has its Exposed Read bit set for the datum, the successor task has prematurely read the datum (i.e., this is a RAW dependence violation), and is squashed [18].

If the Speculative Write and Exposed Read bits are kept on a per-word basis, only dependences on the same word can cause squashes. However, keeping and maintaining such bits on a per-word basis in caches, network

messages, and perhaps directory modules is costly in hardware. Moreover, it does not come naturally to the coherence protocol of multiprocessors, which operate at the granularity of memory lines.

Keeping these bits on a per-line basis is cheaper and compatible with mainstream cache coherence protocols. However, the hardware cannot then disambiguate accesses at word level. Furthermore, it cannot combine different versions of a line that have been updated in different words. Consequently, cross-task RAW and WAW violations, on both the same word and different words of a line (i.e., false sharing), cause squashes.

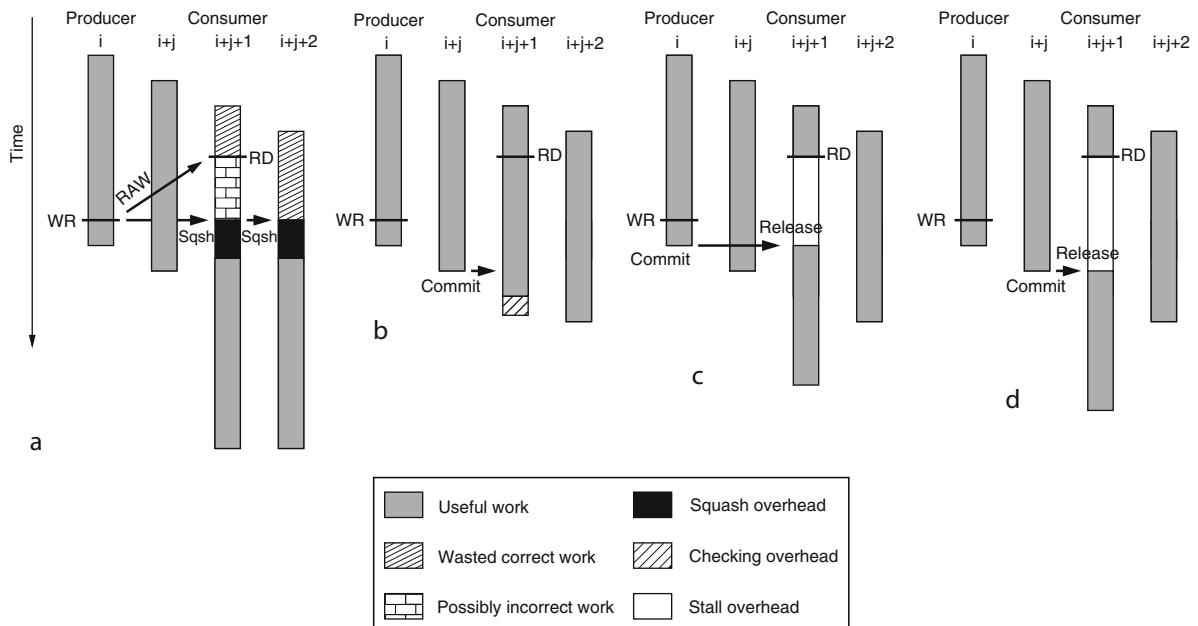
Task squash is a very costly operation. The cost is threefold: overhead of the squash operation itself, loss of whatever correct work has already been performed by the offending task and its successors, and cache misses in the offending task and its successors needed to reload state when restarting. The latter overhead appears because, as part of the squash operation, the speculative state in the cache is invalidated. Figure 3a shows an example of a RAW violation across tasks  $i$  and  $i+j+1$ . The consumer task and its successors are squashed.

### Techniques to Avoid Squashes

Since squashes are so expensive, there are techniques to avoid them. If the compiler can conclude that a certain pair of accesses will frequently cause a data-dependence violation, it can statically insert a synchronization operation that forces the correct task ordering at runtime.

Alternatively, the machine can have hardware support that records, at runtime, where dependence violations occur. Such hardware may record the program counter of the read or writes involved, or the address of the memory location being accessed. Based on this information, when these program counters are reached or the memory location is accessed, the hardware can try one of several techniques to avoid the violation. This section outlines some of the techniques that can be used. A more complete description of the choices is presented by Cintra and Torrellas [5]. Without loss of generality, a RAW violation is assumed.

Based on past history, the predictor may predict that the pair of conflicting accesses are engaged in false sharing. In this case, it can simply allow the read to proceed and then the subsequent write to execute silently, without sending invalidations. Later, before the



**Speculation, Thread-Level. Fig. 3** RAW data-dependence violation that results in a squash (a) or that does not cause a squash due to false sharing or value prediction (b), or consumer stall (c and d)

consumer task is allowed to commit, it is necessary to check whether the sections of the line read by the consumer overlap with the sections of the line written by the producer. This can be easily done if the caches have per-word access bits. If there is no overlap, it was false sharing and the squash is avoided. Figure 3b shows the resulting time line.

When there is a true data dependence between tasks, a squash can be avoided with effective use of value prediction. Specifically, the predictor can predict the value that the producer will produce, speculatively provide it to the consumer's read, and let the consumer proceed. Again, before the consumer is allowed to commit, it is necessary to check that the value provided was correct. The timeline is also shown in Fig. 3b.

In cases where the predictor is unable to predict the value, it can avoid the squash by stalling the consumer task at the time of the read. This case can use two possible approaches. An aggressive approach is to release the consumer task and let it read the current value as soon as the predicted producer task commits. The time line is shown in Fig. 3c. In this case, if an intervening task between the first producer and the consumer later writes the line, the consumer will be squashed. A more conservative approach is not to release the consumer task until it becomes nonspeculative. In this case, the presence of multiple predecessor writers will not squash the consumer. The time line is shown in Fig. 3d.

## Initial Efforts in Thread-Level Speculation

An early proposal for hardware support for a form of speculative parallelization was made by Knight [16] in the context of functional languages. Later, the Multiscalar processor [27] was the first proposal to use a form of TLS within a single-chip multithreaded architecture. A software-only form of TLS was proposed in the LRPD test [25]. Early proposals of hardware-based TLS include the work of several authors [14, 17, 21, 29, 35].

## Other Uses of Thread-Level Speculation

TLS concepts have been used in environments that have goals other than trying to parallelize sequential programs. For example, they have been used to speed up explicitly parallel programs through Speculative Synchronization [22], or for parallel program

debugging [24] or program monitoring [37]. Similar concepts to TLS have been used in systems supporting hardware transactional memory [15] and continuous atomic-block operation [30].

## Machines that Use Thread-Level Speculation

Several machines built by computer manufacturers have hardware support for some form of TLS – although the specific implementation details are typically not disclosed. Such machines include systems designed for Java applications such as Sun Microsystems' MAJC chip [31] and Azul Systems' Vega processor [2]. The most high-profile system with hardware support for speculative threads is Sun Microsystems' ROCK processor [3]. Other manufacturers are rumored to be developing prototypes with similar hardware.

## Bibliography

1. Akkary H, Driscoll M (1998) A dynamic multithreading processor. In: International symposium on microarchitecture, Dallas, November 1998
2. Azul Systems. Vega 3 Processor. <http://www.azulsystems.com/products/vega/processor>
3. Chaudhry S, Cypher R, Ekman M, Karlsson M, Landin A, Yip S, Zeffer H, Tremblay M (2009) Simultaneous speculative threading: a novel pipeline architecture implemented in Sun's ROCK Processor. In: International symposium on computer architecture, Austin, June 2009
4. Cintra M, Martínez JF, Torrellas J (2000) Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In: International symposium on computer architecture, Vancouver, June 2000, pp 13–24
5. Cintra M, Torrellas J (2002) Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In: Proceedings of the 8th High-Performance computer architecture conference, Boston, Feb 2002
6. Figueiredo R, Fortes J (2001) Hardware support for extracting coarse-grain speculative parallelism in distributed shared-memory multiprocessors. In: Proceedings of the international conference on parallel processing, Valencia, Spain, September 2001
7. Frank M, Lee W, Amarasinghe S (2001) A software framework for supporting general purpose applications on raw computation fabrics. Technical report, MIT/LCS Technical Memo MIT-LCS-TM-619, July 2001
8. Franklin M, Sohi G (1996) ARB: a hardware mechanism for dynamic reordering of memory references. IEEE Trans Comput 45(5):552–571

9. Garcia C, Madriles C, Sanchez J, Marcuello P, Gonzalez A, Tullsen D (2005) Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. In: Conference on programming language design and implementation, Chicago, Illinois, June 2005
10. Garzarán M, Prvulovic M, Llabería J, Viñals V, Rauchwerger L, Torrellas J (2005) Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *ACM Trans Archit Code Optim*
11. Garzaran MJ, Prvulovic M, Llabería JM, Viñals V, Rauchwerger L, Torrellas J (2003) Using software logging to support multi-version buffering in thread-level speculation. In: International conference on parallel architectures and compilation techniques, New Orleans, Sept 2003
12. Gopal S, Vijaykumar T, Smith J, Sohi G (1998) Speculative versioning cache. In: International symposium on high-performance computer architecture, Las Vegas, Feb 1998
13. Gupta M, Nim R (1998) Techniques for speculative run-time parallelization of loops. In: Proceedings of supercomputing 1998, ACM Press, Melbourne, Australia, Nov 1998
14. Hammond L, Willey M, Olukotun K (1998) Data speculation support for a chip multiprocessor. In: International conference on architectural support for programming languages and operating systems, San Jose, California, Oct 1998, pp 58–69
15. Herlihy M, Moss E (1993) Transactional memory: architectural support for lock-free data structures. In: International symposium on computer architecture, IEEE Computer Society Press, San Diego, May 1993
16. Knight T (1986) An architecture for mostly functional languages. In: ACM lisp and functional programming conference, ACM Press, New York, Aug 1986, pp 500–519
17. Krishnan V, Torrellas J (1998) Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In: International conference on supercomputing, Melbourne, Australia, July 1998
18. Krishnan V, Torrellas J (1999) A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans Comput* 48(9):866–880
19. Liu W, Tuck J, Ceze L, Ahn W, Strauss K, Renau J, Torrellas J (2006) POSH: A TLS compiler that exploits program structure. In: International symposium on principles and practice of parallel programming, San Diego, Mar 2006
20. Marcuello P, Gonzalez A (1999) Clustered speculative multi-threaded processors. In: International conference on supercomputing, Rhodes, Island, June 1999, pp 365–372
21. Marcuello P, Gonzalez A, Tubella J (1998) Speculative multi-threaded processors. In: International conference on supercomputing, ACM, Melbourne, Australia, July 1998
22. Martinez J, Torrellas J (2002) Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In: International conference on architectural support for programming languages and operating systems, San Jose, Oct 2002
23. Prvulovic M, Garzaran MJ, Rauchwerger L, Torrellas J (2001) Removing architectural bottlenecks to the scalability of speculative parallelization. In: Proceedings of the 28th international symposium on computer architecture (ISCA'01), New York, June 2001, pp 204–215
24. Prvulovic M, Torrellas J (2003) ReEnact: using thread-level speculation to debug data races in multithreaded codes. In: International symposium on computer architecture, San Diego, June 2003
25. Rauchwerger L, Padua D (1995) The LRPD test: speculative runtime parallelization of loops with privatization and reduction parallelization. In: Conference on programming language design and implementation, La Jolla, California, June 1995
26. Rundberg P, Stenstrom P (2000) Low-cost thread-level data dependence speculation on multiprocessors. In: Fourth workshop on multithreaded execution, architecture and compilation, Monterrey, Dec 2000
27. Sohi G, Breach S, Vijaykumar T (1995) Multiscalar processors. In: International Symposium on computer architecture, ACM Press, New York, June 1995
28. Steffan G, Colohan C, Zhai A, Mowry T (2000) A scalable approach to thread-level speculation. In: Proceedings of the 27th Annual International symposium on computer architecture, Vancouver, June 2000, pp 1–12
29. Steffan G, Mowry TC (1998) The potential for using thread-level data speculation to facilitate automatic parallelization. In: International symposium on high-performance computer architecture, Las Vegas, Feb 1998
30. Torrellas J, Ceze L, Tuck J, Cascaval C, Montesinos P, Ahn W, Prvulovic M (2009) The bulk multicore architecture for improved programmability. *Communications of the ACM*, New York
31. Tremblay M (1999) MAJC: microprocessor architecture for java computing. *Hot Chips*, Palo Alto, Aug 1999
32. Tsai J, Huang J, Amlo C, Lilja D, Yew P (1999) The superthreaded processor architecture. *IEEE Trans Comput* 48(9):881–902
33. Vijaykumar T, Sohi G (1998) Task selection for a multiscalar processor. In: International symposium on microarchitecture, Dallas, Nov 1998, pp 81–92
34. Zhai A, Colohan C, Steffan G, Mowry T (2002) Compiler optimization of scalar value communication between speculative threads. In: International conference on architectural support for programming languages and operating systems, San Jose, Oct 2002
35. Zhang Y, Rauchwerger L, Torrellas J (1998) Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors. In: Proceedings of the 4th International symposium on high-performance computer architecture (HPCA), Phoenix, Feb 1998, pp 162–174
36. Zhang Y, Rauchwerger L, Torrellas J (1999) Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors. In: Proceedings of the 5th international symposium on high-performance computer architecture, Orlando, Jan 1999, pp 135–139
37. Zhou P, Qin F, Liu W, Zhou Y, Torrellas (2004) iWatcher: efficient architectural support for software debugging. In: International symposium on computer architecture, IEEE Computer society, München, June 2004

## Speculative Multithreading (SM)

► [Speculation, Thread-Level](#)

## Speculative Parallelization

► [Speculation, Thread-Level](#)

► [Speculative Parallelization of Loops](#)

## Speculative Parallelization of Loops

LAWRENCE RAUCHWERGER  
Texas A&M University, College Station, TX, USA

### Synonyms

[Optimistic loop parallelization](#); [Parallelization](#); [Speculative Parallelization](#); [Speculative Run-Time Parallelization](#); [Thread-level data speculation \(TLDS\)](#); [TLS](#)

### Definition

Speculative loop (thread) level parallelization is a compiler run-time technique that executes optimistically parallelized loops, verifies the correctness of their execution and, when necessary, backtracks to a safe state for possible re-execution. This technique includes a compiler (static) component for the transformation of the loop for speculative parallel execution as well as a run-time component which verifies correctness and re-executes when necessary.

### Discussion

#### Introduction

The correctness of optimizing program transformations, such as loop parallelization, relies on compiler or programmer analysis of the code. Compiler-performed analysis is preferred because it is more productive and is not subject to human error. However, it usually involves a very complex symbolic analysis which often fails to produce an optimizing transformation. Sometimes the

outcome of the code analysis depends on the input values of the considered program block or on computed values, neither of which are available during compilation. Manual code analysis relies on the use of a higher level of semantic analysis which is usually more powerful but not applicable if it depends on run-time computed or input values. To overcome this limitation, the analysis can be (partially) performed at run-time. Run-time analysis can succeed where static analysis fails because it has access to the actual values with which *the program symbolic expressions* are instantiated and thus can make aggressive, instance-specific optimization (e.g., loop parallelization) decisions.

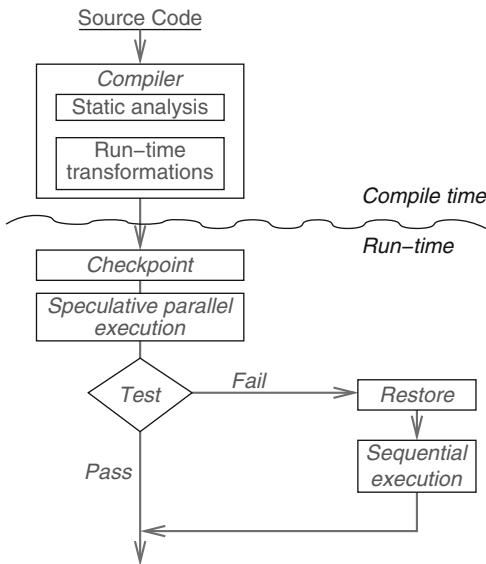
There are essentially two ways of performing the run-time analysis that can (in)validate loop parallelization (and, in general any other optimization):

- Before executing the parallel version of a loop
- During execution of the parallel version of the loop

The analysis that is performed before loop execution can be of various degrees of complexity: from constant time to time proportional to the original loop computation. The parallel execution of a loop before run-time analysis is completed is called speculative (aka optimistic) execution. It represents a speculation because the outcome of the analysis may, in the end, invalidate the (optimistic) parallel execution of the loop (and its results). In this case the state before the speculation has to be restored and the loop is re-executed in a safe manner, e.g., sequentially. [Figure 1](#) shows the global flowchart of the speculative parallelization process.

#### Fundamentals of Loop Parallelization

A loop can be executed in parallel, without synchronizations, if and only if the desired outcome of the loop does not depend upon the relative execution order of the data accesses across its iterations. This problem has been modeled with the help of data dependence analysis [1, 13, 19, 42, 48]. There are three possible types of dependences between two statements that access the same memory location: *flow* (read after write – RAW), *anti* (write after read – WAR), and *output* (write after write – WAW). Flow dependences express a fundamental relationship about the data flow in the program. Anti and output dependences, also known as memory-related



**Speculative Parallelization of Loops.** Fig. 1 Speculative run-time parallelization

dependences, are caused by the reuse of storage for program variables.

When flow data dependences exist between loop iterations, the loop cannot be executed in parallel because the original (sequential) semantics of the loop cannot be preserved. For example, the iterations of the loop in Fig. 2a must be executed in sequential order because iteration  $i+1$  needs the value that is produced in iteration  $i$ , for  $1 \leq i < n$ . The simplest and most desired outcome of the data dependence analysis is when there are no anti, output, or flow dependences. In this case, all the iterations of the loop are independent and can be executed in any order, e.g., concurrently. Such loops are known as DOALL loops. In the absence of flow dependences, which are fundamental to a program's algorithms, the anti and/or output dependences can be removed through *privatization* (or *renaming*) [40], a very effective loop transformation.

*Privatization* creates, for each processor cooperating on the execution of the loop, private copies of the program variables that can give rise to anti-or output dependences (see, e.g., [2, 16, 17, 38, 39]). Figure 2b exemplifies a loop that can be executed in parallel after applying the privatization transformation: The anti-dependences between statement S2 of iteration  $i$  and statement S1 of iteration  $i + 1$ , for  $1 \leq i < n/2$ , are removed by privatizing the temporary variable  $\text{tmp}$ .

This transformation can be applied to a loop variable if it can be proven, statically or dynamically, that every read access to it (e.g., elements of array  $A$ ) is preceded by a write access to the same variable within the same iteration. Of course, variables that are never written (read only) cannot generate any data dependence. Intuitively, variables are privatizable when they are used as workspace (e.g., temporary variables) *within* an iteration.

A semantically higher level transformation is the parallelization of *reduction* operations. Reductions are operations of the form  $x = x \otimes \text{exp}$ , where  $\otimes$  is an associative operator and  $x$  does not occur in  $\text{exp}$  or anywhere else in the loop. A simple, but typical example of a reduction is statement S1 in Fig. 2c. The operator  $\otimes$  is exemplified by the  $+$  operator, the access to the array  $A(:)$  is a *read, modify, write* sequence, and the function performed by the loop is a prefix sum of the values stored in  $A$ . This type of reduction is sometimes called an *update* and occurs quite frequently in programs.

A reduction can be readily transformed into a parallel operation using, e.g., a recursive doubling algorithm [12, 15]. When the operator takes the form  $x = x + \text{exp}$ , the values taken by variable  $x$  can be accumulated in private storage followed by a global reduction operation. There are also other, less scalable methods that use unordered critical sections [7, 48]. When the operator is also *commutative*, its substitution with a parallel algorithm can be done with fewer restrictions (e.g., dynamic scheduling of DOALL loops can be employed).

Thus, the difficulty encountered by compilers in parallelizing loops with reductions arises not from transforming the loop for parallel execution but from correctly identifying and validating reduction patterns in loops. This problem has been handled at compile time mainly by syntactically pattern matching the loop statements with a template of a generic reduction, and then performing a data dependence analysis of the variable under scrutiny to guarantee that it is not used anywhere else in the loop except in the reduction statement and thus does not cause additional dependences [48].

## Compiler Limitation and Run-time Parallelization

In essence, the parallelization of DO (for) loops depends on proving that their memory reference

|                                                                                                                                                          |                                                                                                                        |                                                                                                                  |
|----------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|
| <pre> <b>do</b> i=1, n   A(K(i)) = A(K(i)) + A(K(i-1))   <b>if</b> (A(K(i))) .eq. 0) <b>then</b>     B(i) = A(L(i))   <b>endif</b> <b>enddo</b> a </pre> | <pre> <b>do</b> i = 1, n/2   S1:   tmp = A(2*i)         A(2*i) = A(2*i-1)   S2:   A(2*i-1) = tmp <b>enddo</b> b </pre> | <pre> <b>do</b> i=1, n   <b>do</b> j = 1, m     S1:     A(j) = A(j) + exp()   <b>enddo</b> <b>enddo</b> c </pre> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|

**Speculative Parallelization of Loops.** Fig. 2 Examples of representative loops targeted by automatic parallelization

patterns do not carry data dependences or that there are legal transformations that can remove possible data dependences. The burden of proof relies on the static analysis performed by autoparallelizing compilers. However, there are many situations when this is not possible because either symbolic analysis is not powerful enough or the memory reference pattern is input or data dependent. The technique that can overcome such problems is run-time parallelization because, at run-time, all input values are known and the symbolic evaluation of complex expressions is vastly simplified. In this scenario, the compiler generates, conceptually at least, a parallel and a serial version of the loop as well code for the dynamic dependence analysis using the loop's memory references.

If the decision about the serial or parallel character of the loop is (and can be) made before its execution, then the results computed and written by the loop to memory can be considered to be always correct. Such a technique is called “inspector/executor” [31–33] because the memory references are first “inspected” and then executed in a safe manner, whether sequentially or concurrently. In this context, an “inspector” is obtained by extracting a loop slice that generates (and perhaps records) the relevant memory addresses which can then reveal possible loop-carried data dependences. It is important for such “inspectors” to be scalable and not to become serial bottlenecks. If no dependences are found, then the loop can be scheduled (and executed) as a DOALL, i.e., all its iterations can be executed concurrently. If dependences are found, then the “executor” of the loop has to enforce them using ordered synchronizations at the memory or iteration level such that sequential semantics is preserved. A frequently used solution [23, 24, 33] to this problem has been to first construct the loop dependence graph from the memory trace obtained by the inspector and then to use it to compute a parallel execution schedule.

Finally, the loop is executed in parallel according to the schedule.

The computation of this execution schedule can also be interleaved with the executor [46]. If the reference pattern does not change within a larger scope than the considered loop, then the schedule can be reused, thus reducing the impact of its overhead. There have been several variations of this technique which were reported in [22].

If a loop is executed in parallel *before* its data dependences are uncovered, then it can cause out of order memory references which may lead to incorrect computation. Such an execution model is called *speculative execution*, also known as *optimistic execution*, because its performance is based on the the optimistic assumption that such dependences do in fact not materialize or are quite infrequent. To ensure that even when dependences may occur, the final computation produces sequentially equivalent results, the speculative execution model includes a *restart* from a safe (correct) state mechanism. Such a mechanism implies either saving state (checkpointing) before its speculative modification or writing into a temporary memory which has to be later merged (committed) into the global state of the program.

For example, the references to array *A* in the loop in Fig. 2a depend on some input values stored in array *K* and cannot be statically analyzed by the compiler. An inspector for this loop would analyze the contents of array *K* and decide whether the loop is parallel and then execute it accordingly. A speculative approach is to execute the loop in parallel while at the same time recording all the references to *A*. After the loop has ended, the memory trace is checked for data dependences and, if any are found, the results are discarded and the loop is re-executed sequentially or in some other safer mode. Alternatively, the memory references can be checked as they occur (“on-the-fly”) and, if dependences are

detected the execution can be aborted at that point, the program state repaired and the loop restarted in a safe manner.

There are advantages and disadvantages to using either of the run-time parallelization methods. In the “inspector/executor” approach, the inspector does not modify the program state and thus does not require a restart mechanism with its associated memory overhead. On the other hand, a speculative approach may need to discard its results and restart from a safe state. This implies the allocation of additional memory for checkpointing or buffering state. Inspectors always add to the critical path of the program because they have to be inserted serially before the parallelized loop. Speculative parallelization performs the needed inspection of the memory references *during* the parallel execution almost independently from the actual computation and thus can be almost overlapped with it (assuming available resources). On the other hand, the checkpoint or commit phase introduces some overhead which may add to the critical path. What is perhaps the most important feature of speculative parallelization is its general applicability, even when the memory reference pattern is dependent on the computation of the loop. For example, the code snippet in Fig. 3 shows that the reference to array *NUSED* is dependent on its value which in turn, may have been modified in a different iteration (because the indirection array *IHITS* is not known at compile time).

There are two ways to look at speculation: optimistically assuming that a loop is fully parallel and can be executed as a DOALL or, pessimistically, assuming that the loop has dependences and must be executed as a DOACROSS, i.e., with synchronizations. When a DOALL is expected, then the overhead of the data checking can be done once, after the speculative loop

```

read (IHITS(:))
do k= 1, LST
 j = IHITS(1,k)
 if (NUSED(j).LE.1) then
 NUSED(j)=NUSED(j) -1
 endif
enddo

```

**Speculative Parallelization of Loops.** Fig. 3 A loop example where only speculative parallelization is possible: array indexes are computed during loop execution

has finished. If, however, dependences are expected, then they need to be detected early so that the resulting incorrect computation can be minimized. Early detection means frequent memory reference checks, well before the speculative loop has finished. These speculative approaches are bridged (transformed into one another) through the variation of the frequency (granularity) of the memory reference checks from once per loop to once per reference.

A related, but different limitation of compilers is their inability to analyze and parallelize most while loops. The reason is twofold:

- The classical loop dependence analysis looks for dependences in a bounded iteration space. However, the upper bound of a while loop can only be conservatively established at compile time which results in overly restrictive decisions, possibly inhibiting parallelization.
- The fully parallel execution of a while loop without data dependences (e.g., do loops with possible premature exits) may not be limited to the original iteration space. Iterations may be executed beyond their sequential upper bound, i.e., “overshoot” and thus incorrectly modify the global state.

A possible solution is to use a speculative parallel execution framework which allows discarding any unnecessary work and its effects. Speculation can also be used to estimate an upper bound of the iteration space of while loops.

## DOALL Speculative Parallelization: The LRPD Test

The optimistic version of speculative parallelization executes a loop in parallel and tests *subsequently* if any data dependences could have occurred. If this validation test fails, then the loop is re-executed in a safe manner, starting from a safe state, e.g., sequentially from a previous checkpoint. This approach, known as the *LRPD Test* (Lazy Reduction and Privatization Doall Test), [25, 27] is sketched in the next paragraph. To qualify more loops as parallel, *array privatization* and *reduction parallelization* can be speculatively applied and their validity tested after loop termination.

Consider a do loop for which the compiler cannot statically determine the access pattern of a shared array *A* (Fig. 4a). The compiler allocates the shadow arrays for

```

do i=1, 5
 z = A(K(i))
 if (B(i) .eq. .true.) then
 A(L(i)) = z +C(i)
 endif
enddo
 B(1:5) = (1 0 1 0 1)
 K(1:5) = (1 2 3 4 1)
 L(1:5) = (2 2 4 4 2)

```

a

```

doall i=1, 5
 markread(K(i))
 z = A(K(i))
 if (B(i) .eq. .true.) then
 markwrite(L(i))
 A(L(i)) = z +C(i)
 endif
enddoall

```

b

| PD test                   | Shadow arrays |   |   |   | <i>tw</i> | <i>tm</i> |
|---------------------------|---------------|---|---|---|-----------|-----------|
|                           | 1             | 2 | 3 | 4 |           |           |
| $A_w$                     | 0             | 1 | 0 | 1 | 3         | 2         |
| $A_r$                     | 1             | 1 | 1 | 1 |           |           |
| $A_{np}$                  | 1             | 1 | 1 | 1 |           |           |
| $A_w(:) \wedge A_r(:)$    | 0             | 1 | 0 | 1 |           |           |
| $A_w(:) \wedge A_{np}(:)$ | 0             | 1 | 0 | 1 |           |           |

c

**Speculative Parallelization of Loops.** Fig. 4 Do loop (a) transformed for speculative execution, (b) the markwrite and markread operations update the appropriate shadow arrays, (c) shadow arrays after loop execution. In this example, the test fails

marking the write accesses,  $A_w$ , and the read accesses,  $A_r$ , and an array  $A_{np}$ , for flagging non-privatizable elements. The loop is augmented with code (Fig. 4b) that during the speculative execution will mark the shadow arrays every time  $A$  is referenced (based on specific rules). The result of the marking can be seen in Fig. 4c. The first time an element of  $A$  is written during an iteration, the corresponding element in the write shadow array  $A_w$  is marked. If, during any iteration, an element in  $A$  is read, but never written, then the corresponding element in the read shadow array  $A_r$  is marked. Another shadow array  $A_{np}$  is used to flag the elements of  $A$  that cannot be privatized: An element in  $A_{np}$  is marked if, in any iteration, the corresponding element in  $A$  has been written only after it has been read.

A post-execution analysis, illustrated in Fig. 4c, determines whether there were any cross-iteration dependencies between statements referencing  $A$  as follows. If  $\text{any}(A_w(:) \wedge A_r(:))$  is true, (*any* returns the “OR” of its vector operand’s elements, i.e.,  $\text{any}(v(1 : n)) = (v(1) \vee v(2) \vee \dots \vee v(n))$ ) then there is at least one flow- or anti-dependence that was not removed by privatizing  $A$  (some element is read and written in different iterations). If  $\text{any}(A_{np}(:))$  is true, then  $A$  is not privatizable (some element is read before being written in an iteration). If  $tw$ , the total number of writes marked during the parallel execution, is not equal to  $tm$ , the total number of marks computed after the parallel execution, then there is at least one output dependence (some element is overwritten); however, if  $A$  is privatizable (i.e., if  $\text{any}(A_{np}(:))$  is false), then these dependencies were removed by privatizing  $A$ .

The addition of an  $A_{rx}$  field to the shadow structure and some simple marking logic can extend the previous

algorithm to validate parallel reductions at run-time [25, 27].

For this speculative technique two *safe restart* methods can be used [21, 25].

- The compiler either generates a checkpoint of all global, modified variables before the starting the speculative loop or does it, on demand, before a variable is modified the first time.
- The compiler generates code to allocate temporary, private storage for the global, modified variables. If the test fails the private storage is de-allocated, else its contents are merged into the global storage.

The data structures used for the checkpointing and shadowing can be appropriately chosen depending on the dense (e.g., arrays) or sparse reference characteristics of the loop (e.g., hash tables, linked lists) [44].

Compiler analysis can reduce the overhead of speculation by establishing equivalence classes (in the data dependence sense) of the interesting memory references and then tracking only one representative per class [44].

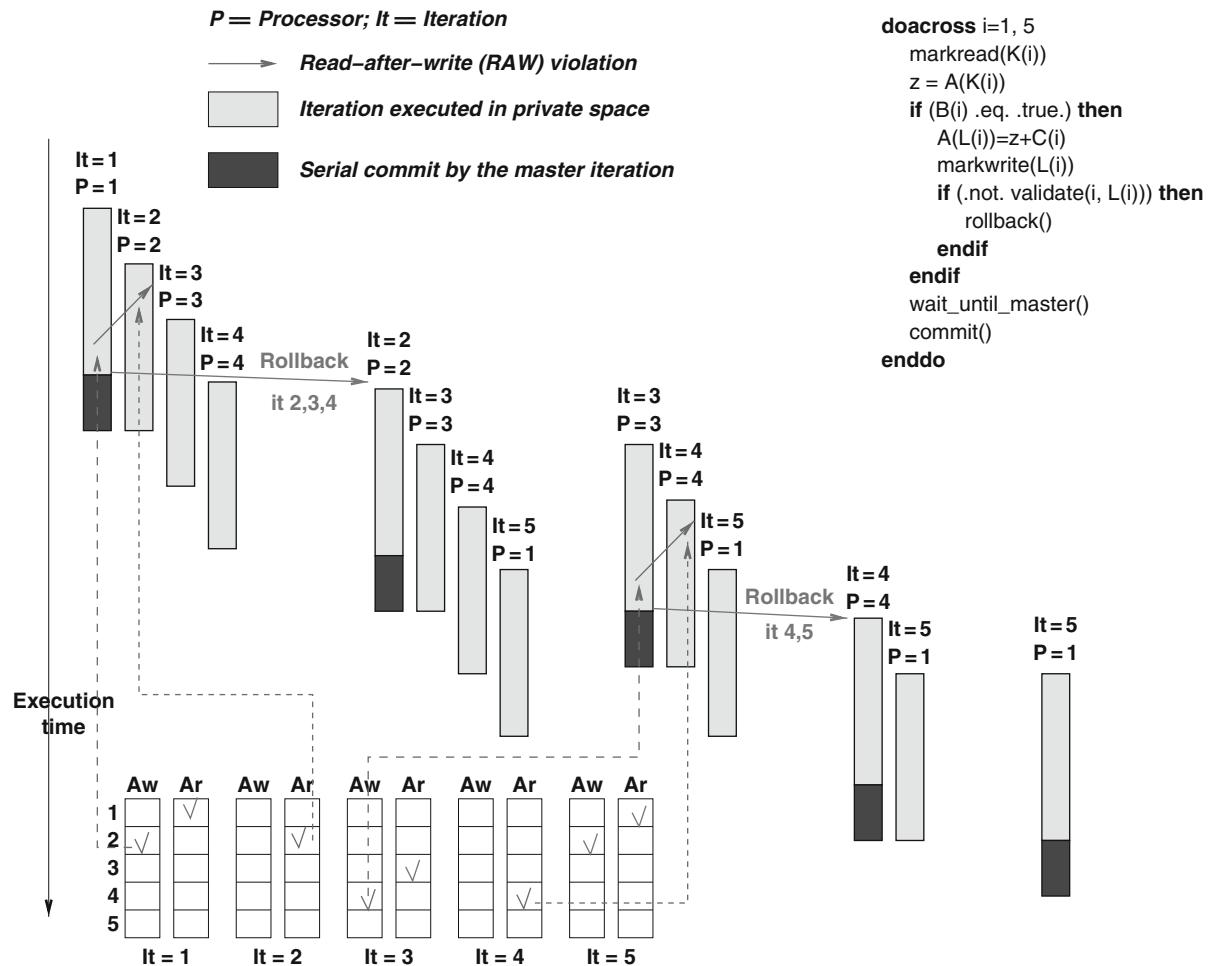
The speculative LRPD test has been later modified into the R-LRPD (Recursive LRPD) test [6] to improve its performance for loops with dependences. When a speculative parallel loop is block scheduled, the R-LRPD test can detect the iteration that is the source of the earliest dependence and thus validate the execution of the loop up to it. The remainder of the loop is then speculatively re-executed in a recursive manner until all work has finished, thus guaranteeing, at worst, a serial time complexity.

## DOACROSS Speculative Parallelization

If dependences are expected, then speculation should be verified frequently, so that incorrect iterations can be restarted as soon as violations have occurred, thus reducing the overall speculation overhead. Approaches that track dependences at low level (access/iteration), usually simulate the operation of a cache coherence protocol-based (hardware-based) TLS (Thread Level Speculation). Figure 5 depicts a sliding-window approach: The updates are recorded in per-iteration private storage and are merged (*committed*) to global storage by the oldest executing iteration, referred to as the *master*. Since the master represents a nonspeculative thread, i.e., it cannot be invalidated by any future

iteration, its updates are known to be correct and the copy-out operation is safe. Then, the next iteration becomes nonspeculative and can eventually commit its updates, etc.

While implementations are diverse, one approach [4, 28] could be that (1) a read access returns the value written by the closest predecessor iteration (*forwarding*), or the nonspeculative value if no such write exists, and (2) a write access to memory location 1 signals a flow dependence violation if a successor iteration has read from location 1. This behavior is achieved by inspecting the shadow vectors  $A_w$  and  $A_r$ , which record per-iteration write/read accesses. For example, in Fig. 5, the call of the function validate by iteration



**Speculative Parallelization of Loops.** Fig. 5 Serial-commit, sliding-window-based Thread Level Speculation execution.

Dashed, red arrows starting from  $A_w/A_r$  represent the source/sink of flow dependences

1, which writes  $A[2]$ , detects a flow dependence violation because iteration 2 already read  $A[2]$ . The master is always correct, so iteration 1 commits its updates, but restarts iterations 2, 3, and 4. Similarly, iterations 3 and 4 are the source and sink of another flow dependence violation.

While maintaining per-iteration shadow vectors is prohibitively expensive in many cases, a sliding window of size  $C$  will reduce the memory overhead to more manageable levels. In particular, since only  $C$  consecutive iterations may execute concurrently, only  $O(C)$  sets of shadow vectors need to be maintained, and then recycled.

Speculative DOACROSS methods are best suited for loops with more frequent data dependences because they can detect dependences earlier thus reducing wasted computation. However, verifying the dependence violation for each memory reference can be quite expensive because it requires global synchronizations. Furthermore, the merge (commit) phase is done in iteration order which constitutes a serial bottleneck. Loops with frequent dependences are not candidates for scalable parallelization regardless of the methods used to detect and process them. They cause, aside from lack of parallelism, frequent back tracking which rapidly degrades performance to possible negative levels.

## While Loop Speculative Parallelization

While loops and do loops with conditional premature exits arise frequently in practice and techniques for extracting their available parallelism [5, 26] are highly desirable. In the most general form, a while loop can be defined as a loop that includes at least one *recurrence*, a *remainder*, and at least one *termination condition*. The dominating recurrence, which precedes the rest of the computation is called the *dispatching recurrence*, or simply the *dispatcher* (Fig. 6a).

Sometimes the termination conditions form part of one of the recurrences, but they can also occur in the remainder, e.g., *conditional exits* from do loops. Assuming, for simplicity, that the *remainder* is fully parallelizable, there are two potential problems in the parallelization of while constructs:

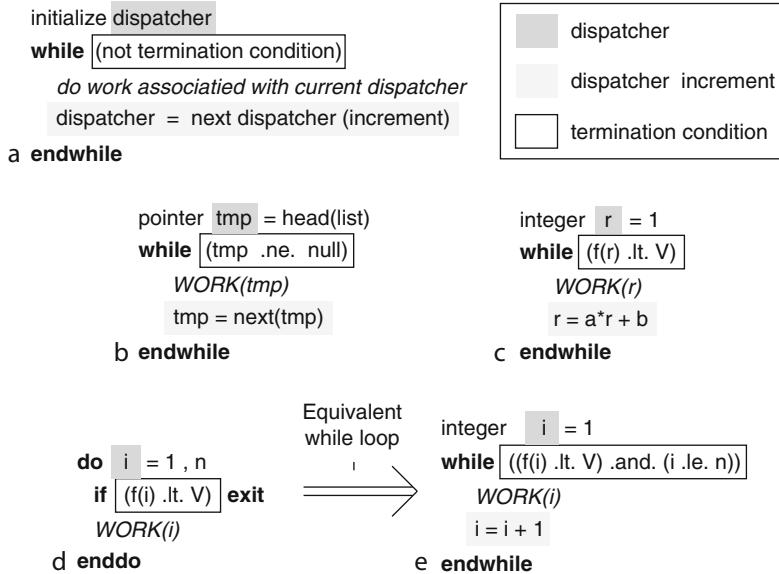
- *Evaluating the recurrences.* If the recurrences cannot be evaluated in parallel, then the iterations of the

loop must be started sequentially, leading in the best case to a pipelined execution.

- *Evaluating the termination conditions.* If the termination conditions (loop exits) cannot be evaluated independently by all iterations, the parallelized while loop could continue to execute beyond the point where the original sequential loop would stop, i.e., it can *overshoot*.

*Evaluating the recurrences concurrently.* In general, the terms of the dispatcher must be evaluated sequentially, e.g., the pointer chasing when traversing a linked list. Because the values of the dispatcher (the pointer) must be evaluated in sequential order, iteration  $i$  of the loop cannot be initiated until the dispatcher for iteration  $i - 1$  has been computed (see Fig. 6b). However, if the dispatching recurrence is associative then it can be evaluated in parallel using, e.g., a parallel prefix algorithm (see Fig. 6c). Better yet, when the dispatcher takes the form of an induction, its values can be computed concurrently by evaluating its symbolic form and thus allowing all iterations of the while loop to execute in parallel. A typical example is represented by a do loop (see Fig. 6d, e).

*Evaluating the termination conditions in parallel.* Another difficulty with parallelizing while loops is that the termination condition (*terminator*) of the loop may be *overshot*, i.e., the loop would continue to execute beyond its sequential (original) counterpart. The *terminator* is defined as *remainder invariant*, or RI, if it is only dependent on the dispatcher and values that are computed outside the loop. If it is dependent on some value computed in the loop, then it is considered to be *remainder variant* or RV. If the *terminator* is RV, then iterations larger than the last (sequentially) valid iteration could be performed in a parallel execution of the loop, i.e., iteration  $i$  cannot decide if the *terminator* is satisfied in the remainder of some iteration  $i' < i$ . Overshooting may also occur if the dispatcher is an induction, or an associative recurrence, and the *terminator* is RI. An exception in which overshooting would not occur is if the dispatcher is a monotonic function, and the *terminator* is a threshold on this function, e.g.,  $d(i) = i^2$ , and  $tc(i) = (d(i) < V)$ , where  $V$  is a constant, and  $d(j)$  and  $tc(j)$  denote the dispatcher and the *terminator*, respectively, for the  $j$ th iteration. Overshooting can also



**Speculative Parallelization of Loops.** Fig. 6 (a) The general structure of `while` loops (b) Pointer chasing (c) Threshold terminator with monotonic dispatcher (d) DO loop with premature exit and (e) its equivalent `while` loop

be avoided when the dispatcher is a general recurrence, and the *terminator* is RI. For example, the dispatcher *tmp* is a pointer used to traverse a linked list, and the *terminator* is (*tmp* = *null*) (see Fig. 6b). The parallelization potential of the *dispatcher* is summarized by the taxonomy of `while` loops given in Table 1.

*Speculative while loop Parallelization.* Executing a loop outside its intended iteration space may result in undesired global state modification which has to be later undone. Such parallel execution can be regarded as *speculative* and is handled similarly to the previously mentioned speculative `do` loop parallelization.

The effects of *overshooting* in speculative parallel `while` loops can be undone after loop termination by restoring the (sequentially equivalent) correct state from a trace of the time (iteration) stamped memory references recorded during speculative execution. This solution may, however, have a large memory overhead for the time-stamped memory trace.

Alternatively, shared variables can be written into temporary storage, and then copied out only if their time stamps are less than or equal to the last valid iteration. There are various techniques for reducing memory overhead that can take advantage of the specific memory reference pattern, e.g., sparse patterns can be stored

in hash tables. Further optimizations include *strip mining* the loop, or using a “sliding window” of iterations whose advance can be throttled adaptively.

An alternative to time-stamping is to attempt to extract a slice from the original `while` loop that can precompute the iteration space. When such a transformation is possible (e.g., traversal of a linked list) then the remainder of the `while` loop can be executed as a `doall`.

*While loops with statically unknown cross-iteration dependences.* When the data dependences of a `While` loop cannot be conclusively analyzed by a static compiler, the loop parallelization can be attempted by combining the LRPD test (applied to the *remainder* loop) with the techniques for `while` loop parallelization described above.

When it can be determined statically (see the taxonomy Table 1) that the parallelized `while` loop will not overshoot, then the shadow variable tracing instrumentation for the LRPD test can be inserted directly into the `while` loop.

In the more complex case when overshooting may occur (see the taxonomy Table 1) the LRPD test can be combined with the `while` loop parallelization methods by augmenting its shadow arrays with the minimum

**Speculative Parallelization of Loops. Table 1** A taxonomy of while loops and their dispatcher's potential for parallel execution. In the table, *mono* is monotonic, *OV* is overshoot, *P* is parallel, *N* is no, *Y* is yes, and *pp* means parallelizable with a parallel prefix computation. *RV* is remainder variant and *RI* is remainder invariant

| Loop terminator | Dispatcher |   |       |   |             |      |         |   |
|-----------------|------------|---|-------|---|-------------|------|---------|---|
|                 | Induction  |   |       |   | Recurrence  |      |         |   |
|                 | mono       |   | other |   | associative |      | general |   |
|                 | OV         | P | OV    | P | OV          | P    | OV      | P |
| RI              | N          | Y | Y     | Y | Y           | Y-pp | N       | N |
| RV              | Y          | Y | Y     | Y | Y           | Y-pp | Y       | N |

iteration that marked each element. The post-execution analysis of the LRPD test has to ignore the shadow array entries with minimum time stamps greater than the last valid iteration.

A special and unwelcome situation arises when the termination condition of the while loop is dependent (data or control) upon a variable which has **not** been found by the LRPD test to be independent. The speculative parallel execution of such a while loop may incorrectly compute its iteration space, or even worse, the termination condition might never be met (an infinite loop). Such loops are very hard to parallelize efficiently.

In general, while loops do not usually lend themselves to scalable parallelization due to their inherent nontrivial recurrence and overshooting potential which is expensive to mitigate.

### Speculative Parallelization as a Parallel Programming Paradigm

The speculative techniques presented thus far concern themselves with the automatic transformation of sequential loops into parallel ones. They can validate, at run-time, if an instance of a loop executed in parallel does not have potential dependences and thus has *exactly* the same output as its sequential original. This validation does not, and could not, realistically check if the parallel and sequential execution have the same results. However, it imposes conditions on the memory reference trace which can ensure that the parallel

loop respects the same dependence graph as the sequential one. In general, the speculative parallelization presented thus far represents a gamble that its results are equal to those of the sequential execution because it executes the same fine-grain algorithm, i.e., the same fine-grain dependence graph as the sequential original loop. This approach has lent itself well to automatic compiler implementation.

There is, however, a higher semantic level, albeit more complex, use of speculative parallelization. Instead of speculating on the execution of a fine-grain dependence graph (constructed from memory references), the programmer can generate the same results by following the correct execution of a coarser, semantically higher level dependence graph. For example, the nodes of such a coarse graph could represent the methods of a container and the edges the order in which they will be invoked. The programmer may also specify high-level properties of the methods involved. For example, the order of “inquire” type method invocations may be declared as interchangeable (commutative) similar with the *read* operations on an array. Other methods may be required to respect the program order, e.g., *add* and *delete*. In general, the cause-and-effect relation of the operations can be user defined. The result is a partial order of operations, a coarse dependence graph. A speculative parallelization of a program at this level of abstraction will ensure that its execution respects the higher level dependence graph. It implements a more relaxed execution model thus possibly improving performance. The Galois system implements this approach to speculative parallel programming [14]. It is based on two key abstractions: (1) *optimistic iterators* that can iterate over *sets* of iterations in any order and (2) a collection of assertions about properties of methods in class libraries. In addition a run-time scheme can detect and undo potentially unsafe operations due to speculation.

The *set iterator* abstracts the possibility selecting for execution elements of a set of iteration in any serial order (similar to the DOANY construct [43]). When an iteration executes it may add new iterations to the iteration space, hence allowing for fully dynamic execution. This is inherently possible because the addition of work can be inserted anywhere in the unordered set without changing the semantics of the loop. The *set iterator*

can become an *optimistic iterator* if the generated work can create conflicts at the iteration level. To exploit parallelism, the system relies on the semantic commutativity property of methods which must be declared by the programmer through special class interface method declarations. Finally, when the program executes, the methods are invoked like transactions (in isolation) and the commutativity property of the sequence of invoked methods is asserted. If it fails, then it is possible that operations have been executed in an illegal order and therefore have to be undone. The execution is rolled back using *anti-methods*, i.e., methods that undo the effects of the offending method. For example, an *add* operation can be undone by a *delete* operation. The entire system is reminiscent of Jefferson's Virtual Time system [10].

Speculating about higher level operations using higher level abstractions allows the exploitation of algorithmic parallelism otherwise impossible to exploit at the memory reference level. It requires, however, the user to reason more about the employed algorithm.

## Future Directions

Run-time parallelization in general, and speculative parallelization in particular, are likely to be essential techniques for both automatic and manual parallelization. Speculation has its drawbacks: The success rate of speculation is rather unpredictable and thus affects performance in a nondeterministic manner. Speculation may waste resources, including power, when it does not produce a speedup. However, given the ubiquitous parallelism encountered today, it is sometimes the only avenue of performance improvement.

It would be of great benefit if meaningful statistics about the success rate of speculative parallelization could be found. Speculation will continue to be used for manual parallelization of irregular programs [14], at least until good parallel algorithms will be found. In this case, high-level speculation is likely to produce better results and result in more expressive programs!.

Speculative parallelization has been integrated into the Hybrid Analysis compiler framework [29, 30]. This framework seamlessly integrates static and dynamic analysis to extract the minimum sufficient conditions that need to be evaluated at run-time in order to validate a parallel execution of a loop. By carefully staging these sufficient conditions in order of their run-time

complexity, the compiler often minimizes the need for a full speculative parallelization, thus reducing the non-determinism of the code performance.

In conclusion, speculative parallelization is a globally applicable parallelization technique that can make the difference between a fully and a partially parallelized program, i.e., between scalable and non-scalable performance.

## Related Entries

- ▶ Debugging
- ▶ Dependencies
- ▶ Dependence Analysis
- ▶ Parallelization, Automatic
- ▶ Race Conditions
- ▶ Run Time Parallelization
- ▶ Speculation, Thread-Level

## Bibliographic Notes and Further Reading

Speculative run-time parallelization has been first mentioned in the context of processes of parallel discrete event simulations by D. Jefferson [10]. In his *virtual time* concurrent processes are launched asynchronously and are tagged with their logical time stamp. When such processes communicate, they compare time stamps to check if their order respects the logical clock of the program (i.e., if data dependences are not violated). If a violation is detected, then anti-messages will recursively undo the effects of the incorrect computations and restart from a safe point.

The LRPD test, i.e., the speculative parallelization of loops, was introduced in [27]. It has later been modified to parallelize loops with dependences [6]. Compiler optimizations [21, 44] have lowered its overhead. The Hybrid Analysis framework has integrated the LRPD test in a compiler framework [30].

A significant amount of later work [4, 11, 20] has followed the hardware based approach to speculative parallelization presented in [8, 36, 37, 45].

Other related work reduces communication overhead via a master-slave model [47] in which the master executes an optimistic (fast) approximation of the code, while the slaves verify the master's correctness. Another framework [3, 41] exploits method-level parallelism for Java applications. The design space of speculative

parallelization has been further widened by allowing tunable memory overhead [18] by mapping more than one memory reference to the same “shadow” structure, but with the penalty of generating false dependence violations.

Software Transactional Memory [9, 35] can be and often is implemented using speculative parallelization techniques.

Debugging of parallel programs in general and detecting memory reference “anomalies” in particular [34] is a related topic to speculative parallelization and data dependence violation detection.

## Bibliography

1. Banerjee U (1988) Dependence analysis for supercomputing. Kluwer, Boston
2. Burke M, Cytron R, Ferrante J, Hsieh W (1989) Automatic generation of nested, fork-join parallelism. *J Supercomput* 2:71–88
3. Chen MK, Olukotun K (1998) Exploiting method level parallelism in single threaded java programs. In: International conference on parallel architectures and compilation techniques PACT’98, IEEE, Paris, pp 176–184
4. Cintra M, Llanos DR (2003) Toward efficient and robust software speculative parallelization on multiprocessors. In: International conference on principle and practice of parallel computing PPoPP’03, ACM, San Diego, pp 13–24
5. Collard J-F (1994) Space-time transformation of while-loops using speculative execution. In: Scalable high performance computing conference, IEEE, Knoxville, pp 429–436
6. Dang F, Yu H, Rauchwerger L (2002) The R-LRPD test: speculative parallelization of partially parallel loops. In: International parallel and distributed processing symposium, Florida
7. Eigenmann R, Hoeflinger J, Li Z, Padua D (1991) Experience in the automatic parallelization of four perfect-benchmark programs. Lecture notes in computer science 589. Proceedings of the fourth workshop on languages and compilers for parallel computing, Santa Clara, pp 65–83
8. Hammond L, Willey M, Olukotun K (1998) Data speculation support for a chip multiprocessor. In: 8th international conference on architectural support for programming languages and operating systems, San Jose, pp 58–69
9. Herlihy M, Shavit N (1995) The art of multiprocessor programming. Morgan Kaufmann, London
10. Jefferson DR (1985) Virtual time. *ACM Trans Program Lang Syst* 7(3):404–425
11. Kazi IH, Lilja DJ (2001) Coarsened-grained thread pipelining: a speculative parallel execution model for shared-memory multiprocessors. *IEEE Trans Parallel Distrib Syst* 12(9):952
12. Kruskal C (1986) Efficient parallel algorithms for graph problems. In: Proceedings of the 1986 international conference on parallel processing, University Park, pp 869–876, Aug 1986
13. Kuck DJ, Kuhn RH, Padua DA, Leisure B, Wolfe M (1981) Dependence graphs and compiler optimizations. In: Proceedings of the 8th ACM symposium on principles of programming languages, Williamsburg, pp 207–218
14. Kulkarni M, Pingali K, Walter B, Ramanarayanan G, Bala K, Paul Chew L (2007) Optimistic parallelism requires abstractions. In: Proceedings of the 2007 ACM SIGPLAN conference on programming language design and implementation, PLDI ’07, ACM, New York, pp 211–222
15. Thomson Leighton F (1992) Introduction to parallel algorithms and architectures: arrays, trees, hypercubes. Morgan Kaufmann, London
16. Li Z (1992) Array privatization for parallel execution of loops. In: Proceedings of the 19th international symposium on computer architecture, Gold Coast, pp 313–322
17. Maydan DE, Amarasinghe SP, Lam MS (1992) Data dependence and data-flow analysis of arrays. In: Proceedings 5th workshop on programming languages and compilers for parallel computing, New Haven
18. Oancea CE, Mycroft A, Harris T (2009) A lightweight in-place implementation for software thread-level speculation. In: International symposium on parallelism in algorithms and architectures SPAA’09, ACM, Calgary, pp 223–232
19. Padua DA, Wolfe MJ (1986) Advanced compiler optimizations for supercomputers. *Commun ACM* 29:1184–1201
20. Papadimitriou S, Mowry TC (2001) Exploring thread-level speculation in software: the effects of memory access tracking granularity. Technical report, CMU
21. Patel D, Rauchwerger L (1999) Implementation issues of loop-level speculative run-time parallelization. In: Proceedings of the 8th international conference on compiler construction (CC’99), Amsterdam. Lecture notes in computer science, vol 1575. Springer, Berlin
22. Rauchwerger L (1998) Run-time parallelization: its time has come. *Parallel Comput* 24(3–4):527. Special issues on languages and compilers for parallel comput
23. Rauchwerger L, Amato N, Padua D (1995) Run-time methods for parallelizing partially parallel loops. In: Proceedings of the 9th ACM international conference on supercomputing, Barcelona, Spain, pp 137–146
24. Rauchwerger L, Amato N, Padua D (1995) A scalable method for run-time loop parallelization. *Int J Parallel Prog* 26(6): 537–576
25. Rauchwerger L, Padua DA (1999) The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Trans Parallel and Distrib Syst* 10(2):160–180
26. Rauchwerger L, Padua DA (1995). Parallelizing WHILE loops for multi-processor systems. In: Proceedings of 9th international parallel processing symposium, Santa Barbara
27. Rauchwerger L, Padua DA (1995) The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In: Proceedings of the SIGPLAN 1995 conference on programming language design and implementation, La Jolla pp 218–232
28. Rundberg P, Stenstrom P (2000) Low-cost thread-level data dependence speculation on multiprocessors. In: 4th workshop on multithreaded execution, architecture and compilation, Monterey

29. Rus S, Pennings M, Rauchwerger L (2007) Sensitivity analysis for automatic parallelization on multi-cores. In: Proceedings of the ACM international conference on supercomputing (ICS07), Seattle
30. Rus S, Hoeflinger J, Rauchwerger L (2003) Hybrid analysis: static & dynamic memory reference analysis. *Int J Parallel Prog* 31(3):251–283
31. Saltz J, Mirchandaney R (1991) The preprocessed doacross loop. In: Schwetman HD (ed) *Proceedings of the 1991 international conference on parallel processing, Software*, vol II. CRC Press, Boca Raton, pp 174–178
32. Saltz J, Mirchandaney R, Crowley K (1989) The doconsider loop. In: *Proceedings of the 1989 international conference on supercomputing*, Irakleion, pp 29–40
33. Saltz J, Mirchandaney R, Crowley K (1991) Run-time parallelization and scheduling of loops. *IEEE Trans Comput* 40(5):603–612
34. Schonberg E (1989) On-the-fly detection of access anomalies. In: *Proceedings of the SIGPLAN 1989 conference on programming language design and implementation*, Portland, pp 285–297
35. Shavit N, Touitou D (1995) Software transactional memory. In: *Proceedings of the fourteenth annual ACM symposium on principles of distributed computing*, PODC '95, ACM, New York, pp 204–213
36. Sohi GS, Breach SE, Vijayakumar TN (1995) Multiscalar processors. In: *22nd international symposium on computer architecture*, Santa Margherita
37. Steffan JG, Mowry TC (1998) The potential for using thread-level data speculation to facilitate automatic parallelization. In: *Proceedings of the 4th international symposium on high-performance computer architecture*, Las Vegas
38. Tu P, Padua D (1992) Array privatization for shared and distributed memory machines. In: *Proceedings 2nd workshop on languages, compilers, and run-time environments for distributed memory machines*, Boulder
39. Tu P, Padua D (1993) Automatic array privatization. In: *Proceedings 6th annual workshop on languages and compilers for parallel computing*, Portland
40. Tu P, Padua D (1995) Efficient building and placing of gating functions. In: *Proceedings of the SIGPLAN 1995 conference on programming language design and implementation*, La Jolla, pp 47–55
41. Welc A, Jagannathan S, Hosking A (2006) Safe futures for Java. In: *International conference object-oriented programming, systems, languages and applications OOP-SLA'06*, ACM, New York, pp 439–453
42. Wolfe M (1989) Optimizing compilers for supercomputers. MIT Press, Boston
43. Wolfe M (1992) Doany: not just another parallel loop. In: *Proceedings 5th annual workshop on programming languages and compilers for parallel computing*, New Haven. Lecture notes in computer science, vol 757. Springer, Berlin
44. Yu H, Rauchwerger L (2000) Run-time parallelization overhead reduction techniques. In: *Proceedings of the 9th international conference on compiler construction (CC 2000)*, Berlin Germany. Lecture notes in computer science vol 1781. Springer, Heidelberg
45. Zhang Y, Rauchwerger L, Torrellas J (1998) Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors. In: *Proceedings of the 4th international symposium on high-performance computer architecture (HPCA)*, Las Vegas, pp 162–174
46. Zhu C, Yew PC (1987) A scheme to enforce data dependence on large multiprocessor systems. *IEEE Trans Softw Eng* 13(6): 726–739
47. Zilles C, Sohi G (2002) Master/slave speculative parallelization. In: *International symposium on microarchitecture Micro-35*, IEEE, Los Alamitos, pp 85–96
48. Zima H (1991) *Supercompilers for parallel and vector computers*. ACM Press, New York

## Speculative Run-Time Parallelization

- ▶ Speculation, Thread-Level
- ▶ Speculative Parallelization of Loops

## Speculative Threading

- ▶ Speculation, Thread-Level

## Speculative Thread-Level Parallelization

- ▶ Speculation, Thread-Level

## Speedup

- ▶ Metrics

## SPIKE

ERIC POLIZZI  
University of Massachusetts, Amherst, MA, USA

## Definition

SPIKE is a polyalgorithm that uses many different strategies for solving large banded linear systems

in parallel. Existing parallel algorithms and software using direct methods for banded matrices are mostly based on LU factorizations. In contrast, SPIKE uses a novel decomposition method (i.e., DS factorization) to balance communication overhead with arithmetic cost to achieve better scalability than other methods. The SPIKE algorithm is similar to a domain decomposition technique that allows performing independent calculations on each subdomain or partition of the linear system, while the interface problem leads to a reduced linear system of much smaller size than that of the original one. Direct, iterative, or approximate schemes can then be used to handle the reduced system in a different way depending on the characteristics of the linear system and the parallel computing platform.

## Discussion

### Introduction

Many science and engineering applications, particularly those involving finite element analysis, give rise to very large sparse linear systems. These systems can often be reordered to produce either banded systems or low-rank perturbations of banded systems in which the width of the band is but a small fraction of the size of the overall problem. In other instances, banded systems can act as effective preconditioners to general sparse systems, which are solved via iterative methods.

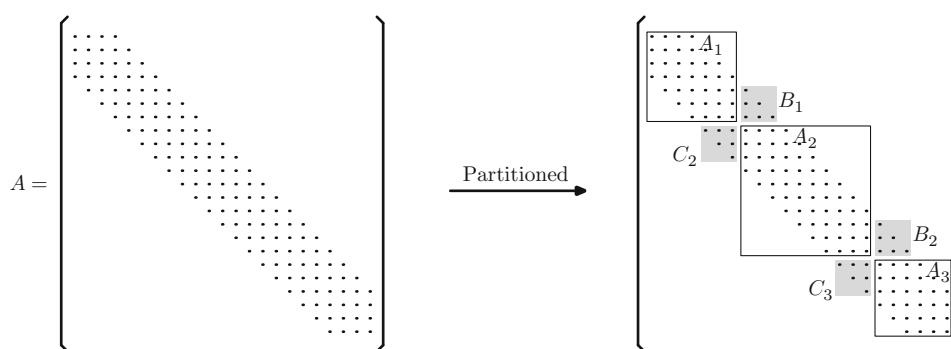
Direct methods for solving linear systems  $AX = F$  are commonly based on the LU decomposition that represents a matrix  $A$  as a product of lower and upper triangular matrices  $A = LU$ . Consequently, solving  $AX = F$  can be achieved by solutions of two triangular systems

$LG = F$  and  $UX = G$ . A parallel LU decomposition for banded linear systems has also been proposed by Cleary and Dongarra in [1] for the ScaLAPACK package [2]. The central idea behind the SPIKE algorithm is a different decomposition for banded linear systems, introduced by A. Sameh in the late 1970s [3], which is ideally suited for parallel implementation as it naturally leads to lower communication cost. Several enhancements and variants of the SPIKE algorithm have since been proposed by Sameh and coauthors in [4–11]. In the case when  $A$  is a banded matrix as depicted in Fig. 1, SPIKE is using a direct partitioning in the context of parallel processing.

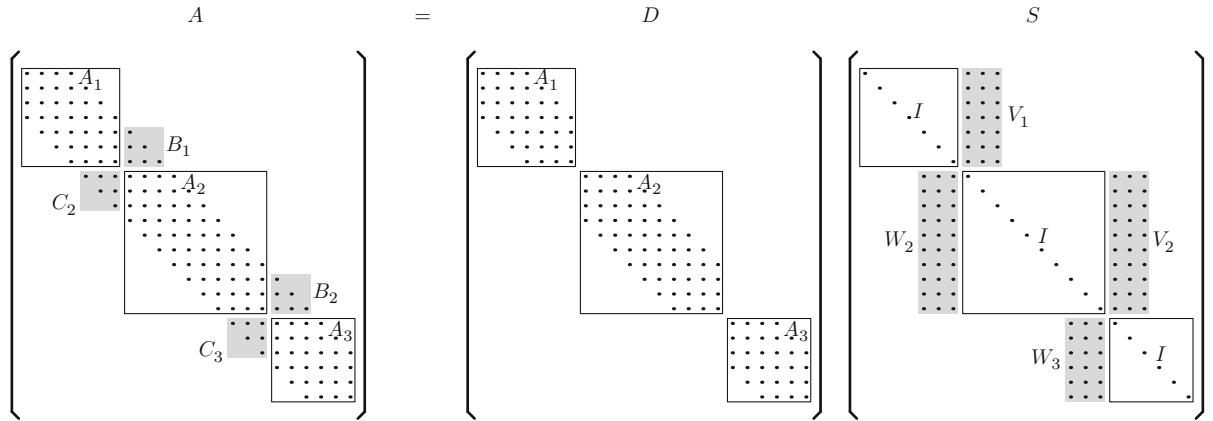
SPIKE relies on the decomposition of a given banded matrix  $A$  into the product of a block-diagonal matrix  $D$ , and another matrix  $S$  which has the structure of an identity matrix with some extra “spikes” (and hence the name of the algorithm). This DS factorization procedure is illustrated in Fig. 2.

Solving  $AX = F$  can then be accomplished in two steps:

1. Solution of block-diagonal system  $DG = F$ . Because  $D$  consists of decoupled systems of each of the diagonal block  $A_i$ , they can be solved in parallel without requiring any communication between the individual systems.
2. Solution of the system  $SX = G$ . This system has a wonderful characteristic that it is also decoupled to a large extent. Except for a reduced system (near the interface of each of the identity blocks), the rest are independent from one another. The natural way to tackle this system is to first solve the reduced system via some parallel algorithms that



SPIKE. Fig. 1 A banded matrix with a conceptual partition



**SPIKE. Fig. 2** SPIKE factorization where  $A = DS, S = D^{-1}A$ . The blocks in the block-diagonal matrix  $D$  are supposed non-singular

require inter-processor communications, followed by retrieval of the rest of the solution without requiring further inter-processor communications.

### The SPIKE Algorithm: Basics

As illustrated in Fig. 1, a  $(N \times N)$  banded matrix  $A$  can be partitioned into a block tridiagonal form  $\{C_j, A_j, B_j\}$ , where  $A_j$  is the  $(n_j \times n_j)$  diagonal block  $j$ , and  $B_j$  (i.e.,  $C_j$ ) is the  $(ku \times ku)$  (i.e.,  $(kl \times kl)$ ) right block (i.e., left block). Using  $p$  partitions, it comes that  $n_j$  is roughly equal to  $N/p$ . In order to ease the description of the SPIKE algorithm but without loss of generality, the size off-diagonal blocks are both supposed equal to  $m$  ( $kl = ku = m$ ). The size of the bandwidth is then defined by  $b = 2m + 1$  where  $b \ll n_j$ . Each partition  $j$  ( $j = 1, \dots, p$ ) can be associated to one processor or one node allowing multilevel of parallelism. Using the  $DS$  factorization illustrated in Fig. 2, the obtained spike matrix  $S$  has a block tridiagonal form  $\{W_j, I_j, V_j\}$ , where  $I_j$  is the  $(n_j \times n_j)$  identity matrix,  $V_j$  and  $W_j$  are the  $(n \times m)$  right and left spikes. The spikes  $V_j$  and  $W_j$  are solutions of the following linear systems:

$$A_j V_j = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ B_j \end{bmatrix}, \quad \text{and} \quad A_j W_j = \begin{bmatrix} C_j \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \quad (1)$$

respectively for  $j = 1, \dots, p-1$  and  $j = 2, \dots, p$ .

Solving the system  $AX = F$  now consists of two steps:

$$(a) \text{ solve } DG = F \quad (2)$$

$$(b) \text{ solve } SX = G. \quad (3)$$

The solution of the linear system  $DG = F$  in Step (a) yields the modified right-hand side  $G$  needed for Step (b). In case of assigning one partition to each processor, Step (a) is performed with perfect parallelism. To solve  $SX = G$  in Step (b), one should observe that the problem can be reduced further by solving a system of much smaller size, which consists of the  $m$  rows of  $S$  immediately above and below each partitioning line. Indeed, the spikes  $V_j$  and  $W_j$  can also be partitioned as follows

$$V_j = \begin{bmatrix} V_j^{(t)} \\ V_j' \\ V_j^{(b)} \end{bmatrix} \quad \text{and} \quad W_j = \begin{bmatrix} W_j^{(t)} \\ W_j' \\ W_j^{(b)} \end{bmatrix} \quad (4)$$

where  $V_j^{(t)}$ ,  $V_j'$ ,  $V_j^{(b)}$ , and  $W_j^{(t)}$ ,  $W_j'$ ,  $W_j^{(b)}$ , are the top  $m$ , the middle  $n_j - 2m$  and the bottom  $m$  rows of  $V_j$  and  $W_j$ , respectively. Here,

$$V_j^{(b)} = [0 \quad I_m] V_j; \quad W_j^{(t)} = [I_m \quad 0] W_j, \quad (5)$$

and

$$V_j^{(t)} = [I_m \quad 0] V_j; \quad W_j^{(b)} = [0 \quad I_m] W_j. \quad (6)$$

Similarly, if  $X_j$  and  $G_j$  are the  $j$ th partitions of  $X$  and  $G$ , it comes

$$X_j = \begin{bmatrix} X_j^{(t)} \\ X_j' \\ X_j^{(b)} \end{bmatrix} \quad \text{and} \quad G_j = \begin{bmatrix} G_j^{(t)} \\ G_j' \\ G_j^{(b)} \end{bmatrix}. \quad (7)$$

It is then possible to extract from a block tridiagonal reduced linear system (8) of size  $2(p-1)m$ , which involves only the top and bottom elements of  $V_j$ ,  $W_j$ ,  $X_j$ , and  $G_j$ . As example, the reduced system obtained for the case of four partitions ( $p=4$ ) is given by

$$\begin{bmatrix} I_m & V_1^{(b)} \\ W_2^{(t)} & I_m & V_2^{(t)} \\ W_2^{(b)} & I_m & V_2^{(b)} \\ & W_3^{(t)} & I_m & V_3^{(t)} \\ & W_3^{(b)} & I_m & V_3^{(b)} \\ & & W_4^{(t)} & I_m \end{bmatrix} \begin{bmatrix} X_1^{(b)} \\ X_2^{(t)} \\ X_2^{(b)} \\ X_3^{(t)} \\ X_3^{(b)} \\ X_4^{(t)} \end{bmatrix} = \begin{bmatrix} G_1^{(b)} \\ G_2^{(t)} \\ G_2^{(b)} \\ G_3^{(t)} \\ G_3^{(b)} \\ G_4^{(t)} \end{bmatrix}. \quad (8)$$

Finally, once the solution of the reduced system is obtained, the global solution  $X$  can be reconstructed from  $X_k^{(b)}$  ( $k=1, \dots, p-1$ ) and  $X_k^{(t)}$  ( $k=2, \dots, p$ ) either by computing

$$\begin{cases} X'_1 = G'_1 - V'_1 X_2^{(t)}, \\ X'_j = G'_j - V'_j X_{j+1}^{(t)} - W'_j X_{j-1}^{(b)}, \quad j = 2, \dots, p-1 \\ X'_p = G'_p - W'_p X_{p-1}^{(b)}, \end{cases} \quad (9)$$

or by solving

$$\begin{cases} A_1 X_1 = F_1 - \begin{bmatrix} 0 \\ I_m \end{bmatrix} B_j X_2^{(t)}, \\ A_j X_j = F_j - \begin{bmatrix} 0 \\ I_m \end{bmatrix} B_j X_{j+1}^{(t)} - \begin{bmatrix} I_m \\ 0 \end{bmatrix} C_j X_{j-1}^{(b)}, \quad j = 2, \dots, p-1 \\ A_p X_p = F_p - \begin{bmatrix} I_m \\ 0 \end{bmatrix} C_j X_{p-1}^{(b)}. \end{cases} \quad (10)$$

## SPIKE: A Hybrid and Polyalgorithm

Multiple options are available for efficient parallel implementation of the SPIKE algorithm depending on

the properties of the linear system as well as the architecture of the parallel platform. More specifically, the following stages of the SPIKE algorithm can be handled in several ways resulting in a polyalgorithm:

1. Factorization of the diagonal blocks  $A_j$ . Depending on the sparsity pattern of the matrix and the size of the bandwidth, these diagonal blocks could be considered either as dense or sparse within the band. For the dense banded case, a number of strategies based on the  $LU$  decomposition of each  $A_i$  can be applied here. This include variants such as  $LU$  with pivoting,  $LU$  without any pivoting but diagonal boosting, as well as a combination of  $LU$  and  $UL$  decompositions, either with or without pivoting. For the sparse banded case, it is common to use a sparse direct linear system solver to reorder and then factorize the diagonal blocks. However, solving the various linear systems for  $A_j$  can also be achieved using an iterative solver with preconditioner. Finally, each partition in the decomposition can be associated with one or several processors (one node), enabling multilevel parallelism.
2. Computation of the spikes. If the spikes  $V_j$  and  $W_j$  are determined entirely, the reduced system (8) can be solved explicitly and equation (9) can be used to retrieve the entire solution. In contrast, if equation (10) is used to retrieve the solution, the spikes may not be computed but only for the top and bottom ( $m \times m$ ) blocks of  $V_j$  and  $W_j$  needed to form the reduced system. It should be noted that the determination of the top and bottom spikes is also not explicitly needed for computing the actions of the multiplications with  $W_j^{(t)}$ ,  $W_j^{(b)}$ ,  $V_j^{(t)}$ , and  $V_j^{(b)}$ . These latter can be realized “on-the-fly” using  $\begin{pmatrix} I_m & 0 \end{pmatrix} A_j^{-1} \begin{pmatrix} I_m \\ 0 \end{pmatrix} C_j$ ,  $\begin{pmatrix} 0 & I_m \end{pmatrix} A_j^{-1} \begin{pmatrix} I_m \\ 0 \end{pmatrix} C_j$ ,  $\begin{pmatrix} I_m & 0 \end{pmatrix} A_j^{-1} \begin{pmatrix} I_m \\ 0 \end{pmatrix} B_j$ ,  $\begin{pmatrix} 0 & I_m \end{pmatrix} A_j^{-1} \begin{pmatrix} I_m \\ 0 \end{pmatrix} B_j$ , respectively.
3. Solution scheme for the reduced system. One of the earliest concerns with the SPIKE algorithm for large number of partitions was to propose a reliable and efficient parallel strategy for solving the reduced

system (8). Krylov subspace-based iterative methods have been the first candidates to fulfill this purpose, while giving to SPIKE its hybrid nature. These iterative methods are often used in conjunction with a block Jacobi preconditioner (i.e., diagonal blocks of the reduced system) if the bottom of the  $V_j$  spikes and the top of the  $W_j$  spikes are computed explicitly. In turn, the matrix-vector multiplication operations of the iterative technique can be done explicitly or implicitly (“on-the-fly”). In order to enhance robustness and scalability for solving the reduced system, two new highly efficient direct methods have been introduced by Polizzi and Sameh in [10, 11]. These SPIKE schemes, which have been named “truncated scheme” for handling diagonally dominant systems, and “recursive scheme” for non-diagonally dominant systems, are presented in the next sections. Here again, a number of different strategies exists for solving the reduced system.

As mentioned above, and in order to minimize memory references, it is sometimes advantageous to factorize the diagonal blocks  $A_j$  using LU without any pivoting but adding a diagonal boosting if a “zero-pivot” is detected. Hence,  $A$  is not exactly the product  $DS$  and rather takes the form  $A = DS + R$ , where  $R$  represents the correction which, even if nonzero, is by design small in some sense. Outer iterations via Krylov subspace schemes or iterative refinement, are then necessary to obtain sufficient accuracy as SPIKE would act on  $M = DS$  (i.e., the approximate SPIKE decomposition for  $M$  is used as effective preconditioner).

Finally, a SPIKE-balance scheme has also been proposed by Golub, Sameh, and Sarin in [12], for addressing the case where the block diagonal  $A_j$  are nearly singular (i.e., ill-conditioned), and when even the LU decomposition with partial pivoting is expected to fail.

## The Truncated SPIKE Scheme for Diagonally Dominant Systems

The truncated SPIKE scheme is an optimized version of the SPIKE algorithm with enhanced use of parallelism for handling diagonally dominant systems. These systems may arise from several science and engineering applications, and are defined if the degree of diagonally

dominance,  $dd$ , of the matrix  $A$  is greater than 1, where  $dd$  is given by

$$dd = \min \frac{|A_{i,i}|}{\sum_{j \neq i} |A_{i,j}|}. \quad (11)$$

It is possible to show from equation (1), that the magnitude of the elements of the right spikes  $V_j$  decay from bottom to top, while the elements of the left spikes  $W_j$  decay in magnitude from top to bottom [13, 14]. Since the size  $n$  of  $A_j$  is much larger than the size  $m$  of the blocks  $B_j$  and  $C_j$ , the bottom blocks of the left spikes  $W_j^{(b)}$  and the top blocks of the right spikes  $V_j^{(t)}$  can be approximately set equal to zero. In fact, the zero accuracy machine is ensured to be reached either in the case of a pronounced decay (i.e., high value for  $dd$ ), or for large ratio  $n_j/m$ . Thus, it follows that the off-diagonal blocks of the reduced system (8) are equal to zero, and the solution of this new block-diagonal “truncated” reduced system can be obtained by solving  $p - 1$  independent  $2m \times 2m$  linear systems in parallel:

$$\begin{bmatrix} I_m & V_j^{(b)} \\ W_{j+1}^{(t)} & I_m \end{bmatrix} \begin{bmatrix} X_j^{(b)} \\ X_{j+1}^{(t)} \end{bmatrix} = \begin{bmatrix} G_j^{(b)} \\ G_{j+1}^{(t)} \end{bmatrix}. \quad (12)$$

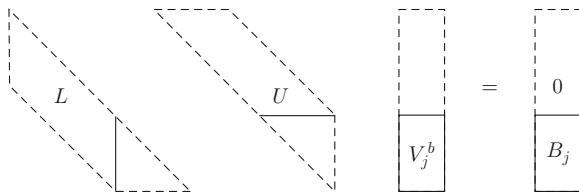
These systems can be solved directly using a block-LU factorization, where the solution steps consist of the following: (a) Form  $E = I_m - W_{j+1}^{(t)} V_j^{(b)}$ , (b) Solve  $EX_{j+1}^{(t)} = G_{j+1}^{(t)} - W_{j+1}^{(t)} G_j^{(b)}$  to obtain  $X_{j+1}^{(t)}$ , (c) Compute  $X_j^{(b)} = G_j^{(b)} - V_j^{(b)} X_{j+1}^{(t)}$ .

Solving the reduced system via the truncated SPIKE algorithm for diagonally dominant systems demonstrates then linear scalability with the number partitions. The truncated scheme is often associated with an outer-iterative refinement step to increase the solution accuracy.

Within the framework of the truncated scheme, two other major contributions have also been proposed for improving computing performance and scalability of the factorization stage: (a) a LU/UL strategy, and (b) a new unconventional partitioning scheme.

## LU/UL Strategy

The truncated scheme facilitates different new options for the factorization step that make possible to avoid the computation of the entire spikes. As illustrated in



**SPIKE.** Fig. 3 The bottom of the spike  $V_j$  can be computed using only the bottom  $m \times m$  blocks of  $L$  and  $U$ . Similarly, the top of the spike  $W_j$  may be obtained if one performs the  $UL$ -factorization

Fig. 3, computational solve times can be drastically reduced by using the  $LU$  factorization without pivoting on each diagonal block  $A_j$ . Obtaining the top block of  $W_j$ , however, would still require computing the entire spike with complete forward and backward sweeps. Another approach consists of performing also the  $UL$ -factorization of the block  $A_j$  without pivoting. Similar to the  $LU$ -factorization, this allows obtaining the top block of  $W_j$  involving only the top  $m \times m$  blocks of the new  $\dot{U}$  and  $\dot{L}$  matrices. Numerical experiments indicate that the time consumed by this  $LU/UL$  strategy is much less than that taken by performing only one  $LU$  factorization per diagonal block and generating the entire left spikes.

### Unconventional Partitioning Schemes

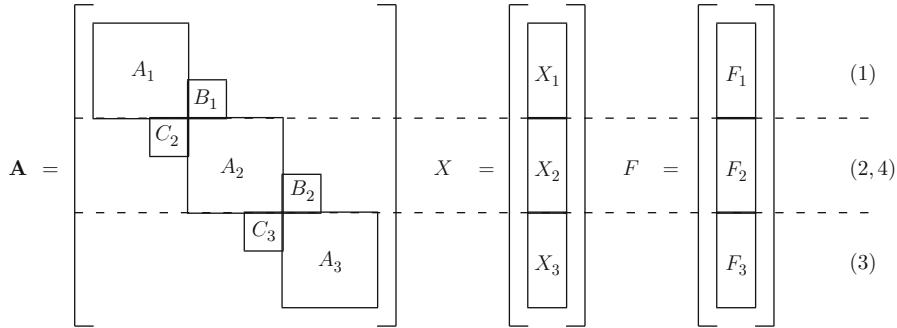
A new partitioning scheme can be introduced to avoid performing both  $LU$  and  $UL$  factorization for a given  $A_j$  ( $j = 2, \dots, p - 1$ ). This scheme acts on a new parallel distribution of the system matrix, which considers less partition than processors. Practically, if  $k$  represents an even number of processors (or nodes in the case of multilevel parallelism), the new number of partitions will be equal to  $p = (k + 2)/2$ . The new block matrices  $A_j$ ,  $j = 1, \dots, p$  can be associated to the first  $p$  processors while processors  $p + 1$  to  $k$  hold another copy of the block matrix  $A_j$ ,  $j = 2, \dots, p - 1$ . Figure 4 illustrates the new partitioning of the matrix right-hand side and solution, for the case  $k = 4$  and  $p = 3$ .

The diagonal blocks  $A_j$  associated to processors 1 to  $p - 1$  are then factorized using  $LU$  without pivoting, while a  $UL$  factorization is used for the diagonal blocks associated with processors  $p$  to  $k$ . In the example of  $k = 4$ ,  $L_j U_j \leftarrow A_j$  for  $j = 1, 2$  (processors 1, 2), and  $\dot{U}_j \dot{L}_j \leftarrow$

$A_j$  for  $j = 2, 3$  (processors 4, 3). As described above using the  $LU/UL$  strategy, the  $V_j^{(b)}$  ( $j = 1, \dots, p - 1$ ) can be obtained with minimal computational efforts via the  $LU$  solve step on processors 1 to  $p - 1$ , while the  $W_j^{(t)}$  ( $j = 2, \dots, p$ ) can be obtained in the similar way but now on different processors using the  $UL$  solve step. Using this new partitioning scheme, the size of the partitions does increase but the number of arithmetic operations by partition decreases as well as the size of the truncated reduced system. This scheme achieves better balance between the computational cost of solving the subproblems and the communication overhead. In addition to increasing scalability results for large number of processors, the scheme also addresses the “bottleneck” of the small number of processors case as described below.

### Speed-Up Performances on Small Number of Processors

One of the main focus in the development of parallel algorithms for solving linear systems aims at achieving linear scalability on large number of processors. However, the emergence of multicore computing platforms in these recent years has brought new emphasis for parallel algorithms on achieving net speedup over the corresponding best sequential algorithms on small number of processors/cores. Clearly, parallel algorithms often inherit extensive preprocessing stages with increased memory references or arithmetics, leading to counter-performances on small number of cores. For parallel banded solvers, in particular, four to eight cores may be usually needed to solve linear systems as fast as the best corresponding sequential solver in LAPACK [15]. It is then important to note that the new partitioning scheme for the truncated SPIKE algorithm has also been designed to address this issue while offering a speedup of two from only two cores. While the two-cores (two-partitions) case can take advantage of a single  $LU$  or  $UL$  factorization for  $A_1$  and  $A_2$ , respectively, the efforts to solve the reduced system become minimal (i.e., as compared to a  $LU$  decomposition on the overall system, the number of arithmetic operations is essentially divided by two in the SPIKE factorization and solve stages). When the number of cores increases, and without accounting for the communication costs (which are minimal for the truncated scheme), the



**SPIKE. Fig. 4** Illustration of the unconventional partitioning of the linear system for the truncated SPIKE algorithm in the case of 4 processors.  $A_1$  is sent to processor 1,  $A_2$  to processors 2 and 4, and  $A_3$  to processor 3.

speedup are expected ideally equal to the number of partitions:  $\times 2$  on two cores,  $\times 3$  on four cores,  $\times 5$  on eight cores, etc. Thereafter, the SPIKE performance will approach linear scalability as the number of cores increases.

### The Recursive SPIKE Scheme for Non-Diagonally Dominant Systems

For non-diagonally dominant systems and large number of partitions, solving the reduced system (8) using Krylov subspace iterative method with or without preconditioner may result in high interprocessor communication cost. Interestingly, the truncated scheme for the two-cores (two-partitions) case is as well applicable for non-diagonally dominant systems since the reduced system (12) contains only one diagonal block. It should be noted, however, that this scheme may necessitate outer-refinement steps, since the diagonal boosting used to handle the “zero-pivot” in the  $LU$  and  $UL$  factorization stages, are more likely to appear for non-diagonally dominant systems.

For larger number of partitions, a new direct approach named “recursive” scheme has been proposed for solving the reduced system and enhancing robustness, accuracy, and scalability. This recursive scheme consists of successive iterations of the SPIKE algorithm from systems to reduced systems, resulting in better balance between the costs of computation and communication. This scheme assumes that the original number of (conventional) partitions is given by  $p = 2^d$  ( $d > 1$ ). The bottom and top blocks of the  $V_j$  and  $W_j$  spikes are then computed explicitly to form the reduced system. In practice, a modified version of the reduced system is

preferred which also includes the top block  $V_1^{(t)}$  and bottom block  $W_p^{(b)}$ . For the case  $p = 4$ , the original reduced system (8) is now represented by the following “reduced spike matrix”:

$$\left( \begin{array}{cc|cc} I_m & V_1^{(t)} & & \\ I_m & V_1^{(b)} & & \\ W_2^{(t)} & I_m & V_2^{(t)} & \\ W_2^{(b)} & I_m & V_2^{(b)} & \\ \hline & W_3^{(t)} & I_m & V_3^{(t)} \\ & W_3^{(b)} & I_m & V_3^{(b)} \\ & W_4^{(t)} & I_m & \\ & W_4^{(b)} & I_m & \end{array} \right) \left[ \begin{array}{c} X_1^{(t)} \\ X_1^{(b)} \\ X_2^{(t)} \\ X_2^{(b)} \\ \hline X_3^{(t)} \\ X_3^{(b)} \\ X_4^{(t)} \\ X_4^{(b)} \end{array} \right] = \left[ \begin{array}{c} G_1^{(t)} \\ G_1^{(b)} \\ G_2^{(t)} \\ G_2^{(b)} \\ \hline G_3^{(t)} \\ G_3^{(b)} \\ G_4^{(t)} \\ G_4^{(b)} \end{array} \right]. \quad (13)$$

This reduced spike system matrix contains  $p$  partitions with  $p$  diagonal block identities. The system can be easily redistributed in parallel using only  $p/2$  partitions which are factorized by SPIKE recursively up until obtaining two partitions only. It can be shown [10] that the two partitions case leading to a  $2m \times 2m$  linear system presented in (12), constitutes the basic computational kernel of the recursive SPIKE scheme.

### The SPIKE Solver: Current and Future Implementation

Since the publications of the first SPIKE algorithm in the late seventies [3, 4], many variations and new schemes have been implemented. In recent years, a comprehensive MPI-Fortran 90 SPIKE package for distributed memory architecture has been developed by the author.

This implementation includes, in particular, all the different family of SPIKE algorithms: recursive, truncated, and on-the-fly schemes. These SPIKE solvers rely on a hierarchy of computational modules, starting with the data locality-rich BLAS level-3, up to the blocked LAPACK [15] algorithms for handling dense banded systems, or up to the direct sparse solver PARDISO [16] for handling sparse banded systems, with SPIKE being on the outermost level of the hierarchy. The package also includes new primitives for banded matrices that make efficient use of BLAS level-3 routines. Those include banded triangular solvers with multiple right-hand sides, banded matrix-matrix multiplications, and LU, UL factorizations with diagonal boosting strategy.

In addition, the large number of options/decision schemes available for SPIKE created the need for the automatic generation of a sophisticated runtime decision tree “SPIKE-ADAPT” that has been developed by Intel. This adaptive layer indicates the most appropriate version of the SPIKE algorithm capable of achieving the highest performance for solving banded systems that are dense within the band. The relevant linear system parameters in this case are system size, number of nodes/processors to be used, bandwidth of the linear system, and degree of diagonal dominance. SPIKE and SPIKE-ADAPT have been regrouped into one package, named “Intel Adaptive Spike-Based Solver,” which has been released to the public in June 2008 on the Intel whatif web site [17].

The SPIKE package also includes a SPIKE-PARDISO scheme for addressing banded linear systems with large sparse bandwidth while offering a basic distributed version of the current shared memory PARDISO package. The capabilities and domain applicability of the SPIKE-PARDISO scheme have recently been significantly enhanced by Manguoglu, Sameh, and Schenk in [18] to address general sparse systems. In this approach, a weighted reordering strategy is used to extract efficient banded preconditioners that are solved via SPIKE-PARDISO including new specific PARDISO features for computing the relevant bottom and top tips of the spikes.

While the current parallel distributed SPIKE package does offer HPC users a new and valuable tool for solving large-scale problems arising from many areas in science and engineering, the growing size of the number of cores in compute node forestalls distributed programming model (i.e., MPI) for many users. On the

other hand, the scalability of the LAPACK banded algorithms on multicore node or SMP is first and foremost dependent on the threaded capabilities of the underlying BLAS routines. A new implementation of the SPIKE solver recently initiated by the author is concerned with a shared memory programming model (i.e., OpenMP) that can consistently match the LAPACK functions for solving banded systems. This SPIKE Open-MP project is expected to offer high efficient threaded alternatives for solving banded linear systems on current and emerging multicore architectures.

## Related Entries

- ▶ [BLAS \(Basic Linear Algebra Subprograms\)](#)
- ▶ [Collective Communication](#)
- ▶ [Dense Linear System Solvers](#)
- ▶ [Linear Algebra, Numerical](#)
- ▶ [Load Balancing, Distributed Memory](#)
- ▶ [Metrics](#)
- ▶ [PARDISO](#)
- ▶ [Preconditioners for Sparse Iterative Methods](#)
- ▶ [ScaLAPACK](#)

## Bibliography Notes and Further Reading

As mentioned in the introduction, the main ideas of the SPIKE algorithm has been introduced in the late 1970s [3, 4], since then, many improvements and variations have been proposed [5–12, 18]. In particular, the highly efficient truncated and recursive schemes for solving the reduced system presented here are discussed in more detail in [10, 11]. All the main SPIKE algorithm variations have been regrouped into a comprehensive MPI-based SPIKE solver package in [17], where the associated SPIKE’s user guide contains more detailed information on the capabilities of the different SPIKE schemes and their domain of applicability.

## Bibliography

1. Cleary A, Dongarra J (1997) Implementation in ScaLAPACK of divide and conquer algorithms for banded and tridiagonal linear systems. University of Tennessee Computer Science Technical Report, UT-CS-97-358
2. Blackford LS, Choi J, Cleary A, Dazevedo E, Demmel J, Dhillon I, et al (1997) ScaLAPACK users guide. Society for Industrial and Appl. Math, Philadelphia
3. Sameh A (1977) Numerical parallel algorithms: a survey. In: Kuck D, Lawrie D, Sameh A (eds) High speed computer and algorithm organization. Academic, New York, pp 207–228

4. Sameh A, Kuck D (1978) On stable parallel linear system solvers. *J ACM* 25:81–91
5. Sameh A (1983) On two numerical algorithms for multiprocessors. In: Proceedings of NATO adv res workshop on high-speed comp. Series F: computer and systems sciences, vol 7. Springer, Berlin, pp 311–328
6. Lawrie D, Sameh A (1984) The computation and communication complexity of a parallel banded system solver. *ACM Trans Math Software* 10(2):185–195
7. Dongarra J, Sameh A (1984) On some parallel banded system solvers. *Parallel Comput* 1:223–235
8. Berry M, Sameh A (1988) Multiprocessor schemes for solving block tridiagonal linear systems. *Int J Supercomput Appl* 2(3): 37–57
9. Sameh A, Sarin V (1999) Hybrid parallel linear solvers. *Int J Comput Fluid Dyn* 12:213–223
10. Polizzi E, Sameh A (2006) A parallel hybrid banded system solver: the SPIKE algorithm. *Parallel Comput* 32(2):177–194
11. Polizzi E, Sameh A (2007) SPIKE: A parallel environment for solving banded linear systems. *Comput Fluids* 36:113–120
12. Golub G, Sameh V, Sarin V (2001) A parallel balance scheme for banded linear systems. *Numer Linear Algebr Appl* 8(5):297–316
13. Demko S, Moss WF, Smith PW (1984) Decay rates for inverses of band matrices. *Math Comput* 43(168):491–499
14. Mikkelsen CCK, Manguoglu M (2008) Analysis of the truncated spike algorithm. *SIAM J Matrix Anal Appl* 30(4):1500–1519
15. Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, et al (1999) LAPACK users guide, 3rd edn. Society for Industrial and Appl. Math, Philadelphia
16. Schenk O, Grtner K (2004) Solving unsymmetric sparse systems of linear equations with PARDISO. *J Future Gener Comput Syst* 20(3):475–487
17. A distributed memory version of the SPIKE package can be obtained from <http://software.intel.com/en-us/articles/intel-adaptive-spike-based-solver/>
18. Manguoglu M, Sameh A, Schenk O (2009) PSPIKE: a parallel hybrid sparse linear system solver. Lecture notes in computer science, vol 5704. Springer, Berlin, pp 797–808

## Spiral

MARKUS PÜSCHEL<sup>1</sup>, FRANZ FRANCHETTI<sup>2</sup>,  
 YEVGEN VORONENKO<sup>2</sup>  
<sup>1</sup>ETH Zurich, Zurich, Switzerland  
<sup>2</sup>Carnegie Mellon University, Pittsburgh, PA, USA

### Definition

Spiral is a program generation system (software that generates other softwares) for linear transforms and an increasing list of other mathematical functions. The goal of Spiral is to automate the development and porting of performance libraries. Linear transforms include the

discrete Fourier transform (DFT), discrete cosine transforms, convolution, and the discrete wavelet transform. The input to Spiral consists of a high-level mathematical algorithm specification and selected architectural and microarchitectural parameters. The output is performance-optimized code in a high-level language such as C, possibly augmented with vector intrinsics and threading instructions.

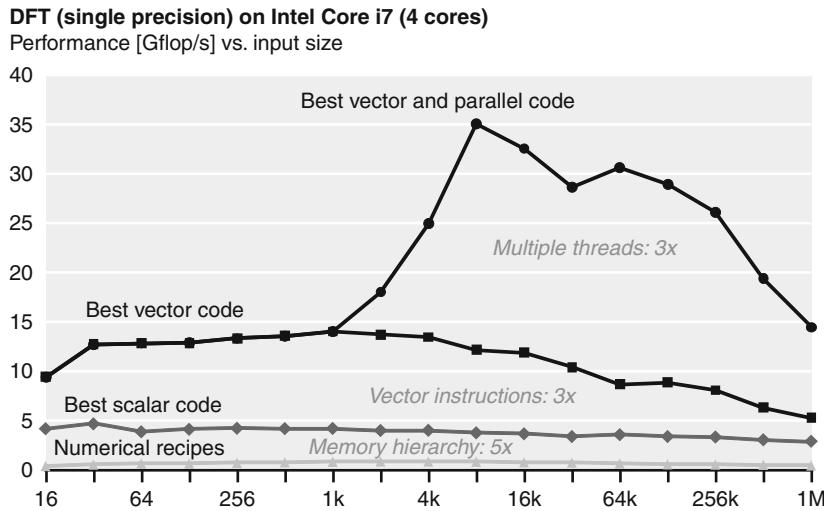
## Discussion

### Introduction

The advent of computers with multiple cores, SIMD (single-instruction multiple-data) vector instruction sets, and deep memory hierarchies has a dramatic effect on the development of high-performance software. The problem is particularly apparent for functions that perform mathematical computations, which form the core of most data or information processing applications. Namely, on a current workstation, the performance difference between a straightforward implementation of an optimal (minimizing operations count) algorithm and the fastest possible implementation is typically 10–100 times.

As an example, consider Fig. 1, which shows the performance (in gigafloating point operations per second) of four implementations of the discrete Fourier transform for varying input sizes on a quadcore Intel Core i7. Each one uses a fast algorithm with roughly the same operations count. Yet the difference between the slowest and the fastest is 12–35 times. The bottom line is the code from Numerical Recipes [20]. The best standard C code is about five times faster due to memory hierarchy optimizations and constant precomputation. Proper use of explicit vector intrinsics yields another three times. Explicit threading for the four cores, properly done, yields another three times for large sizes.

The plot shows that the compiler cannot perform these optimizations as is true for most mathematical functions. The reason lies in both the compiler's lack of domain knowledge needed for the necessary transformations and the large set of optimization choices with uncertain outcome that the compiler cannot assess. Hence the optimization task falls with the programmer and requires considerable skill. Further, the optimizations are usually platform specific, and hence have to be repeated with every new generation of computers.



**Spiral. Fig. 1** Performance of different implementations of the discrete Fourier transform (DFT) and reason for the performance difference (From [10])

Spiral overcomes these problems by completely automating the implementation and optimization processes for the functions it supports. Complete automation means that Spiral produces source code for a given function given only a very high-level representation of the algorithms for this function and a high-level platform description. After algorithm and platform knowledge are inserted, Spiral can generate various types of code including for fixed and general input size, threaded or vectorized.

The approach taken by Spiral is based on the following key principles:

- Algorithm knowledge for a given mathematical function is represented in the form of *breakdown rules* in a *domain-specific language*. Each rule represents a divide-and-conquer algorithm. The language is based on mathematics, and is declarative and platform independent. These properties enable the mapping to various forms of parallelism from algorithm knowledge that is inserted only once. They also enable the derivation of the library structure for general input size implementations by computing the so-called *recursion step closure*.
- Platform knowledge is organized into *paradigms*. A paradigm is a feature of a platform that requires structural optimization and possibly source code extensions. Examples include shared memory parallelism and SIMD vector processing. Each paradigm

consists of a set of *parameterized rewrite rules* and *base cases* expressed in the same language as the algorithm knowledge. The base cases constitute a subset of the domain-specific language that maps well to a paradigm. The rewrite rules interact with the breakdown rules to produce algorithms that are base cases, which means they are structurally optimized for the considered paradigm. Examples of parameters include the SIMD vector length or the cacheline size. Paradigms are designed to be composable.

- Spiral uses *empirical search* to automatically explore choices in a feedback loop. This is done by generating candidate implementations and evaluating their performance. Even though theoretically unsatisfying, search enables further optimization for intricate microarchitectural details that may be unknown or are not well understood.

In summary, Spiral integrates techniques from mathematics, programming languages, compilers, automatic performance tuning, and symbolic computation. The entire Spiral system combines aspects of a compiler, generative programming, and an expert system.

The remainder of this section describes the framework underlying Spiral and the inner workings of the actual system. The presentation focuses on linear transforms; extensions of Spiral beyond transforms are briefly discussed in the end.

## Algorithm Representation

*Linear transforms.* A linear transform is a function

$$x \mapsto Mx,$$

where  $M$  is a fixed matrix,  $x$  is the input vector, and  $y = Mx$  the output vector. Different transforms correspond to different matrices  $M$ . For simplicity,  $M$  is referred to as transform in the following. Most transforms  $M$  are square  $n \times n$ , which implies that  $x$  and  $y$  are of length  $n$ . Most transforms exist for all  $n = 1, 2, \dots$ .

Possibly the most well-known transform is the DFT, defined by the  $n \times n$  matrix:

$$\text{DFT}_n = [\omega_n^{k\ell}]_{0 \leq k, \ell < n}, \quad \omega_n = e^{-2\pi i/n}, \quad i = \sqrt{-1}.$$

Other examples include the discrete Hartley transform,

$$\text{DHT}_n = [\cos(2\pi k\ell/n) + \sin(2\pi k\ell/n)]_{0 \leq k, \ell < n},$$

the discrete cosine transform (DCT) of type 2,

$$\text{DCT-2}_n = [\cos(k(\ell + \frac{1}{2})\pi/n)]_{0 \leq k, \ell < n},$$

as well as other types of discrete cosine and sine transforms, the Walsh–Hadamard transform, the real DFT, the discrete wavelet transform, the inverses and other variants of the preceding transforms, and finite impulse response filters.

*Fast transform algorithms:* SPL. If  $M$  is  $n \times n$  and has few or no zero entries, then a direct computation of  $y = Mx$  requires  $O(n^2)$  many operations. However, all the transforms mentioned above have fast algorithms that reduce their complexity below that, typically to  $O(n \log(n))$ . Every algorithm can be expressed as a factorization of the transform matrix  $M$  into a product of sparse matrices. As an example, assume  $M = M_1 M_2 M_3 M_4$ ; then  $y = Mx$  can be computed in four steps as

$$t = M_4 x, u = M_3 t, v = M_2 u, y = M_1 v.$$

If the  $M_i$  are sufficiently sparse, this reduces the operations count.

The sparse matrices occurring in transform algorithms have a structure that can be formally expressed using basic matrices and matrix operators such as the direct sum and the tensor or Kronecker product. This notation forms the basis for the language SPL (signal processing language) explained next.

Basic matrices include the  $n \times n$  identity matrix  $I_n$ , diagonal matrices  $D_n = \text{diag}(a_0, \dots, a_{n-1})$ , the  $2 \times 2$

butterfly matrix

$$F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix},$$

the stride permutation matrix  $L_k^n$ , defined for  $n = km$  by the underlying permutation

$$\ell_k^n : im + j \mapsto jk + i, \quad 0 \leq i < k, \quad 0 \leq j < m, \quad (1)$$

and several others.

Matrix operators include the matrix product, the direct sum

$$A \oplus B = \begin{bmatrix} A \\ B \end{bmatrix},$$

and the tensor product

$$A \otimes B = [a_{k,\ell} B]_{0 \leq k, \ell < n}, \quad \text{for } A = [a_{k,\ell}]_{0 \leq k, \ell < n}.$$

Most important are the tensor products where  $A$  or  $B$  is the identity:

$$I_n \otimes B = \begin{bmatrix} B \\ & \ddots \\ & & B \end{bmatrix},$$

and, for example,

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \otimes I_3 = \begin{bmatrix} aI_3 & bI_3 \\ cI_3 & dI_3 \end{bmatrix} = \begin{bmatrix} a & b & & \\ & a & b & \\ & & a & b \\ c & d & & \\ & c & d & \\ & & c & d \end{bmatrix}.$$

A (partial) description of SPL in Backus–Naur form is provided in [Table 1](#).

*Algorithms as SPL breakdown rules.* Using SPL, the algorithm knowledge in Spiral is captured by *breakdown rules*. A breakdown rule represents a one-step divide-and-conquer algorithm of a transform. This means the transform is factorized into sparse matrices involving other, typically smaller, transforms.

**Spiral. Table 1** A subset of SPL in Backus–Naur form;  $n, k$  are positive integers,  $a_i$  are real or complex numbers

|                                                                                                                                     |                  |
|-------------------------------------------------------------------------------------------------------------------------------------|------------------|
| $\langle \text{spl} \rangle ::= \langle \text{generic} \rangle   \langle \text{basic} \rangle   \langle \text{transform} \rangle  $ |                  |
| $\langle \text{spl} \rangle \dots \dots \langle \text{spl} \rangle  $                                                               | (product)        |
| $\langle \text{spl} \rangle \oplus \dots \oplus \langle \text{spl} \rangle  $                                                       | (direct sum)     |
| $\langle \text{spl} \rangle \otimes \dots \otimes \langle \text{spl} \rangle  $                                                     | (tensor product) |
| ...                                                                                                                                 |                  |
| $\langle \text{generic} \rangle ::= \text{diag}(a_0, \dots, a_{n-1})   \dots$                                                       |                  |
| $\langle \text{basic} \rangle ::= I_n   L_k^n   F_2   \dots$                                                                        |                  |
| $\langle \text{transform} \rangle ::= \text{DFT}_n   \text{DHT}_n   \text{DCT-2}_n   \dots$                                         |                  |

The most well-known example is the general-radix Cooley–Tukey fast Fourier transform (FFT):

$$\text{DFT}_n \rightarrow (\text{DFT}_k \otimes I_m) T_m^n (I_k \otimes \text{DFT}_m) L_k^n, \quad n = km, \quad (2)$$

where  $T_m^n$  is the diagonal matrix of *twiddle factors*. For  $n = 16 = 4 \times 4$ , the factorization is visualized in Fig. 2 together with the associated data-flow graph. The smaller DFT<sub>4</sub>'s are boxes of equal shades of gray.

To terminate the recursion, base cases are needed. For example, for two-powers  $n$ , a size two base case is sufficient:

$$\text{DFT}_2 \rightarrow F_2. \quad (3)$$

A few points are worth noting about this representation of transform algorithms:

- The representation (2) is *point free*, i.e., the input vector is not present.
- The representation (2) is declarative.
- Since the rule (2) is a matrix equation, it can be manipulated using matrix identities. For example, both sides can be inverted or transposed, to obtain an inverse or transposed transform algorithm.
- A breakdown rule may have degrees of freedom. An example is the choice of  $k$  in (2).
- A rule like (2) does not specify how to compute the smaller transforms. This implies that rules have to be applied recursively until an algorithm is completely specified. Because of this and the availability of different rules for the same transform, there is a large set of choices. In other words, the relatively few existing rules yield a very large space of possible algorithms. This makes rules a very efficient representation of algorithm knowledge. For example, for  $n = 2^\ell$ , (2) alone yields  $\Theta(5^\ell / \ell^{3/2})$  different algorithms, all with roughly the same operations count.

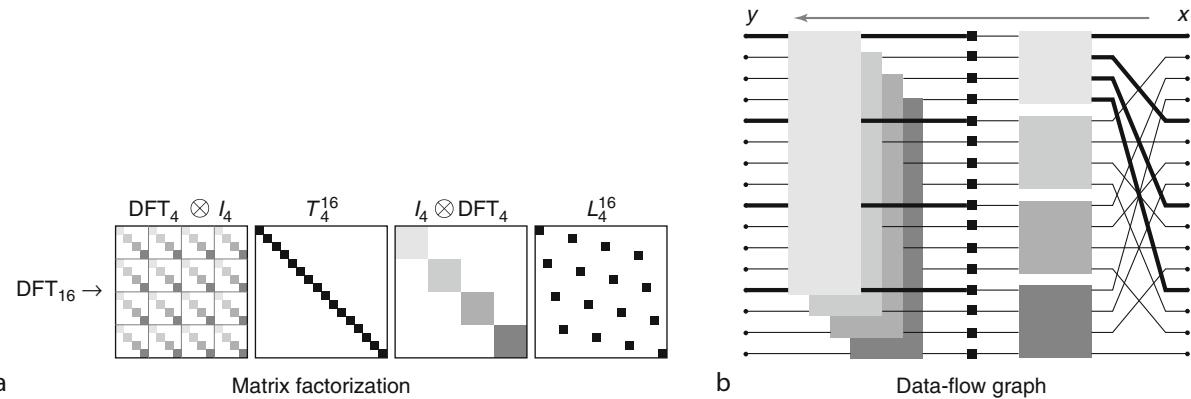
Spiral contains about 200 breakdown rules for about 40 transforms, some of which are auxiliary. The most important rules for the DFT, without complete specification, are shown in Table 2. Note the occurrence of auxiliary transforms.

### Spiral Program Generation: Overview

The task performed by Spiral is to translate the algorithm knowledge (represented as in Table 2) for a given transform into optimized source code (we assume C/C++) for a given platform.

The exact approach for generating the code depends on the type of code that has to be generated. The most important distinctions are the following:

- *Fixed input size versus general input size*: If the input size is known (e.g., “DFT of size 4” as shown in Table 3a and b), the algorithm to be used and other decisions can be determined at program generation time and can be inlined. The result is a function containing only loops and basic blocks of straightline code. If the input size is not known, it becomes an additional input and the implementation becomes recursive (Table 3c). The actual algorithm, i.e., recursive computation, is now chosen at runtime once the input size is known.
- *Straightline code versus loop code (fixed input size only)*: Straightline code (Table 3a) is only suitable for small sizes, but can be faster, due to reduced overhead and increased opportunities for algebraic simplifications. Loop code (Table 3b) requires additional optimizations that merge redundant loops.
- *Scalar code versus parallel code*: Code that is parallelized for SIMD vector extensions or multiple cores requires specific optimizations and the use of explicit vector intrinsics or threading directives.



**Spiral. Fig. 2** Cooley-Tukey FFT (2) for  $16 = 4 \times 4$  as SPL rule and as (complex) data-flow graph (from right to left). Some lines are bold to emphasize the strided access of the  $DFT_4$ s (From [10])

**Spiral. Table 2** A selection of breakdown rules representing algorithm knowledge for the DFT. rDFT is an auxiliary transform and has two parameters. RDFT is a version of the real DFT

|                                                                                                                                       |                                             |
|---------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------|
| $DFT_n \rightarrow (DFT_k \otimes I_m) T_m^n (I_k \otimes DFT_m) L_k^n,$                                                              | (Cooley-Tukey FFT) $n = km$                 |
| $DFT_n \rightarrow V_n^{-1} (DFT_k \otimes I_m) (I_k \otimes DFT_m) V_n,$                                                             | (Prime-factor FFT) $n = km, \gcd(k, m) = 1$ |
| $DFT_n \rightarrow W_n^{-1} (I_h \oplus DFT_{p-1}) E_n (I_l \oplus DFT_{p-1}) W_n,$                                                   | (Rader FFT) $n$ prime                       |
| $DFT_n \rightarrow B'_n D_m DFT_m D'_m DFT_m D''_m B_n,$                                                                              | (Bluestein FFT) $n > 2m$                    |
| $DFT_n \rightarrow P_{k,2m}^\top (DFT_{2m} \oplus (I_{k-1} \otimes_i C_{2m} rDFT_{2m,i/2k})) (RDFT_{2k} \otimes I_m),$                | $n = 2km$                                   |
| $RDFT_n \rightarrow (P_{k,m}^\top \otimes I_2) (RDFT_{2m} \oplus (I_{k-1} \otimes_i D_{2m} rDFT_{2m,i/2k})) (RDFT_{2k} \otimes I_m),$ | $n = 2km$                                   |
| $rDFT_{n,u} \rightarrow L_m^{2n} (I_k \otimes_i rDFT_{2m,(i+u)/k}) (rDFT_{2k,u} \otimes I_m),$                                        | $n = 2km$                                   |

**Spiral. Table 3** Code types

| (a) Fixed input size, unrolled                                                                                                                                                                                                                        | (b) Fixed input size, looped                                                                                                                                                                                                                                                                                                      | (c) General input size library, recursive                                                                                                                                                                                                                                               |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>void dft_4(cpx *Y, cpx *X){     cpx s, t, t2, t3;     t = (X[0] + X[2]);     t2 = (X[0] - X[2]);     t3 = (X[1] + X[3]);     s = _I_* (X[1] - X[3]);     Y[0] = (t + t3);     Y[2] = (t - t3);     Y[1] = (t2 + s);     Y[3] = (t2 - s); }</pre> | <pre>void dft_4(cpx *Y, cpx *X){     cpx T[4];     cpx W[2] = {1, _I_};     for(int i = 0; i &lt;= 1; i++) {         cpx w = W[i];         T[2*i] = (X[i] + X[i+2]);         T[2*i+1] = w*(X[i] - X[i+2]);     }     for(int j = 0; j &lt;= 1; j++) {         Y[j] = T[j] + T[j+2];         Y[2+j] = T[j] - T[j+2];     } }</pre> | <pre>struct dft : public Env{     dft(int n); // constructor     void compute(cpx *Y, cpx *X);     int _rule, f, n;     char *_dat;     Env *ch1, *ch2; };  void dft::compute(cpx *Y, cpx *X) {     ch2-&gt;compute(Y, X, n, f, n, f);     ch1-&gt;compute(Y, Y, n, f, n, n/f); }</pre> |

The program generation process is explained in the next four sections corresponding to four different code types of increasing difficulty. The order matches the historic development, since for each move to the next code type

at least one new idea had to be introduced. The types and main ideas (in parentheses) are

- Fixed input size straightline code (SPL, breakdown rules, feedback loop)

- Fixed input size loop code ( $\Sigma$ -SPL, loop merging)
- Fixed input size parallel code (paradigms, tagged rewriting)
- General input size code (recursion step closure, parameterization)

Spiral generates code for fixed input size transforms (first three bullets), as shown in Fig. 3. The input is the transform symbol (e.g., “DFT”) and the size (e.g., “128”). The output is a C function that computes the transform ( $y = \text{DFT}_{128} x$  in this case). Depending on code type, not all blocks in Fig. 3 may be used.

The block diagram for the general input size code is shown later.

### Fixed Input Size: Straightline Code

Given as input to Spiral is a transform symbol (“DFT”) and the input size. The program generation does not need the parallelization and loop optimization blocks. Further, no  $\Sigma$ -SPL is needed, which means the SPL-to- $\Sigma$ -SPL block and the  $\Sigma$ -SPL-to-code block are joined to one SPL-to-code block.

*Algorithm generation.* Spiral uses a rewrite system that recursively applies the breakdown rules (e.g., Table 2) to generate a complete SPL algorithm for the transform. As mentioned before, there are many choices

due to the choice of rule and the degree of freedom in some rules (e.g.,  $k$  in (2)).

*SPL to C code and optimization.* The SPL expression is then compiled into actual C code using the internal SPL compiler, which recursively applies the translation rules sketched in Table 4.

All loops are unrolled and code-level optimizations are applied. These include array scalarization, constant propagation, and algebraic simplification.

*Performance evaluation.* The runtime of the resulting code is measured and fed into the search block that controls the algorithm generation.

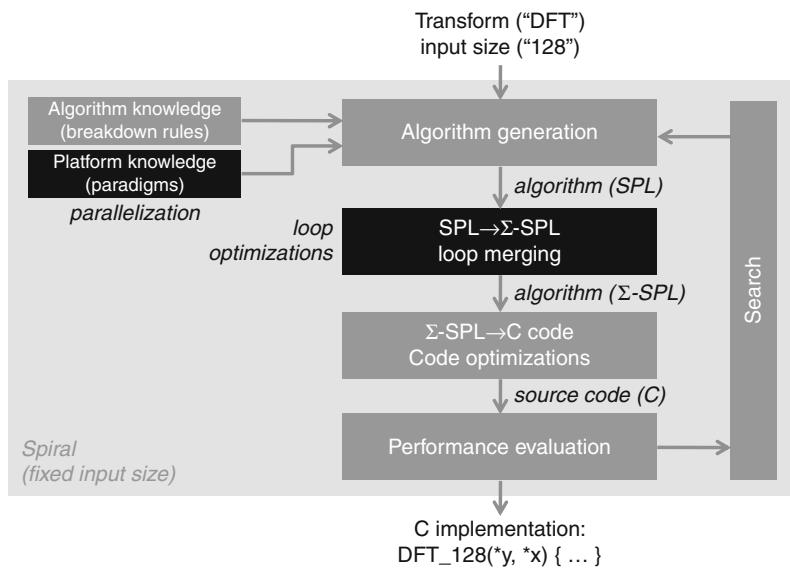
*Search.* The search drives a feedback loop that generates and evaluates different algorithms to find the fastest. Dynamic programming has proven to work best in many cases, but other techniques including evolutionary search or bandit-based Monte Carlo exploration have also been studied.

### Fixed Input Size: Loop Code

The approach to generating straightline code can also be used to generate loop code (Table 4 yields loops), but the code will be inefficient.

*The problem: Loop merging.* To illustrate the problem, consider the SPL expression

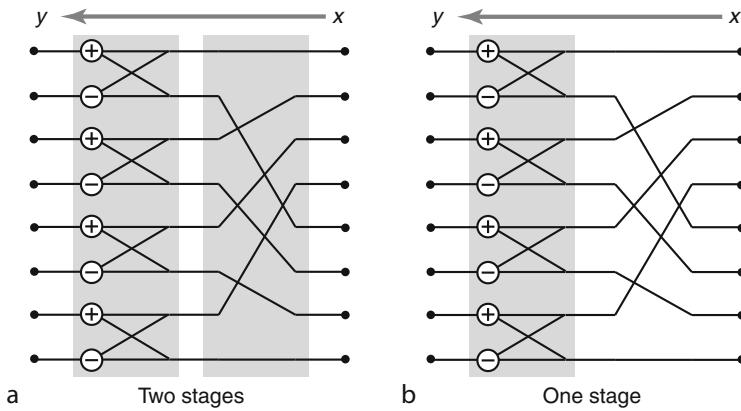
$$(I_4 \otimes F_2)L_4^8.$$



**Spiral. Fig. 3** Spiral program generator for fixed input size functions. For straightline code, no  $\Sigma$ -SPL is needed and SPL is translated directly into C code

**Spiral. Table 4** Translation of SPL to code. The subscript of  $A, B$  specifies the (square) matrix size.  $x [b : s : e]$  denotes (Matlab style) the subvector of  $x$  starting at  $b$ , ending at  $e$ , and extracted at stride  $s$ .  $D$  is a diagonal matrix, whose diagonal elements are stored in an array with the same name

| SPL expression $S$ | Pseudo code for $y = Sx$                                                                     |
|--------------------|----------------------------------------------------------------------------------------------|
| $A_n B_n$          | <code for: $t = Bx$ ><br><code for: $y = At$ >                                               |
| $I_m \otimes A_n$  | for ( $i=0$ ; $i < m$ ; $i++$ )<br><code for: $y[i*n:1:i*n+n-1] = A(x[i*n:1:i*n+n-1])$ >     |
| $A_m \otimes I_n$  | for ( $i=0$ ; $i < n$ ; $i++$ )<br><code for: $y[i:n:i+m*n-n] = A(x[i:n:i+m*n-n])$ >         |
| $D_n$              | for ( $i=0$ ; $i < n$ ; $i++$ )<br>$y[i] = D[i]*x[i];$                                       |
| $L_k^{km}$         | for ( $i=0$ ; $i < k$ ; $i++$ )<br>for ( $j=0$ ; $j < m$ ; $j++$ )<br>$y[i*m+j] = x[j*k+i];$ |
| $F_2$              | $y[0] = x[0] + x[1];$<br>$y[1] = x[0] - x[1];$                                               |



**Spiral. Fig. 4** The loop merging problem for  $(I_4 \otimes F_2)L_4^8$

The application of Table 4 yields the code visualized in Fig. 4a:

```
// Input: double x[8], output: y[8]
double t[8];
for(int i=0; i<4; i++) {
 for (int j=0; j<2; j++) {
 t[i*2+j] = x[j*4+i];
 }
}
for (int j=0; j<4; j++) {
 y[2*j] = t[2*j] + t[2*j+1];
 y[2*j+1] = t[2*j] - t[2*j+1];
}
```

This is known to be suboptimal since the permutation (first loop) can be fused with the subsequent computation loop, thus eliminating one pass through the data (Fig. 4b):

```
// Input: double x[8], output: y[8]
for (int j=0; j<4; j++) {
 y[2*j] = x[j] + x[j+4];
 y[2*j+1] = x[j] - x[j+4];
}
```

This transformation cannot be expressed in SPL and, in the general case, is difficult to perform on C code. To solve this problem,  $\Sigma$ -SPL was developed, an extension

of SPL that can express loops. The loop merging is then performed by rewriting  $\Sigma$ -SPL expressions.

$\Sigma$ -SPL.  $\Sigma$ -SPL adds four basic components to SPL:

1. Index mapping functions
2. Scalar functions
3. Parameterized matrices
4. Iterative sum  $\Sigma$

These are defined next.

An integer interval is denoted by  $\mathbb{I}_n = \{0, \dots, n-1\}$ , and an index mapping function  $f$  with domain  $\mathbb{I}_n$  and range  $\mathbb{I}_N$  is denoted by

$$f^{n \rightarrow N} : \mathbb{I}_n \rightarrow \mathbb{I}_N; i \mapsto f(i).$$

An example is the stride function

$$h_{b,s}^{n \rightarrow N} : \mathbb{I}_n \rightarrow \mathbb{I}_N; i \mapsto b + is, \quad \text{for } s|N. \quad (4)$$

Permutations are written as  $f^{n \rightarrow n} = f^n$ , such as the stride permutation in (1).

A scalar function  $f : \mathbb{I}_n \rightarrow \mathbb{C}; i \mapsto f(i)$  maps an integer interval to the domain of complex or real numbers, and is abbreviated as  $f^{n \rightarrow \mathbb{C}}$ . Scalar functions are used to describe diagonal matrices.

$\Sigma$ -SPL adds four types of parameterized matrices to SPL (gather, scatter, permutation, diagonal):

$$G(f^{n \rightarrow N}), S(f^{n \rightarrow N}), P(f^n), \text{ and } \text{diag}(f^{n \rightarrow \mathbb{C}}).$$

Their translation into actual code (which also defines the matrices) is shown in Table 5. For example,

$$G(h_{0,1}^{n \rightarrow N}) = \begin{bmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{bmatrix}, \quad S(h_{0,1}^{n \rightarrow N}) = G(h_{0,1})^\top.$$

Spiral. Table 5 Translation of  $\Sigma$ -SPL to code

| $\Sigma$ -SPL expression $S$                | Code for $y = Sx$                                                  |
|---------------------------------------------|--------------------------------------------------------------------|
| $G(f^{n \rightarrow N})$                    | <pre>for(i=0; i&lt;n; i++)     y[i] = x[f(i)];</pre>               |
| $S(f^{n \rightarrow N})$                    | <pre>for(i=0; i&lt;n; i++)     y[f(i)] = x[i];</pre>               |
| $P(f^n)$                                    | <pre>for(i=0; i&lt;n; i++)     y[i] = x[f(i)];</pre>               |
| $\text{diag}(f^{n \rightarrow \mathbb{C}})$ | <pre>for(i=0; i&lt;n; i++)     y[i] = f(i)*x[i];</pre>             |
| $\sum_{i=0}^{k-1} A_i$                      | <pre>for(i=0; i&lt;k; i++)     &lt;code for: y = A_i * x&gt;</pre> |

Finally,  $\Sigma$ -SPL adds the iterative matrix sum

$$\sum_{i=0}^{n-1} A_i$$

to represent loops. The  $A_i$  are restricted such that no two  $A_i$  have a nonzero entry in the same row.

The following example shows how  $\otimes$  is converted into a sum.  $A$  is assumed to be  $n \times n$ , and domain and range in the occurring stride functions are omitted for simplicity.

$$\begin{aligned} I_k \otimes A &= \begin{bmatrix} A & & \\ & \ddots & \\ & & A \end{bmatrix} = \begin{bmatrix} A & & & \\ & \ddots & & \\ & & \ddots & \\ & & & A \end{bmatrix} + \cdots + \begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & A \end{bmatrix} \\ &= S(h_{0,1})AG(h_{0,1}) + \dots \\ &\quad + S(h_{(k-1)n,1})AG(h_{(k-1)n,1}) \\ &= \sum_{i=0}^{k-1} S(h_{in,1})AG(h_{in,1}) \end{aligned}$$

Intuitively, the conversion to  $\Sigma$ -SPL makes the loop structure of  $y = (I_k \otimes A)x$  explicit. In each iteration  $i$ ,  $G(\cdot)$  and  $S(\cdot)$  specify how to read and write a portion of the input and output, respectively, to be processed by  $A$ .

*Loop merging using  $\Sigma$ -SPL and rewriting.* Using  $\Sigma$ -SPL, the loop merging problem identified before in the example  $(I_4 \otimes F_2)L_4^8$  is solved by the loop optimization block in Fig. 3 as follows:

$$\begin{aligned} (I_4 \otimes F_2)L_4^8 &\rightarrow \left( \sum_{i=0}^3 S(h_{2i,1})F_2G(h_{2i,1}) \right) P(\ell_4^8) \\ &\rightarrow \sum_{i=0}^3 \left( S(h_{2i,1})F_2G(\ell_4^8 \circ h_{2i,1}) \right) \\ &\rightarrow \sum_{i=0}^3 \left( S(h_{2i,1})F_2G(h_{i,2}) \right) \end{aligned}$$

The first step translates SPL into  $\Sigma$ -SPL. The second step performs the loop merging by composing the permutation  $\ell_4^8$  with the index functions of the subsequent gathers. The third step simplifies the resulting index functions. After that, actual C code is generated using Table 5.

Besides the added loop optimizations block in Fig. 3, the program generation for loop code operates iteratively exactly as for straightline code.

### Fixed Input Size: Parallel Code

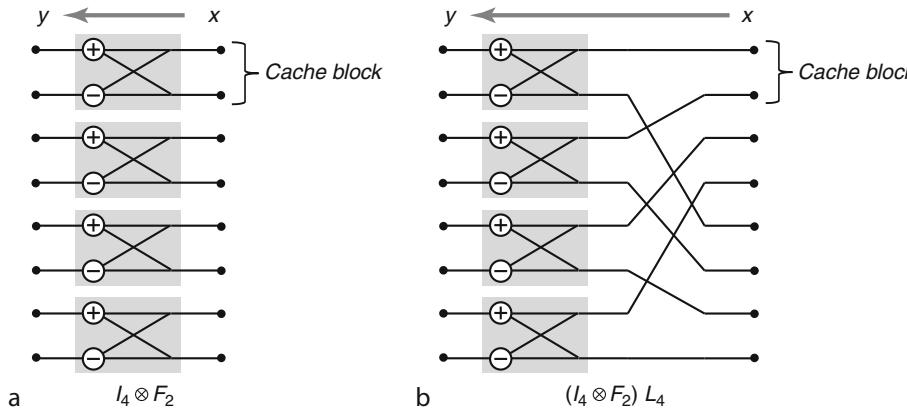
As was illustrated in Fig. 1, for compute functions, compilers usually fail to optimally (or at all) exploit the parallelism offered by a platform. Hence, the task falls

with the programmer, who has to leave the standard C programming model and insert explicit threading or OpenMP loops for shared memory parallelism and so-called intrinsics for vector instruction sets. However, doing so in a straightforward way does not necessarily yield good performance.

*The problem: Algorithm structure.* To illustrate the problem, consider a target platform with four cores that share a cache with a cache block size of two complex numbers.

The first goal is to obtain parallel code with four threads for  $I_4 \otimes F_2$  visualized in Fig. 5a. The computation is data parallel; hence, the loop suggested in Table 4 can be replaced, for example, by an OpenMP parallel loop. Note that each processor “owns” as working set exactly one cache block; hence, the parallelization will be efficient.

Now consider again the SPL expression  $(I_4 \otimes F_2)L_4^8$  visualized in Fig. 5b. The computation is again data parallel, but the access pattern has changed such that always two processors access the same cache block. This produces false sharing, which triggers the cache coherency protocol and reduces performance. The problem is obviously the permutation  $L_4^8$ . Since the rules (e.g., those in Table 2) contain many, and various, permutations, a straightforward mapping to parallel code will yield highly suboptimal performance. To solve this problem inside Spiral, another rewrite system is introduced to restructure algorithms before mapping to parallel code. The restructuring will be different for different forms of parallelism, called paradigms.



**Spiral. Fig. 5** Mapping SPL constructs to four threads. Each thread computes one  $F_2$ . Both computations are data parallel, but (a) produces no false sharing, whereas (b) does

*Paradigms and tagged rewriting.* A paradigm in Spiral is a feature of the target platform that requires structural optimization. Typically, a paradigm is a form of parallelism. Examples include shared memory parallelism (SMP) and SIMD parallelism. A paradigm may be parameterized, for example, by the vector length  $v$  for SIMD parallelism. In Spiral, a paradigm manifests itself by another rewrite system provided by the additional parallelism block in Fig. 3 (and backend extensions in the  $\Sigma$ -SPL to C code block to produce the actual code).

The goal of the new rewrite system is to structurally optimize a given SPL expression into a form that can be efficiently mapped to a given paradigm. The rewrite system is built from three main components:

- *Tags* encode the paradigm and relevant parameters. Examples include the tags “ $\text{vec}(v)$ ” for SIMD vector extensions and the tag “ $\text{smp}(p, \mu)$ ” for SMP. The meaning of the parameters is explained later.
- *Base cases* are SPL constructs that can be mapped well to a given paradigm. As illustrated above, one example is any  $I_p \otimes A_n$  for  $p$ -way SMP.
- *Tagged rewrite rules* are mathematical identities that translate general SPL expressions toward base cases. An example is the rule (assuming  $p|n$ )

$$\underbrace{A_m \otimes I_n}_{\text{smp}(p, \mu)} \rightarrow \underbrace{L_m^{mn}}_{\text{smp}(p, \mu)} \left( I_p \otimes (I_{n/p} \otimes A_m) \right) \underbrace{L_n^{mn}}_{\text{smp}(p, \mu)} .$$

The rule extracts the  $p$ -way parallel loop (base case)  $I_p \otimes (I_{n/p} \otimes A_m)$  from  $A_m \otimes I_n$ . The stride permutations  $L_m^{mn}$  and  $L_n^{mn}$  are handled by further rewriting.

*Example:* SMP. For SMP, the tag  $\text{smp}(p, \mu)$  contains the number of processors  $p$  and the cache block size  $\mu$ . Base cases include  $I_p \otimes A_n$  and  $P \otimes I_\mu$ , where  $P$  is any permutation.  $P \otimes I_\mu$  moves data in blocks of size  $\mu$ ; hence false sharing is avoided. From these, other base cases can be built recursively as captured by the sketched grammar in Table 6.

Some SMP rewrite rules are shown in Table 7. Note that the rewriting is not unique, and not every sequence of rules terminates. Once all tags disappear, the rewriting terminates.

*Example:* SIMD. For SIMD, the tag  $\text{vec}(v)$  contains only the vector length  $v$ . The most important base case is  $A_n \otimes I_v$ , which can be mapped to vector code by generating scalar code for  $A_n$  and replacing every operation by its corresponding  $v$ -way vector operation. Other base cases include  $L_v^{2v}$ ,  $L_2^{2v}$ , and  $L_v^{v^2}$ , which are generated automatically from the instruction set [9]. Similar to Table 6, the entire set of vector base cases is specified

by a grammar recursively built from the above special constructs.

*Parallelization by rewriting.* In Spiral, parallelization adds the new parallelization block in Fig. 3. The parallelization rules are applied interleaved with the breakdown rules to generate SPL algorithms that have the right structure for the desired paradigm. For example, for the DFT it may operate as follows:

$$\begin{aligned} \text{DFT}_{mn} &\rightarrow \underbrace{\left( (\text{DFT}_m \otimes I_n) T_n^{mn} (I_m \otimes \text{DFT}_n) L_m^{mn} \right)}_{\text{smp}(p, \mu)} \\ &\dots \\ &\rightarrow \underbrace{(\text{DFT}_m \otimes I_n)}_{\text{smp}(p, \mu)} \underbrace{T_n^{mn}}_{\text{smp}(p, \mu)} \underbrace{(I_m \otimes \text{DFT}_n)}_{\text{smp}(p, \mu)} \underbrace{L_m^{mn}}_{\text{smp}(p, \mu)} \\ &\dots \\ &\rightarrow ((L_m^{mp} \otimes I_{n/p}) \otimes I_\mu) (I_p \otimes (\text{DFT}_m \otimes I_{n/p})) \\ &\quad ((L_p^{mp} \otimes I_{n/p}) \otimes I_\mu) T_m^{mn} (I_p \otimes (I_{m/p} \otimes \text{DFT}_n)) \\ &\quad (I_p \otimes L_{m/p}^{mn/p}) ((L_p^{pn} \otimes I_{m/p}) \otimes I_\mu) \end{aligned}$$

**Spiral. Table 6**  $\text{smp}(p, \mu)$  base cases in Backus–Naur form;  $n$  is a positive integer,  $a_i$  are real or complex numbers

|                                                                                                        |
|--------------------------------------------------------------------------------------------------------|
| $\langle \text{smp} \rangle ::= \langle \text{generic} \rangle \mid \langle \text{basic} \rangle \mid$ |
| $\langle \text{smp} \rangle \cdots \cdots \langle \text{smp} \rangle \mid \text{ (product)}$           |
| $\langle \text{smp} \rangle \oplus \dots \oplus \langle \text{smp} \rangle \mid \text{ (direct sum)}$  |
| $I_n \otimes \langle \text{smp} \rangle \mid \text{ (tensor product)}$                                 |
| $\dots$                                                                                                |
| $\langle \text{generic} \rangle ::= \text{diag}(a_0, \dots, a_{n-1}) \mid \dots$                       |
| $\langle \text{basic} \rangle ::= I_p \otimes A_n \mid P \otimes I_\mu \mid \dots$                     |

**Spiral. Table 7** Examples of  $\text{smp}(p, \mu)$  rewrite rules

|                                                     |               |                                                                                                                                                                                                                |
|-----------------------------------------------------|---------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\underbrace{AB}_{\text{smp}(p, \mu)}$              | $\rightarrow$ | $\underbrace{A}_{\text{smp}(p, \mu)} \underbrace{B}_{\text{smp}(p, \mu)}$                                                                                                                                      |
| $\underbrace{A_m \otimes I_n}_{\text{smp}(p, \mu)}$ | $\rightarrow$ | $\underbrace{(L_m^{mp} \otimes I_{n/p}) (I_p \otimes (A_m \otimes I_{n/p})) (L_p^{mp} \otimes I_{n/p})}_{\text{smp}(p, \mu)}$                                                                                  |
| $\underbrace{L_m^{mn}}_{\text{smp}(p, \mu)}$        | $\rightarrow$ | $\begin{cases} \underbrace{(I_p \otimes L_{m/p}^{mn/p}) (L_p^{pn} \otimes I_{m/p})}_{\text{smp}(p, \mu)} \\ \underbrace{(L_m^{pm} \otimes I_{n/p}) (I_p \otimes L_m^{mn/p})}_{\text{smp}(p, \mu)} \end{cases}$ |
| $\underbrace{I_m \otimes A_n}_{\text{smp}(p, \mu)}$ | $\rightarrow$ | $I_p \otimes (I_{m/p} \otimes A_n)$                                                                                                                                                                            |
| $\underbrace{(P \otimes I_n)}_{\text{smp}(p, \mu)}$ | $\rightarrow$ | $(P \otimes I_{n/\mu}) \otimes I_\mu$                                                                                                                                                                          |

First, Spiral applies the breakdown rule (2). Then the parallelization rules transform the resulting SPL expression in several steps. Note how the final expression has only access patterns (permutations) of the form  $P \otimes I_\mu$  and all computations are in the form  $I_p \otimes A$  (and the diagonal  $T_m^{mn}$ ). The smaller DFTs can be expanded in different ways, for example, by rewriting for SIMD. Further choices are used for search.

The remaining operation of Spiral including  $\Sigma$ -SPL conversion and search proceeds as before.

## General Input Size

An implementation that can compute a transform for arbitrary input size is fundamentally different from one for fixed input size (compare Table 3b and c). If the input size  $n$  is fixed, for example,  $n = 4$ , the computation is

$$(x, y) \rightarrow \text{dft\_4}(y, x)$$

and all decisions such as the choice of recursion until base cases are reached can be made at implementation time. In an equivalent implementation (called library) for general input size  $n$ ,

$$(n, x, y) \rightarrow \text{dft}(n, y, x)$$

the recursion is fixed only once the input size is known. Formally, the computation now becomes

$$n \rightarrow ((x, y) \rightarrow \text{dft}(n, y, x))$$

which is an example of function currying. A C++ implementation is sketched in [Table 3c](#), where the two steps would take the form

```
dft * f = new dft(n); // initialization
f->compute(y, x); // computation
```

The first step determines the recursion to be taken using search or heuristics, and precomputes the twiddle factors needed for the computation. The second step performs the actual computation. The underlying assumption is that the cost of the first step is amortized by a sufficient number of computations. This model is used by FFTW [15] and the libraries generated by Spiral.

To support the above model, the implementation needs recursive functions. The major problem is that the optimizations introduced before operate in nontrivial ways across function boundaries, thus creating more functions than expected. The challenge is to derive these functions automatically.

*The problem: Loop merging across function boundaries.* To illustrate the problem, consider the Cooley-Tukey FFT (2). A direct recursive implementation would consist of four steps corresponding to the four matrix factors in (2). Two of the steps would call smaller DFTs:

```
void dft(int n, cpx *y, cpx *x) {
 int k = choose_factor(n);
 int m = n/k;
 cpx *t1 = Permute x with L(n,k);
 // t2 = (I_k tensor DFT_m)*t1
 for(int i=0; i<k; ++i)
 dft(m, t2 + m*i, t1 + m*i);
 // t3 = T^n_m*t2, f() computes
 // diagonal entries of T
 for(int i=0; i<n; ++i)
 t3[i] = f(i) * t2[i];
 // y = (DFT_k tensor I_m)*t3,
 // cannot call dft() recursively,
 // need strided I/O
 for(int i=0; i<m; ++i)
 dft_stride(k, m, y + i, t3 + i);
}
// to be implemented
void dft_stride(int n, int stride,
 cpx *Y, cpx *X);
```

Note how even this simple implementation is not self-contained. A new function `dft_stride` is needed that accesses the input in a stride and produces the output at the same stride (see the data flow in [Fig. 2](#)).

However, as explained before, loops should be merged where possible. For fixed size code, Spiral would merge the first loop with the second, and the third loop with the fourth, using  $\Sigma$ -SPL rewriting. The same can be done in the general size recursive implementation, but the merging crosses function boundaries:

```
void dft(int n, cpx *y, cpx *x) {
 int k = choose_factor(n);
 // t1 = (I_k tensor DFT_m)L(n,k)*x
 for(int i=0; i < k; ++i)
 dft_iostride(m, k, 1, t1 + m*i,
 x + m*i);
 // y = (DFT_k tensor I_m) T^n_m
 // diagonal entries of T are now
 // precomputed in precomp_f[]
 for(int i=0; i < m; ++i)
 dft_scaled(k, m, precomp_f[i],
 y + i, t1 + i);
}

// to be implemented
void dft_iostride(int n, int istride,
 int ostride, cpx *y, cpx *x);
void dft_scaled(int n, int stride,
 cpx *d, cpx *y, cpx *x);
```

Now there are two additional functions: `dft_iostride` reads at a stride and writes at a different stride, and `dft_scaled` first scales the input and then performs a DFT at a stride.

So at least three functions are needed with different signatures. However, the two additional functions are also implemented recursively, possibly spawning new functions. Calling these functions *recursion steps*, the main challenge is to automatically derive the complete set of recursion steps needed, called the “recursion step closure.” Further, for each recursion step in the closure, the signature has to be derived.

*Recursion step closure by  $\Sigma$ -SPL rewriting.* Spiral derives the recursion step closure using  $\Sigma$ -SPL and the same rewriting system that is used for loop merging.

For example, the two additional recursion steps in the optimized implementation above are automatically obtained from (2) as follows. Recursion steps are marked by overbraces.

$$\begin{aligned}
 \overbrace{\text{DFT}_n} &\rightarrow (\overbrace{\text{DFT}_{n/k}} \otimes I_k) T_k^n (I_{n/k} \otimes \overbrace{\text{DFT}_k}) L_{n/k}^n \\
 &\rightarrow \left( \sum_{i=0}^{k-1} S(h_{i,k}) \overbrace{\text{DFT}_{n/k}} G(h_{i,k}) \right) \\
 &\quad \text{diag}(f) \left( \sum_{j=0}^{n/k-1} S(h_{jk,1}) \overbrace{\text{DFT}_k} G(h_{jk,1}) \right) P(\ell_{n/k}^n) \\
 &\rightarrow \sum_{i=0}^{k-1} S(h_{i,k}) \overbrace{\text{DFT}_{n/k}} \text{diag}(f \circ h_{i,k}) G(h_{i,k}) \\
 &\quad \sum_{j=0}^{n/k-1} S(h_{jk,1}) \overbrace{\text{DFT}_k} G(h_{jk,1}) \\
 &\rightarrow \sum_{i=0}^{k-1} \overbrace{S(h_{i,k}) \text{DFT}_{n/k} \text{diag}(f \circ h_{i,k}) G(h_{i,k})} \\
 &\quad \sum_{j=0}^{n/k-1} \overbrace{S(h_{jk,1}) \text{DFT}_k G(h_{jk,1})} \tag{5}
 \end{aligned}$$

The first step applies the breakdown rule (2). The second step converts to  $\Sigma$ -SPL. The third step performs loop merging as explained before. The fourth step expands the braces to include the context. The two expressions under the braces correspond to the two functions

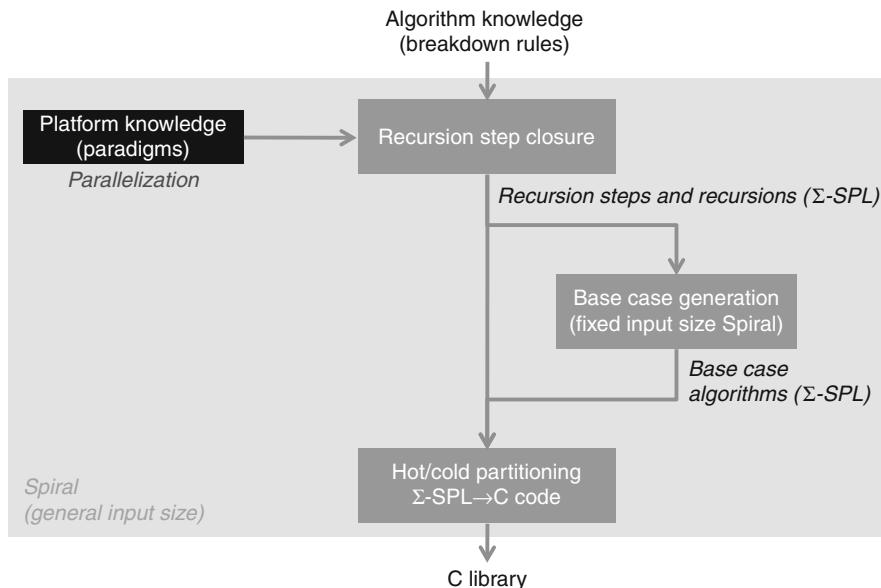
`dft_iostride` and `dft_scaled`. The process is now repeated for the expression under the braces until closure is reached. In this example, only one additional function is needed, i.e., the recursion step closure consists of four mutually recursive functions. The derivation of the recursion steps also yields a  $\Sigma$ -SPL specification of the actual recursion, i.e., their implementation by a recursive function (e.g., (5) for  $\text{DFT}_n$ ).

For the best performance, the braces may be extended to also include the loop represented by the iterative sum. Moving the loop into the function enables better C/C++ compiler optimizations.

If the implementation is vectorized or parallelized, the initial breakdown rules are first rewritten as explained before and then the closure is computed. The size of the closure is typically increased in this case.

*Program generation for general input size: Overview.* The overall process is visualized in Fig. 6. The input to Spiral is now a (sufficient) set of breakdown rules for a given transform or transforms. The rules are parallelized if desired, using the appropriate paradigms; then the recursion step closure is computed, which also yields the actual recursions.

The resulting recursion steps need base cases for termination. These are generated using the algorithm generation block from the fixed input size Spiral (Fig. 3)



**Spiral. Fig. 6** Spiral program generator for general input size libraries

for a range of small sizes (e.g.,  $n \leq 32$ ) to improve performance. These, the recursion steps, and the recursions are fed into the final block to generate the final library. Among other junctions, the block performs the *hot/cold partitioning* that determines which parameters in a recursion step are precomputed during initialization and which become parameters of the actual compute function. Finally, the actual code is generated (which now includes recursive functions) and integrated into a common infrastructure to obtain a complete library.

Many details are omitted in this description and are provided in [29, 30].

## Extensions

A major question is whether the approach taken by Spiral can be extended beyond the domain of linear transforms, while maintaining both the basic principles outlined in the introduction and the ability to automatically perform the necessary transformations and reasoning. First progress in this direction was made in [7] with the introduction of the operator language (OL). OL generalizes SPL by considering operators that may be nonlinear and may have more than one vector input or output. Important constructs such as the tensor product are generalized to operators. First results on program generation for functions such as radar imaging, Viterbi decoding, matrix multiplication, and the physical layer functions of wireless communication protocols have already been developed.

## Related Entries

- [ATLAS \(Automatically Tuned Linear Algebra Software\)](#)
- [FFT \(Fast Fourier Transform\)](#)
- [FFTW](#)

## Bibliographic Notes and Further Reading

Spiral is based on early ideas on using tensor products to map FFT algorithms to parallel supercomputers [16]. The first paper describing SPL and the SPL compiler is [32]. See also [14] for basic block optimizations for transforms. The first complete basic Spiral system including SPL algorithm generation and search was presented in [23], with a more extensive treatment in [24]

and probably the best overview paper [22], which fully develops SPL for a variety of transforms. The path to complete automation in the transform domain continued with  $\Sigma$ -SPL and loop merging [11], the introduction of rewriting systems for SIMD vectorization [8, 13] and base case generation [9], SMP parallelization [12], and distributed memory parallelization [2, 3]. The final step to generating general size, parallel, adaptive libraries was made in [29, 30]. The generated libraries are modeled after FFTW [15], which is written by hand but uses generated basic blocks [14].

The most important extensions of Spiral are the following. Extensions to generate Verilog for field-programmable gate-arrays (FPGAs) are presented in [18, 19]. Search techniques other than dynamic programming are developed in [5, 26]. The use of learning to avoid search was studied in [6, 25]. Finally, [4, 7, 17] make the first steps toward extending Spiral beyond the transform domain including the first OL description. The Spiral project website with more information and all publications is given in [1].

A good introduction to FFTs using tensor products is given in the books [27, 28]. A comprehensive overview of algorithms for Fourier/cosine/sine transforms is given in [21, 31]. A good introduction to mapping FFTs to multicore platforms is given in [10].

## Bibliography

1. Spiral project website. [www.spiral.net](http://www.spiral.net)
2. Bonelli A, Franchetti F, Lorenz J, Püschel M, Ueberhuber CW (2006) Automatic performance optimization of the discrete Fourier transform on distributed memory computers. In: International symposium on parallel and distributed processing and application (ISPA), Lecture notes in computer science, vol 4330. Springer, Berlin, pp 818–832
3. Chellappa S, Franchetti F, Püschel M (2009) High performance linear transform program generation for the Cell BE. In: Proceedings of the high performance embedded computing (HPEC), Lexington, 22–23 September 2009
4. de Mesmay F, Chellappa S, Franchetti F, Püschel M (2010) Computer generation of efficient software Viterbi decoders. In: International conference on high performance embedded architectures and compilers (HiPEAC), Lecture notes in computer science, vol 5952. Springer, Berlin, pp 353–368
5. de Mesmay F, Rimmel A, Voronenko Y, Püschel M (2009) Bandit-based optimization on graphs with application to library performance tuning. In: International conference on machine learning (ICML), ACM international conference proceedings series, vol 382. ACM, New York, pp 729–736

6. de Mesmay F, Voronenko Y, Püschel M (2010) Offline library adaptation using automatically generated heuristics. In: International parallel and distributed processing symposium (IPDPS)
7. Franchetti F, de Mesmay F, McFarlin D, Püschel M (2009) Operator language: a program generation framework for fast kernels. In: IFIP working conference on domain specific languages (DSL WC), Lecture notes in computer science, vol 5658. Springer, Berlin, pp 385–410
8. Franchetti F, Püschel M (2002) A SIMD vectorizing compiler for digital signal processing algorithms. In: International parallel and distributed processing symposium (IPDPS). pp 20–26
9. Franchetti F, Püschel M (2008) Generating SIMD vectorized permutations. In: International conference on compiler construction (CC), Lecture notes in computer science, vol 4959. Springer, Berlin, pp 116–131
10. Franchetti F, Püschel M, Voronenko Y, Chellappa S, Moura JMF (2009) Discrete Fourier transform on multicore. *IEEE Signal Proc Mag* 26(6):90–102
11. Franchetti F, Voronenko Y, Püschel M (2005) Formal loop merging for signal transforms. In: Programming languages design and implementation (PLDI). ACM, New York, pp 315–326
12. Franchetti F, Voronenko Y, Püschel M (2006) FFT program generation for shared memory: SMP and multicore. In: Supercomputing (SC). ACM, New York
13. Franchetti F, Voronenko Y, Püschel M (2006) A rewriting system for the vectorization of signal transforms. In: High performance computing for computational science (VECPAR), Lecture notes in computer science, vol 4395. Springer, Berlin, pp 363–377
14. Frigo M (1999) A fast Fourier transform compiler. In: Proceedings of the programming language design and implementation (PLDI). ACM, New York, pp 169–180
15. Frigo M, Johnson SG (2005) The design and implementation of FFTW3. *Proc IEEE* 93(2):216–231
16. Johnson J, Johnson RW, Rodriguez D, Tolimieri R (1990) A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures. *IEEE Trans Circ Sys* 9:449–500
17. McFarlin D, Franchetti F, Moura JMF, Püschel M (2009) High performance synthetic aperture radar image formation on commodity architectures. *Proc SPIE* 7337:733708
18. Milder PA, Franchetti F, Hoe JC, Püschel M (2008) Formal datapath representation and manipulation for implementing DSP transforms. In: Design automation conference (DAC). ACM, New York, pp 385–390
19. Nordin G, Milder PA, Hoe JC, Püschel M (2005) Automatic generation of customized discrete Fourier transform IPs. In: Design automation conference (DAC). ACM, New York, pp 471–474
20. Press WH, Flannery BP, Teukolsky SA, Vetterling WT (1992) Numerical recipes in C: the art of scientific computing, 2nd edn. Cambridge University Press, Cambridge
21. Püschel M, Moura JMF (2008) Algebraic signal processing theory: Cooley-Tukey type algorithms for DCTs and DSTs. *IEEE Trans Signal Proces* 56(4):1502–1521
22. Püschel M, Moura JMF, Johnson J, Padua D, Veloso M, Singer B, Xiong J, Franchetti F, Gacic A, Voronenko Y, Chen K, Johnson RW, Rizzolo N (2005) SPIRAL: code generation for DSP transforms. *Proc IEEE* (Special Issue on Program Generation, Optimization, and Adaptation) 93(2):232–275
23. Püschel M, Singer B, Veloso M, Moura JMF (2001) Fast automatic generation of DSP algorithms. In: International conference on computational science (ICCS), Lecture notes in computer science, vol 2073. Springer, Berlin, pp 97–106
24. Püschel M, Singer B, Xiong J, Moura JMF, Johnson J, Padua D, Veloso M, Johnson RW (2004) SPIRAL: a generator for platform-adapted libraries of signal processing algorithms. *J High Perform Comput Appl* 18(1):21–45
25. Singer B, Veloso M (2001) Learning to generate fast signal processing implementations. In: International conference on machine learning (ICML). Morgan Kaufmann, San Francisco, pp 529–536
26. Singer B, Veloso M (2001) Stochastic search for signal processing algorithm optimization. In: Supercomputing (SC). ACM, New York, p 22
27. Tolimieri R, An M, Lu C (1997) Algorithms for discrete Fourier transforms and convolution, 2nd edn. Springer, Berlin
28. Van Loan C (1992) Computational framework of the fast Fourier transform. SIAM, Philadelphia
29. Voronenko Y (2008) Library generation for linear transforms. Ph.D. thesis, Electrical and Computer Engineering, Carnegie Mellon University
30. Voronenko Y, de Mesmay F, Püschel M (2009) Computer generation of general size linear transform libraries. In: International symposium on code generation and optimization (CGO). IEEE Computer Society, Washington, DC, pp 102–113
31. Voronenko Y, Püschel M (2009) Algebraic signal processing theory: Cooley-Tukey type algorithms for real DFTs. *IEEE Trans Signal Proces* 57(1):205–222
32. Xiong J, Johnson J, Johnson RW, Padua D (2001) SPL: a language and compiler for DSP algorithms. In: Programming languages design and implementation (PLDI). ACM, New York, pp 298–308

## SPMD Computational Model

FREDERICA DAREMA

National Science Foundation, Arlington, VA, USA

### Definition of the Subject

The Single Program – Multiple Data (SPMD) parallel programming paradigm is premised on the concept that all processes participating in the execution of a program work cooperatively to execute that program, but at any given instance different processes may execute different instruction-streams, and act on different data

and on different sections in the program, and whereby these processes dynamically self-schedule themselves, according to the program workflow and through synchronization constructs embedded in the application program. SPMD programs comprise of serial, parallel, and replicate sections.

## Introduction

The (SPMD) Single Program-Multiple Data model [1–6] is premised on the concept that all processes participating in the (parallel) execution of a program work cooperatively to execute this program, but at any given instance, through synchronization constructs embedded in the application program, different processes may execute different instruction-streams, and act on different data and on different sections in the program; thus the name of the model: Single Program – Multiple Data. The model was proposed by the author in January 1984 [1], as a means for expressing and enabling parallel execution of applications on highly parallel MIMD computational platforms. The term “single-program” was used for emphasis on the parallel execution of a given program (i.e., concurrent execution of tasks in a given program), in distinction from the environments of that time, where OS-level concurrent tasks would (concurrently) execute different programs on the multiprocessors of that time (e.g., IBM 3081). The initial motivation for the SPMD model was to enable the expression of the (then) high degrees of parallelism supported by the IBM Research Parallel Processor Prototype (RP3) [7] parallel computer system. The model was first implemented in the Environment for Parallel Execution (EPEX) [2–6] programming environment (one of the first general-purpose parallel programming environments, and of course the first to implement SPMD). This entry is an excerpt of a more comprehensive treatise on the SPMD [8] which puts the motivation for the SPMD in the context of the landscape of the computer platforms and software support of the early- to mid-1980s, and discusses experiences, effectiveness, and impact of the SPMD in expediting adoption of parallelism in that time frame and as it has influenced derivative programming environments in the intervening 25 years.

The SPMD model fostered a new approach to parallel programming, differing than the Fork & Join (or Master-Slave) model predominantly pursued by others at around the 1984 timeframe. In SPMD, all processes

that participate in the parallel execution of a program commence execution of the program, and each process is dynamically self-scheduled and selects work-tasks to execute (self-allocated work), based on parallelization directives embedded in the program and according to the program workflow. In the SPMD model, a process represents a separate instantiation of the program, and processes participating in the cooperative execution of a program (also referred to as parallel processes) execute distinct instruction streams in a coordinated way through these embedded parallelization directives (also referred to as synchronization directives). With respect to parallel execution, in general a program consists of sections that are executed by one process (*serial sections*) and sections that can be executed by multiple cooperating processes (i.e., *parallel sections*, that can be executed by several processes in a cooperative and concurrent manner, and *replicate sections*, where the computations are executed by every process; parallel sections may include one or more levels of nested parallel sections). Regardless of the section considered, the beginning and end of serial and of parallel sections (and nested parallel sections) are points of synchronization of the processes (*synchronization points*), that is, points of coordination and control of the execution path of the participating processes. The definition in [1, 2]: “*all processes working together will execute the very same program*,” uses the word “same” to emphasize concurrent execution of a given program by these (“multiple”) processes, and not to imply that all these processes execute identical instruction streams (as it has been interpreted by some). In expressing parallelism through the SPMD model, the flow of control is distinguished in two classes: the *parallel flow of control* is the flow of control followed by the processes while executing a serial or a parallel section in the program; the *global flow of control* is the flow of control followed by the processes as they step through the synchronization points of the program, according to the program workflow. In parallel execution with the SPMD model, the participating processes follow a different *parallel flow of control*, but all the processes follow the same *global flow of control*.

Conceptually, the SPMD model has been from the outset a general model, enabling to express parallelism for concurrent execution of distinct instruction streams and allowing application-level parallelization control

and dynamic scheduling of work-tasks ready to execute (with process self-scheduling). The model is able to support general MIMD task-level parallelism, including nested parallelism, and is applicable to a range of parallel architectures (those considered at the time the model was proposed and other parallel and distributed architectures that have subsequently appeared) in more efficient and general ways than the then proposed alternate parallel programming models, such as those based on Fork-and-Join, SIMD, and data-parallel and systolic-arrays approaches. In fact SPMD allows combining with and implementing such models as parallel execution models invoked under the SPMD rubric. Applied to parallel architectures supporting shared memory (for example RP3) the SPMD is a global-view programming model. From its inception, SPMD allowed efficient implementations of expressing parallelism, through parallelization directives inserted in the initial serial application programs, the ensuing “parallelized” program compiled with the standard serial compilers. The parallelization directives enabled application-level dynamic scheduling and control of the parallel execution, with efficient runtime implementations, requiring minimal interaction with the OS, avoiding (the heavy) OS level-synchronization overheads, and without requiring new, parallel OS services. These were important considerations in the 1984 time frame, the SPMD showed early-on that it allowed the parallelization of non-trivial programs; many were production-level application programs, for example, from the areas of applied physics, aerospace, and design automation. Thus the SPMD expedited adoption of parallelism in the mid-1980s, as the model enabled to determine that it was not difficult to express parallelism, and map and execute such non-trivial applications on parallel machines, and thus exploit parallelism and do so flexibly and adaptively, and with efficiency.

SPMD enabled creating parallel versions of FORTRAN and C programs, and also enabled the parallel execution of such programs without the need for new parallel languages. To date, there are many environments that have implemented SPMD, primarily for science and engineering parallel computing, but also for commercial applications. The most widely used today being MPI [9] for distributed multiprocessing systems or “message-passing” systems, and later used also for shared memory systems; the predecessor of

MPI being PVM [10], which appeared in the late-1980s as one of the first popular implementations of SPMD for “message-passing” systems. The SPMD model through its implementation in MPI is widely used for exploiting today’s heterogeneous complex parallel and distributed computational platforms, including computational grids [11], which embody many types of processors, multiple levels of memory hierarchy, and multiple levels of networks and communication, and which may employ a combination of SPMD as well as Fork-and-Join programming environments. Other influential programming environments also based on SPMD include OpenMP [12] for shared-memory parallel programming in FORTRAN, C, and C++; Titanium [13] for parallelization of Java programs; and Split-C [14] and Unified Parallel C (UPC) [15] for parallel execution of C programs.

## The SPMD Model

In the SPMD model, a parallel program conceptually consists of serial, parallel (including nested parallel), and replicate sections. The program data are distinguished into application data and synchronization data; synchronization data are shared among the parallel processes, and application data are distinguished into private and into shared data. In the following are discussed how these parallelization aspects are treated in SPMD:

- *Serial sections* are executed by one process (either a designated process, or more generally, and typically, the first one to arrive at that section). The other processes that arrive at this serial section, through the “*serial-section synchronization directives*” are directed to bypass its execution; there, they either wait at the end of the section for its execution to complete, or they may proceed to seek and execute available work-task(s) in subsequent section(s), if that is allowed by the program workflow dependencies (and with appropriate synchronizations imposed – soft and hard barriers, discussed later on in their EPEX implementation).
- *Parallel sections* are executed concurrently by processes that arrive at the beginning of such a section. These processes, through “*parallel-section synchronization directives*” dynamically self-schedule themselves and get allocated with the next available parallel work-task to execute. As each of these

processes arriving in a parallel section is self-assigned a work - task, and as each process completes its task of work, these processes can seek the next available work-task to execute in that parallel section; or if all work tasks in the parallel section have been allocated to processes, the remaining process(es) can proceed to the end of that parallel section, and either wait there till all the work of the parallel section is completed, or continue-on to the next section of the program or other subsequent sections of the program as allowed by the program workflow dependencies (and with appropriate synchronizations imposed – *soft* and *hard* barriers, discussed later on in their EPEX implementation). Parallel loops in programs are the predominant form of a parallel section and thus major targets of parallel execution; however, SPMD also supports other forms of general task parallelism. Later in this section nested parallelism support in SPMD is discussed.

- *Replicate sections* of the program are sections allowed to be executed by all processes (it is the default mode of execution that was envisioned when the SPMD was first proposed). Such sections involve parts of the computation (typically small portion of the total amount of the computation) and where allowing all processes to replicate the execution of the computation is more efficient than having one process execute the section (while the others wait) and then make the result of the computation available to other processes (as shared data or as communicated messages). This is applicable, for example, where replicate execution avoids: the serialization overhead, the busy-waiting and polling a synchronization semaphore by the other processes (with potential additional overhead due for example contention in this semaphore, and also potential network contention), and the overhead of making the resulting data available to the other process (e.g., in shared memory architectures, potential of contention upon access of the resulting data placed in shared memory; or network contention upon broadcasting the results to the other processes through message passing, in logically distributed, “message-passing” architectures). Especially for message-passing environments, such overheads can be rather high, and in such cases the approach of replicate computations can be a more efficient alternative for certain portions of the computation.
- *Shared and Private Data:* These refer to application data and synchronization data. For parallel machine architectures which support shared as well as private memory, parallelism is expressed by considering two types of data: *shared data* and *private data*; each of the parallel processes having read and write capability on the shared data, while private data are exclusive to each process (and typically the private data of a process would reside in the local memory of the processor on which the process executes). *Application shared data* and *synchronization data* are declared as *shared data*. SPMD was originally proposed for architectures supporting a mix of shared and (logically local – private) memory, and the approach followed in the original SPMD implementation was for the default to be the *private data* (to each parallel process); this decision was made (by the author) on the basis that it is easier (especially for the user) to identify the application shared data and the synchronization data, rather than explicitly defining the private data as other approaches have done. For message-passing architectures, where the SPMD was also applied, the *application shared data* are partitioned into per process private data, and “sharing” of such data is effected through “messages” exchanged between parallel processes, as needed during the course of the computation. It is the belief of this author that expressing parallelism through message-passing is more challenging than supporting parallelism through a “shared-memory plus private memory” architecture support, and that is also the case with the kinds of programming models for expressing parallelism (more on this in later discussion on MPI).
- *I/O:* The SPMD model allows general flexibility in handling I/O. Any of the parallel processes executing a parallel program are allowed to perform I/O, and (as discussed later-on in the context of the EPEX programming environment) all processes can have read/write access to any of the parallel program’s files. One can argue that the typical approach would be for I/O to be treated as a serial action, although that is not necessarily always so. For example, upon

commencing execution of a program, one process typically would be assigned to read the input data file, and perform the program state initialization. However, there are many other instances in the program execution, for example, within parallel sections, where it may be more efficient to allow each process to read data from a file, and as needed write data back into a file. In this case, for flexibility, random-access file structure would be more desirable, than sequential record files; it is then not necessary to attach a file to a given process, which is a possible but more restrictive approach.

In the early implementations of the SPMD, where processes were implemented by heavy-weight mechanisms (such as VMs – Virtual Machines), to enable efficient parallel execution, all processes were created in the beginning of the program execution, and this has been applied to most SPMD implementations since then. Noted, however, that the SPMD model does not preclude by principle the case where additional processes can be created and participate in the program execution; this was allowed through EPEX implemented directives, allowing such additional processes to skip the already executed portion of the entire computation. However, at the time SPMD was proposed and within the systems then available for the implementation of the model, creation of such processes and inheriting program state was expensive, unlike capabilities that were enabled later on, like creation of light-weight threads.

Nesting in SPMD can be implemented by allowing enough processes to enter an (outer) parallel loop, for example, allocate several processes – a group of processes – to a given outer loop iteration, allowing only one of these processes to execute the outer loop portion (for example, with appropriate directives treating it as a “nested serial”) and then allow all processes in this group to execute in parallel the inner loop. In fact this author experimented with such approaches, as well as a hybrid of SPMD combined with Fork-and-Join capabilities to implement nesting, but they were never implemented in a production way, due to the overheads of spawning VM/SP virtual processes, but also because in the mid-1980s timeframe the degrees of parallelism in the applications (e.g., dimensionality of the outer parallel loops in a program) exceeded the numbers of processors in the commercial and in the

prototype systems of that time, so exploiting nesting was more of an intellectual endeavor rather than a practical need. With threading capabilities, other implementations can be used to support more elegantly execution of nested parallel computations, for example, by each process spawning threads to execute the inner iterations of the loop, like for example in OpenMP. Later in this entry is discussed the need for nested or multilevel threading capabilities, gang scheduling of threads, partially shared data and active data-distribution notions, in the context of multi-level parallelization hierarchies that will be encountered in the emerging and future parallel architectures.

The SPMD model does not restrict how many processes can run on each parallel processor. While often one process executes per processor, there are cases where multiple processes may be spawned to execute per processor, for example, to mask memory latency, shared memory contention, I/O, etc. Also SPMD does not require a given process to be attached to a given processor; however, the typical approach is to bind a process (or a thread to a processor), to avoid cost of migration, context-switch, and better exploitation of (local data) cache; however, there are situations where, as long as there is no thrashing, a process may be moved, for example, to improve performance, and in other cases for fault tolerance or recovery [16, 17].

Other SPMD features are provided as illustrative examples of the EPEX programming environment discussed next.

## The EPEX Programming Environment – Implementation of the SPMD Model

The first implementation of SPMD was in the EPEX environment [2–5], initially implemented on IBM multiprocessors like the dual processor IBM/3081 and, later, the six-processor IBM/3090 vector multiprocessor (through MVS/XA [18]). The IBM/3081 through the VM/SP OS [19] supported execution of multiple virtual machines (VMs), running in time-shared mode (on each node of the 3081). This capability was used to simulate parallel execution by a large number of processors working together on a single program, each simulated parallel processor and its local memory (correspondingly for each of the parallel processes) being simulated by a VM (a virtual machine). Although in

practice the 3081 hardware supported 2-way parallelism (and the 3090 hardware supported 6-way parallelism), the simulation environment created was used to simulate up to 64 parallel processors; of course, in principle the environment could simulate even higher numbers of processors. To simulate execution on parallel platforms, like RP3, the VM/SP OS support of the Writable Shared Segments (VM/WSS) capability was used, which allowed to establish a portion of the virtual machines' memory as shared across a set of VMs (in read and write modalities); this feature allowed to simulate the shared-memory of a parallel system (like RP3), while the private memory of each virtual machine simulated the local memory of an RP3 processor, and the private memory of the corresponding process. The VM/WSS was also used to emulate message-passing environments, by using the WSS-based (virtual) shared memory as a "mailbox" for messages, and thus allowing to also experiment and to demonstrate the use of SPMD as a model also supporting message-passing parallel computation. These capabilities were used to create a simulation platform for the EPEX programming environment, and were also used for the development of other simulation tools for analysis of execution on various parallel machines with shared-plus-local memory systems as well as for "message - passing" distributed systems. While over the last two decades most parallel systems have been used with MPI (a message-passing environment), it behooves elaborating on shared-plus-local memory systems because the emerging multicore-based architectures are expected to support such memory organizations within the likely set of memory hierarchies.

Initially, parallelization of programs under EPEX was enabled by implementing the "serial," "parallel," and "barrier" parallelization directives through synchronization subroutine calls inserted in the program; the first EPEX implementation was applied to parallelization of FORTRAN programs, and shared data declared explicitly, through FORTRAN Shared COMMON statements. Shortly thereafter, the synchronization subroutine calls were replaced by corresponding parallelization macros, with the development of a pre-processor (source-to-source translator from macros to subroutine calls). Denoting the parallelization directives through macros, and likewise for the declaration of application shared data, allowed converting serial

programs to their parallel counterparts, more easily and elegantly. Additional synchronization constructs were also expressed through other appropriate macros. All these parallelization macros inserted in the program were then expanded by the "EPEX Preprocessor" into the parallel program version, by inserting the SPMD parallelization subroutines (specifically referred to here is the EPEX FORTRAN Preprocessor source-to-source translator [20–22]; later a preprocessor for C was also developed [23]). Then the ensuing "parallelized program" was compiled through a standard serial FORTRAN or C compiler. By '87-'88 the SPMD-based EPEX system had been installed in ten IBM sites (including IBM Scientific Centers, such as those in Palo Alto, CA-USA and Rome, Italy), three National and International Labs (LANL, ANL, and CERN); two industrial partners (MartinMarietta and Grumman), and eight universities.

The syntax and utilization of the initial EPEX implementation is presented in [2–5]. The macros implementing SPMD and the EPEX preprocessor are presented in detail in [6, 19–21] which document in more detail the preprocessor-based EPEX environment. Here, for reference, an illustrative sample of the set of the EPEX macros is given with a brief description of their use:

- The traditional FORTRAN DO-loop was designated through the:

```
@DO [stmt#|label] index = n1, n2, [n3]
 [CHUNK = chunksize]
 ... loop body ...
[stmt#] @ENDO [label] [WAIT | NOWAIT]
```

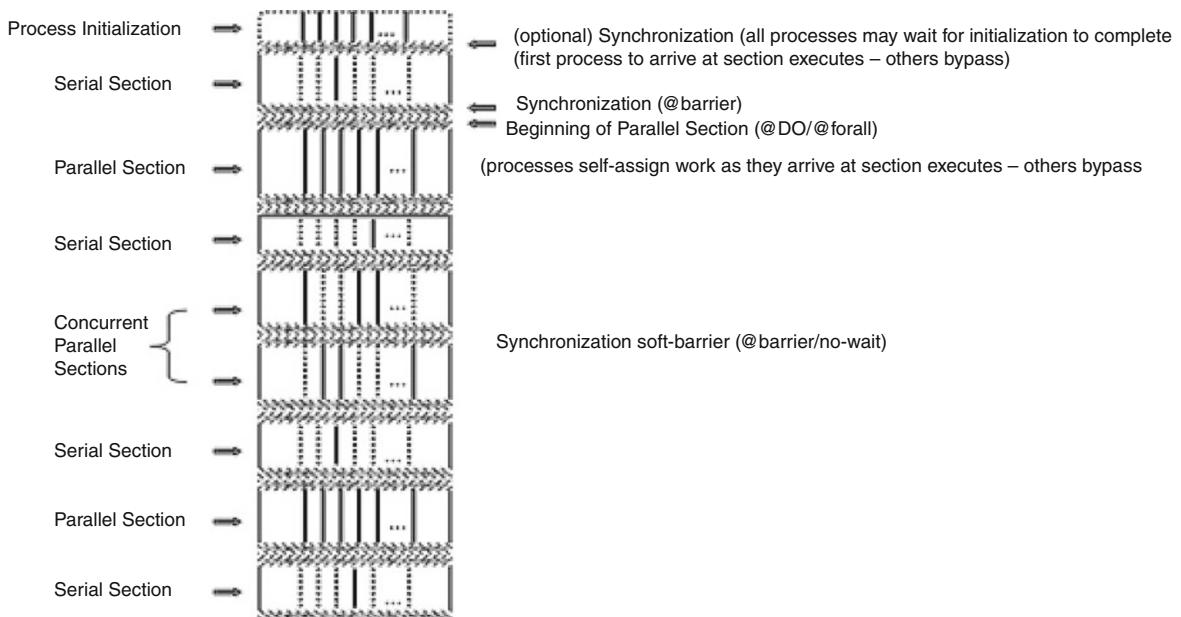
Processes entering the @DO get self-assigned with the next available loop-iteration(s), through synchronization routines utilizing the Fetch-and-Add (F&A) [24] RP3 primitive (implemented in the simulated environment through the Compare&Swap instruction); processes that arrive after all the work in the loop has been allocated or all the work in the loop has been completed proceed to the end of the section (@ENDO); there they may wait for the loop execution to be completed, or proceed to subsequent section(s), as allowed by the program workflow dependencies. The CHUNK option allows processes to get assigned more than one iterations at a time, thus decreasing the overhead of parallelism; the WAIT option requires processes to wait for the

execution of the preceding section (in this case the loop body) to be completed before continuing to the next section of the program; the NOWAIT option allowed processes that reach the end of the section to continue execution of subsequent section(s), thus allowing multiple concurrent sections of the program (or subroutines) to be executed concurrently, as allowed by dependencies in the program workflow.

- The serial section is bounded by the @SERBEG and @SEREND macros. The default is for the first process to encounter the @SERBEG to execute the section, the other processes proceed to @SEREND, with [WAIT | NOWAIT] options similar to the ones for a parallel loop. The environment also allows designating a given process to execute a given serial section.
- The @WAITFOR macro can be inserted at any point of the program and designates that processes cannot go beyond that point, until a *logical condition* specified by the argument of the macro is satisfied; for instance, this construct can be used for imposing various kinds of soft barriers – for example, for the processes to wait until execution of the preceding section(s) is completed.

- The @BARRIER macro can be inserted at any point of the program and forces all processes participating in the execution of the program to arrive at that point of the program (hard barrier).
- The parallel processes are endowed with an identity [@MYNUM]; the MYNUM for each process can be a parameterized assignment. This property can be used in several ways: for example, for designating a given process to perform I/O to a file if that is desired, or execute a given section (e.g., a serial section) that can be designated to be executed by a given process (of course in general this is not mandatory, as the default option for a section, and more generally the next available work-task is to be executed on a “first-arrives-executes” basis).
- @SHARED [application shared data: parameters, arrays] allows the user to designate the application shared data; by default all other data of the application are assumed as private to each parallel process; @SHARED [synchronization data] are created automatically by the preprocessor and designated all the shared data used by the synchronization constructs.

A schematic of an example of SPMD execution is shown in Fig. 1.



SPMD Computational Model. Fig. 1 Schematic of an example of SMPD MIMD task-parallel execution

- Handling of I/O: In [25, 26] are discussed implementations of ideas for parallel I/O presented in the previous section, which allowed I/O by multiple processes, and not necessarily serializing I/O. A file could be opened, written into and closed, and then made available to other processes. This could be done at the end of a parallel section or at the end of an outmost iteration of the program, depending on the problem needs; such implementations might be enabled via a barrier at the end of the outermost loop, or if no barrier was imposed the file maybe accessed (in read-write mode) by another process (which may need to spin-wait if the file had not been closed by the previous process). The @MYNUM designation allowed to tie a given process to a file if that was desirable. The advent of newer file-systems (such as GPFS [27, 28]) allows more flexible mechanisms for parallel I/O in SPMD programs, and parallel I/O is supported in programming environments, like OpenMP. Furthermore, as mass storage technologies evolve from disk to solid-state and nano-devices, it is possible that parallel I/O could be implemented through other than file-based methods (perhaps something akin to a “DBMS-like” I/O management).

Through the EPEX parallelization directives, the participating processes executing cooperatively the program step through the synchronization points in the program, and are dynamically allocated the next work-task or execution path to take. In that sense all participating processes follow the same global flow of control in the program. In executing serial or parallel sections, each process is allocated the next available work-task, and thus they may execute different instructions at any given time and act on different parts of the program (including potentially different procedures or subroutines), as consistent with the workflow dependencies in the program. A given process may follow a different parallel flow of control during different passes of an outer iteration of the entire program. The synchronization routines provided in EPEX allowed executing outer iterations of the entire program (that is repeated execution of the serial and parallel sections of the program) without necessarily imposing the need for the programmer to introduce explicit blocking at the end of the outer iteration of the program.

The EPEX programming environment allowed to parallelize and simulate the parallel execution of a large number of applications, to understand their characteristics, and assess the efficacy and effectiveness of parallelism. In the 1984 – 1986 time-span, over 40 applications were parallelized with SPMD, and executed in parallel under the EPEX programming environment. The set of these applications (mostly numeric, but also non-numeric) included: Fluid Dynamics (e.g., advanced turbulent-flow programs [29], and shallow water and heart blood-flow problems); Radiation Transport (Discrete Ordinates methods and Monte-Carlo); Design Automation (Chip placement and Routing by Simulated Annealing [30–33]; and Fault simulation); Seismic, Reservoir Modeling, Weather Modeling, Pollution Propagation; Physics Applications (Band Structure, Spin-Lattice, Shell Model); Applied Physics and Chemistry Applications (Molecular Dynamics); Epidemic Spread; Computer Graphics and Image Processing (e.g., ray-tracing); Numerical and non-numeric Libraries (FFTs, Linear Solvers, Eigen-solvers, sorting), etc.

## Advancing into the Future: Directions, Opportunities, Challenges, and Approaches

*Emerging Computational Platforms and Emerging Applications Systems.* Increasingly, large-scale distributed systems are deploying as their building blocks high-performance multicore processors (and in combination with special-purpose processor chips, like GPUs), as are the emerging petaflops and future hexaflops platforms. In fact, it is also conceivable that this kind of heterogeneity of processors will be eventually embedded in the multicore chip itself. Such systems, with 1,000s to 100s of thousands of tightly coupled nodes, will be enabled as Grids-in-a-Box (GiBs). Computational platforms include both high-end systems as well as globally-distributed, meta-computing, heterogeneous, networked and adaptive platforms, ranging from assemblies of networked workstations, to networked supercomputing clusters or combinations thereof, together with their associated peripherals such as storage and visualization systems. All these hardware platforms will have potentially not only multiple levels of processors, but also multiple levels of memory hierarchies (at the cache, main memory and storage

levels), and multiple levels of interconnecting networks, with multiple levels of latencies (variable at inter-node and intra-node levels) and bandwidths (differing for different links, differing based on traffic). Moving into the exascale domain, the challenges are augmented as we are faced with prospects of addressing billion-way concurrency, and in the presence of heterogeneity of processing units in a processing node, increasing degrees in the multiple levels of memory hierarchies, and multiple levels of interconnects hierarchies; these are the Grids-in-a-Box, referred to earlier-on, with significantly added complexity, because of the wider range of granularity, needing to expose concurrency from the fine grain to many more levels of coarser grain. The questions range from: how to express parallelism and optimally map and execute applications on such platforms, to how to enable load balancing, and at what level (or levels) is load balancing applied. It becomes evident that static parallelization approaches are inadequate, that one needs programming models that allow expressing parallelism so that it can be exploited dynamically, for load balancing and hiding latency, and for expressing dynamic flow control and synchronization possibly at multiple levels. The ideas of dynamic runtime compiler [34] capabilities discussed below are becoming more imperative.

Furthermore, new application paradigms (e.g., DDDAS – Dynamic Data Driven Applications Systems [35, 36]) that have emerged over the last several years and which entail dynamic on-line integration and feed-back and control between the computations of complex application models and measurement aspects of the application system, leading to *SuperGrids* [37] of integrated computational and instrumentation platforms, as well as other sensory devices and control platforms. In the context of this entry on programming environments and programming models, the implications are that these environments and models will need to support seamlessly execution of applications encompassing dynamically integrated high-end computing with real-time data-acquisition and control components and requirements.

*New Programming Environments and Runtime-Support Technologies* Such environments require technology approaches which break down traditional barriers in existing software components in the application development support and runtime layers, to deliver QoS in application execution. That is, together

with new programming models, new compiler technology is needed, such as the runtime-compiler system (RCS [38]), where part of the compiler becomes embedded in the runtime and where the compiler interacts with the system monitoring and resource managers, as well as performance models of the underlying hardware and software, using such capabilities for optimizing the mapping of the application on the underlying complex platform(s). This runtime-compiler system is aware of the heterogeneity in the underlying architecture of the platforms, such as multi-level hierarchy of processing nodes, memories, and interconnects, with differing architecture, memory organization, and latencies, and will link to appropriately selected components (*dynamic application composition*) to generate consistent code at runtime. Representative examples of developing runtime-compiler capabilities are given in [39] and [40]. Together with the “runtime-compiler” capabilities [33], called for novel approaches and substantial enhancements in computational models to actualize the distributed applications software, including user-provided *assists* to facilitate and enhance the runtime-compiler’s ability to analyze task and data dependencies in the application programs, resolve dependencies, and dynamically optimize mapping across a complex set of processing nodes and memory structure of distributed platforms (such as Grids and GiBs, multicore and GPU-based), with multiple levels of processors and processing nodes, interconnects, and memory hierarchies. It is the thesis of this entry that the models needed should facilitate the RCS (runtime compiling system) to map applications, without requiring detailed resource management specifications by the user and without requiring specification of data location (e.g., proximity – or PGAS). Rather the programming models should incorporate advanced concepts such as “*active data-distribution*,” that is the user specifies that these data are candidates for *active or runtime distribution*, and the RCS determines at runtime how to map them, determines partial sharing, copy/move between memory hierarchies, through dynamic adaptive resource management, decoupled execution and data location/placement, memory consistency models, and multithreaded hierarchical concurrency. Such capabilities may be materialized through a hybrid combination of language, library models, development of OS-supported “hierarchical threading” capabilities and partial sharing of data approaches.

In order to adequately support the future parallel systems, it is advocated here that the new software technologies need to adopt a more integrated view of the architectural layers and software components of a computing system (hardware/software co-design), consisting of the applications, the application support environments (languages, compilers, application libraries, linkers, run-time support, security, visualization, etc.), operating systems (scheduling, resource allocation and management, etc), computing platform architectures, processing nodes and network layers, and also support systems encompassing the computational and application measurement systems. Furthermore, such environments need to include approaches for engineering such systems, at the hardware, systems software, and at the applications levels, so that they execute with optimized efficiency with respect to runtime, quality of service, performance, power utilization, fault tolerance, and reliability. Such capabilities require robust software frameworks, encompassing the systems software layers and the application layers and cognizant of the underlying hardware platform resources.

## Summary

This entry has provided an overview of the SPMD model, its origin and its initial implementations, its effectiveness and impact in expediting adoption of parallelism in the mid-80s, and its use as a vehicle for exploiting parallel and distributed computational platforms for the intervening 25 years. As we consider what are the possible new parallel programming models for new and future computing platforms, as well as present and emerging compute-, data-intensive, dynamically integrated applications executing on such platforms, it is becoming imperative to advance programming models and programming environments by building from past experiences and opening new directions through synergistic approaches of models, advanced runtime-compiler methods, user assists, multi-level threading OS services, and ensuring that the approaches and technologies developed are created in the context of end-to-end software architectural frameworks.

## Acknowledgment

The IBM RP3 Project became the ground for my inspiration and work on the SPMD model; I'm forever grateful for being part of the RP3 team and will always value my collaborations with the RP3 team.

## Bibliography

1. DAREMA-ROGERS F (1984) IBM Internal Communication, Jan 1984
2. DAREMA-ROGERS F, GEORGE D, NORTON VA, PFISTER G (1984) A VM parallel environment. Proceedings of the IBM Kingston Parallel Processing Symposium, 27-29 Nov 1984 (IBM Confidential)
3. DAREMA-ROGERS F, GEORGE DA, NORTON VA, PFISTER GF (1985) A VM based parallel environment. IBM research report, RC11225
4. DAREMA-ROGERS F, GEORGE DA, NORTON VA, PFISTER GF (1985) Environment and system interface for VM/EPEX. IBM research report RI1381, 9/19/1985
5. DAREMA-ROGERS F, GEORGE DA, NORTON VA, PFISTER GF (1985) Using a single-program-multiple-data model for parallel execution of scientific applications. SIAM Conference on Parallel Processing for Scientific Computing, November, 1985, and IBM research report RI1552, 11/19/1985
6. DAREMA F, GEORGE DA, NORTON VA, PFISTER GF (1988) A single-program-multiple-data computational model for EPEX-FORTRAN. Parallel Comput 7:11-24 (received April 1987 upon publication release by IBM - IBM Technical Disclosure Bulletin 29(9) February 1987)
7. PFISTER G ET AL (1984) The research parallel processor prototype (RP3). Proceedings of the IBM Kingston Parallel Processing Symposium, 27-29 Nov 1984 (IBM Confidential); and Proceedings of the ICPP, August 1985
8. DAREMA F Historical and future perspectives on the SPMD computational model. – Forthcoming Publication
9. MPI standard – draft released by the MPI Forum. <http://www.mcs.anl.gov/Projects/mpi/standard.html>
10. PVM – Parallel Virtual Machine. [http://www.csm.ornl.gov/pvm/pvm\\_home.html](http://www.csm.ornl.gov/pvm/pvm_home.html)
11. FOSTER I, KESSELMAN C (EDS) (1999) The grid: blueprint for a new computing infrastructure. Morgan Kaufmann
12. OpenMP. <http://openmp.org/wp/>
13. YELICK KA, SEMENZATO L, PIKE G, MIYAMOTO C, LIBLIT B, KRISHNAMURTHY A, HILFINGER PN, GRAHAM SL, GAY D, COLELLA P, AIKEN A (1998) Concurrency: practice and experience. vol 10, No. 11-13, September-November. An earlier version was presented at the Workshop on Java for High-Performance Network Computing, Palo Alto, CA, Feb 1998
14. CULLER DE, ARPACI-DUSSEAU AC, GOLDSTEIN SC, KRISHNAMURTHY A, LUMETTA S, EICKEN T, YELICK KA (1993) Parallel programming in Split-C. SC, pp 262-273
15. EL-GHAZAWI T, CARLSON W, STERLING T, YELICK KA (2005) UPC: distributed shared-memory programming. Wiley, Hoboken
16. BRONEVETSKY G, MARQUES D, PINGALI K, STODGHILL P (2000) Automated application-level checkpointing of MPI programs. ACM SIGPLAN 38(10):84-94
17. BRONEVETSKY G, MARQUES D, PINGALI K, SZWED P, SCHULZ M (2004) Application-level checkpointing for shared memory programs. ACM Comp Ar 32(5):235-247
18. GEORGE DA MVS/XA EPEX – Environment for parallel execution. IBM research report RC 13158, 9/28/87
19. VM/System Product (VM/SP). [http://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe\\_PP3081.html](http://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP3081.html) (and MVS/XA also supported)

20. Stone JM, Darema-Rogers F, Norton VA, Pfister GF Introduction to the VM/EPEX FORTRAN Preprocessor. IBM research report RC 11407, 9/30/85
21. Stone JM, Darema-Rogers F, Norton VA, Pfister GF The VM/EPEX FORTRAN Preprocessor Reference. IBM research report RC 11408, 9/30/85
22. Bolmarcich T, Darema-Rogers F Tutorial for EPEX/FORTRAN Program Parallelization and Execution. IBM research report RC12515, 2/18/87
23. Whet-Ling C, Norton A (1986) VM/EPEX C preprocessor user's manual. Version 1.0. Technical report RC 12246, IBM T.J. Watson Research Center, Yorktown Heights, NY, October 1986
24. Gottlieb A, Kruskal CP (1981) Coordinating parallel processors: a partial unification. Computer Architecture News, pp 16–24, October 1981
25. Darema-Rogers F I/O capabilities in the VM/EPEX system. IBM research report, RC 12219, 10/9/86
26. Darema F (1987) Applications environment for the IBM research parallel processor prototype (RP3). IBM research report RC 12627, 3/27/87; and Proceedings of the International Conference on Supercomputing (ICS), (published by Springer, Athens, Greece, June 1988)
27. GPFS: <http://www.almaden.ibm.com/StorageSystems/projects/gpfs>
28. Schmuck F, Haskin R (2002) GPFS: a shared-disk file system for large computing clusters. (pdf). Proceedings of the FAST'02 Conference on File and Storage Technologies. USENIX, Monterey, California, USA, pp 231–244. ISBN 1-880446-03-0. [http://www.usenix.org/events/fast02/full\\_papers/schmuck/schmuck.pdf](http://www.usenix.org/events/fast02/full_papers/schmuck/schmuck.pdf). Accessed 18 Jan 2008
29. ARC3D: Pulliam TH Euler and thin layer navier stokes codes: ARC2D, ARC3D. Computational fluid dynamics, A workshop held at the University of Tennessee Space Institute, UTSE publ. E02-4005-023-84, 1984 {other related application programs parallelized included: SIMPLE and HYDRO-1}
30. Darema-Rogers F, Kirkpatrick S, Norton VA (1987) Parallel techniques for chip placement by simulated annealing. Proceedings of the International Conference on Computer-Aided Design. pp 91–94
31. Darema F, Kirkpatrick S, Norton VA (1987) Simulated annealing on shared memory parallel systems. IBM Journal of R&D 31:391–402
32. Jayaraman R, Darema F Error analysis of parallel simulated annealing techniques. Proceedings of the ICCD'88, Rye, NY, 10/3-5/88
33. Greening D, Darema F Rectangular spatial decomposition methods for parallel simulated annealing. IBM research report, RC14636, 5/2/89, and in the Proceedings of the International Conference on Supercomputing '89. Crete-Greece
34. Darema F (2009) Report on cyberinfrastructures of cyber-applications-systems & cyber-systems-software, submitted for external publication
35. DDDAS. [www.cise.nsf.gov/dddas](http://www.cise.nsf.gov/dddas)
36. DDDAS. [www.dddas.org](http://www.dddas.org)
37. Darema F (2005) Grid computing and beyond: the context of dynamic data driven applications systems. Proc IEEE (Special Issue on Grid Computing) 93(3):692–697
38. Darema F (2000) New software architecture for complex applications development and runtime support. Int J High-Perform Comput (Special Issue on Programming Environments, Clusters, and Computational Grids For Scientific Computing) 14(3)
39. GRADS project. <http://www.hipersoft.rice.edu/grads/>
40. Adve VS, Sanders WH A compiler-enabled model- and measurement-driven adaptation environment for dependability and performance. – <http://www.perform.csl.illinois.edu/projects/newNSFNGS.html>, LLVL compiler – <http://llvml.org>
- 
- ## SSE
- ▶ [AMD Opteron Processor Barcelona](#)
  - ▶ [Intel Core Microarchitecture, x86 Processor Family](#)
  - ▶ [Vector Extensions, Instruction-Set Architecture \(ISA\)](#)
- 
- ## Stalemate
- ▶ [Deadlocks](#)
- 
- ## State Space Search
- ▶ [Combinatorial Search](#)
- 
- ## Stream Processing
- ▶ [Stream Programming Languages](#)
- 
- ## Stream Programming Languages
- RYAN NEWTON  
Intel Corporation, Hudson, MA, USA
- ### Synonyms
- [Complex event processing](#); [Event stream processing](#); [Stream processing](#)

## Definition

Stream Programming Languages are specialized to the processing of data streams. A key feature of stream programming languages is that they structure programs not as a series of instructions, but as a graph of computation *kernels* that communicate via data streams. Stream programming languages are implicitly parallel and contain data, task, and pipeline parallelism. They offer some of the best examples of high-performance, fully automatic parallelization of implicitly parallel code.

## Discussion

### Overview

Stream processing is found in many areas of computer science. Accordingly, it has received distinct and somewhat divergent treatments. For example, most work on stream processing in the context of compilers, graphics, and computer architecture has focused on streams with predictable data-rates, such as in digital signal processing (DSP) applications. These can be modeled using *synchronous dataflow*, are deterministic (being a subclass of Kahn process networks), and are especially attractive as parallel programming models. Specialized languages for dataflow programming go at least as far back the SISAL (Streams and Iteration in a Single Assignment Language) in 1983. Somewhat different are *streaming databases* that are concerned with asynchronous (unpredictable) streams of timestamped events. Typical tasks include searching for temporal patterns and relating streams to each other and to data in stored tables, which may be termed “complex event processing.”

This entry considers a broad definition of stream processing, requiring only that kernel functions and data streams be the dominant means of computation and communication. This includes both highly restrictive models – statically known, fixed graph and data-rates – and less restrictive ones. More restrictions allow for better performance and more automatic parallelism, whereas less restrictive models handle a broader class of applications.

Even this broad definition has necessarily blurry boundaries. For example, while stream-processing models typically allow only pure *message passing* communication, they can be extended to allow various forms of shared memory between kernel functions. General purpose programming languages equipped

with libraries for stream processing are likely to fall into this category, as are streaming databases that include shared, modifiable tables.

### Example Programming Models

This entry classifies stream programming designs along two major axes: *Graph Construction Method* and *Messaging Discipline*. The following three fictional programming models will be used to illustrate the design space:

- **StreamStatic:** a language with a statically known, fixed graph of kernel functions, as well as statically known, fixed data-rates on edges.
- **StreamDynamic:** a general purpose language that can manipulate streams as first-class values, constructing and destroying graphs on the fly and allowing arbitrary access to shared memory.
- **StreamDB:** Kernel graphs that are changed dynamically through transactions. Includes shared memory in the form of modifiable tables. More restricted and with different functionality than StreamDynamic.

Snippets of code corresponding to these three languages appear in Figs. 1–3, respectively. Kernels, being nothing more than functions, are usually defined in a manner resembling function definitions. In StreamDynamic the same language is used for graph construction as for kernel definition, whereas StreamStatic employs a distinct notation for graph topologies. Only in StreamDB are the definitions of kernels nonobvious. Whenever a new stream is defined from an

```
metadata: pop 1 push 1
stream out AddAccum(stream in) {
 state { int s = 0; }
 execute() {
 s++;
 out.push(s + in.pop());
 }
}

// A static graph topology, in literal
// textual notation:
IntIn -> AddAccum -> IntOutput;
```

**Stream Programming Languages.** Fig. 1 A kernel function in StreamStatic might appear as in the above pseudocode, explicitly specifying the amount of data produced and consumed (pushed and popped) by the kernel during each invocation. This particular kernel carries a state that persists between invocations

```
-- Streams and tables coexist
CREATE TABLE Prices (
 Id string PRIMARY KEY,
 Price double);
CREATE INPUT STREAM PriceIncrs (
 Id string,
 Increase double);
-- A 'kernel' is constructed by building new streams from old
CREATE STREAM NormedPriceIncrs AS
 SELECT Id, Increase / 10.0
 FROM PriceIncrs;
-- Table modified based on streaming data:
UPDATE Prices USING NormedPriceIncrs
 SET Price = Prices.Price + NormedPriceIncrs.Increase,
 WHERE Prices.Id == NormedPriceIncrs.Id;
```

**Stream Programming Languages.** Fig. 2 A pseudocode example of StreamDB code

```
// A function that creates N copies of a kernel
fun pipeline(int n, Function fn, Stream S) {
 if (n==0) return S;
 else pipeline(n-1, fn, map(fn,S));
}

// Graph wiring is implicit in program execution:
pipeline(10, AddAccum, OrigStream)
```

**Stream Programming Languages.** Fig. 3 A pseudocode example of StreamDynamic code

old one with a `SELECT` statement, the expressions used in the `SELECT` statement form the body of a new kernel function. However, in streaming databases, much more functionality is built into the primitive operators, supporting, for example, a variety of windowing and grouping operations on streaming data.

**Messaging** StreamStatic assumes that each kernel specifies exactly how many inputs it consumes on each inbound edge, and outputs it produces on outbound edges. StreamDynamic instead uses asynchronous event streams and allows *nondeterministic merge*, which interleaves the elements of two streams in the real-time order in which they occur. StreamDB is similar to StreamDynamic, but timestamps all stream elements at their source, and then maintains a deterministic semantics with respect to those explicit timestamps.

### Methods for Graph Wiring

Our three fictional languages use different mechanisms for constructing graphs. StreamStatic uses a literal textual encoding of the graph (Fig. 1). StreamDB also uses a direct encoding of a graph in the text of a query,

but in the form of named streams with explicit dependencies (Fig. 2). In contrast, in StreamDynamic the graph is implicit, resulting from applying operators such as `map` to stream objects (Fig. 3).

There are other common graph construction methods not represented in these fictional languages: first, GUI-based construction of graphs, as found in LabView[12] and StreamBase[1]; second, an API for adding edges in a general purpose language, along the lines of “`connect(x,y);`”. Compared to implicit graph wiring via the manipulation of stream values (as seen in StreamDynamic or in real systems such as FlumeJava[2]), APIs of this kind are more explicit about edges and usually more verbose.

Finally, an advanced technique for constructing stream graphs is *metaprogramming*. Metaprogramming, or staged execution, simply refers to any program that generates another program. In the case of streaming, the most common situation is that a static kernel graph is desired, yet the programmer wishes to use loops and abstraction in the construction of (potentially complex) graphs. Therefore the need for

metaprogramming in streaming is analogous to hardware description languages, which also have a static target (logic gates) but need abstraction in the source language.

While it would be possible to write a program to generate the textual graph descriptions used by, for example, StreamStatic, a much more disciplined form of metaprogramming can be provided by stream-processing DSLs (domain specific languages), which can integrate the type-checking of the metalanguage and the target kernel language. Indeed, this is the approach taken by DSLs such as WaveScript[9] and to a lesser extent StreamIt[].

### Parallelism in Stream Programs

Once constructed, a kernel graph exposes *pipeline*, *task*, and *data* parallelism. This terminology, used by the StreamIt authors [4] among others, distinguishes between producer/consumer parallelism (pipeline) and kernels lacking that relationship, such as the siblings in a fork-join pattern (task parallelism). Data parallelism in this context refers specifically to the ability to execute a kernel simultaneously on multiple elements in a single stream. Data parallelism is not always possible with a stateful kernel function – equivalent to a loop-carried dependence.

Stream programs expose abundant parallelism. It is the job of a scheduler or program optimizer to manage that parallelism and map it onto hardware. Fully automatic solutions to this problem remain difficult, but less difficult in a restrictive stream programming model than many other programming models. The key advantage is that stream programs define independent kernels with local data access and explicit, predictable communication.

This advantage enables stream programming models like StreamStatic to target a wide range of hardware architectures, including traditional cache-based CPUs (both multicore and vector parallelism), as well as GPUs, and architectures with software-controlled communication, such as the RAW tiled architecture [11] or the Cell processor. For example, the DSL StreamIt[4] (which subsumes StreamStatic) is an example of a programming language that has targeted all these platforms as well as networked clusters of workstations.

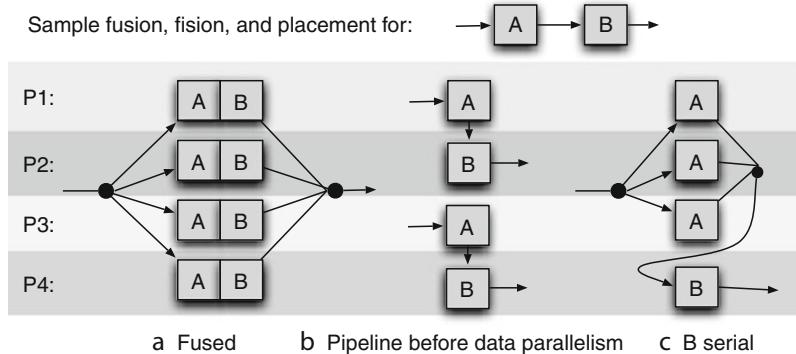
### Optimization and Scheduling

An effective stream-processing implementation must accomplish three things: adjust granularity, choose between sources of parallelism, and place computations. Granularity here refers to how much data is processed in bulk, as well as whether or not kernels are combined (fused). Second, because stream programs include multiple kinds of parallelism, it is necessary to balance pipeline/task parallelism and data parallelism. Finally, placement and ordering presents a variant of the traditional task-scheduling problem.

Consider the two-kernel pipeline shown in Fig. 4. Even the simple composition of two kernels exposes several trade-offs. Three possible placements of kernels A and B onto four processors are shown in the figure. Placement (a) illustrates a common scenario: fusing kernels and leveraging data parallelism rather than pipeline parallelism. Yet choosing pipeline parallelism (placement (b)) could be preferable if, for example, A and B's combined working sets exceed the cache capacity of a processor. (Note that in a steady state streams provide enough data to keep A and B simultaneously executing – software pipelining.) Finally, placement (c) illustrates another common scenario – a kernel carries state and cannot be parallelized. In this case, kernel B must be executed serially, so it must not be fused with A or it would serialize A.

Note that there is a complicated interplay between batch size, kernel working set, and fusion/placement. One simple model for kernel working set assumes its size is linear in the size of its input  $x$ : that is,  $mx + k$  where  $k$  represents memory that must be read irrespective of batch size. A large  $k$  might, for example, suggest the selection of placement (b) in Fig. 4 because if  $k = 0$  batch size could be tuned to make placement (a) attractive. But a phase-ordering problem arises: Solving either batch size or fusion/fission independently (rather than simultaneously) can sacrifice opportunities to achieve optimal placement.

A related complication is that while the choices made by a stream-processing system could take place either statically or dynamically, granularity adjustment is more difficult to perform dynamically. For example, while dynamic work-stealing on a shared-memory computer can provide load balance and place kernels onto processors, it cannot deal with fine-grained kernels



**Stream Programming Languages. Fig. 4** Sample placements of two kernels, A and B onto processors P1–P4

that perform only a few FLOPS per invocation. That said, as long as kernels are compiled to handle batches of data, the exact size of batches can be adjusted at runtime.

### Program Transformation

If optimizations are performed statically by a compiler, they can take the form of source-to-source transformations. The most important optimizations fall into the following four categories:

- High-level / Algebraic – domain - specific optimizations, for example, canceling an FFT and an inverse FFT (see Haskell [8], WaveScript [9])
- Batch-processing / Execution Scaling – choosing batch size
- Fusion – combining and inlining kernels
- Fission / Data Parallelism – choosing N-ways to split a kernel

StreamStatic, for example, could follow the example of StreamIT and perform fusion, fission, and scaling as source-to-source program transformations, resulting in a number of kernels that match the target number of processors and leaving only a one-to-one placement problem to be solved by a simulated annealing optimization process. To achieve load balance during this process, StreamIt happens to use static work-estimation for kernels, but profiling is another option. In fact, given more dynamic data-rates, as in StreamDynamic and StreamDB, profile-based optimization and auto-tuning techniques are a good way to enable some of the above

static program transformations. (WaveScript takes this approach [9].) To our knowledge no systems attempt to apply high-level / algebraic optimizations dynamically.

### Algorithms

Dynamic systems along the lines of StreamDynamic tend to use work-stealing, as do most of today's popular task-scheduling frameworks. Further, many streaming databases use heuristics for kernel/operator migration to achieve load balance in a distributed setting. On the other hand, streaming models with fixed data-rates permit static scheduling strategies and much work has focused there. Scheduling predictable stream programs bears some resemblance to the well-studied problem of mapping a coarse-grained task-graph onto a multiprocessor (surveyed in [6]). But there is a notable difference: stream programs run continuously and are usually scheduled for throughput (with some exceptions [10]). Likewise, traditional graph-partitioning methods can be applied, but they do not, in general, capture all the degrees of freedom present in stream programs, whose graphs are not fixed, but are subject to transformations such as data-parallel fission of kernels.

An overview of scheduling work on stream processing specifically can be found in [13], focusing on embedded and real-time uses. The ideal would be an algorithm that simultaneously optimizes all aspects of a streaming program. A recent paper [5] proposed one such system that uses integer linear programming to simultaneously perform fission and processor placement for StreamIt programs.

## Data Structures and Synchronization

Static and dynamic scheduling approaches employ different data structures. Dynamic approaches targeting shared memory multiprocessors are likely to rely on concurrent data structures including queues. In contrast, a completely static schedule typically requires no synchronization on data structures. Rather it may repeat a fixed schedule, using a global barrier synchronization at the end of each iteration of the schedule.

## Relation to Other Parallel Programming Models

### Functional Reactive Programming

Like StreamDynamic, FRP enables general purpose functional programs (usually in Haskell) to manipulate streams of values. FRP in particular deals not with discrete streams of elements, but with semantically continuous *signals*. Sampling of signals is performed by the runtime, or separately from the main definition of programs. Most FRP implementations are not high-performance, but parallel implementations and staged implementations that generate efficient code exist.

### Relation to Data-parallel Models

With the increasing popularity of the *MapReduce* paradigm, there is a lot of interest in programming models for massively parallel manipulation of data. The FlumeJava library [2] (built on Google's MapReduce) bears a lot of resemblance to a stream-processing system. The user composes parallel operations on collections, and FlumeJava applies fusion transformations to pipelines of parallel operations. Similar systems like Cascade and Dryad explicitly construct dataflow graphs. These systems have a different application focus, target hardware, and scale than most of the work on stream processing. Further, they focus on batch processing of data in parallel, not on continuous execution.

### Relation to Fork-Join Shared-Memory Parallelism

Fork-join parallelism in the form of parallel subroutine calls and parallel loops, as found in Cilk [7] and OpenMP [3], are attractive because of their relative ease of incorporation into legacy codes. In contrast,

stream processing requires that a program be factored into kernels and that communication become explicit. Once a program is ported, however, it is perhaps easier to assure that it is correct and deterministic. Stream-processing languages provide a complete programming model encompassing computation and communication. In contrast, fork-join models treat the issues of control-flow and work-decomposition, but do not directly address data decomposition or communication.

## Future Directions

This entry has described a family of programming models that have already accomplished a lot. Unfortunately, the systems described above have seen little application to industrial stream-processing problems. Most stream-processing codes are written in general purpose languages without any special support for stream processing. A major future challenge is to increase adoption of the body of techniques developed for stream processing.

Libraries, rather than stream-programming languages may have an advantage in this respect. Further, stream programming systems that are *wide spectrum* may prove desirable in the future – models that can reproduce the best results of more restrictive programming models, while also allowing graceful degradation into more general programming, therefore not confining the user strictly.

## Related Entries

- ▶ [Cell Processor](#)

## Bibliography

1. <http://www.streambase.com/>
2. Chambers C, Raniwala A, Perry F, Adams S, Henry RR, Bradshaw R, Weizenbaum N (2010) Flumejava: easy, efficient data-parallel pipelines. In: PLDI '10: proceedings of the 2010 ACM SIGPLAN conference on programming language design and implementation. ACM, New York, pp 363–375
3. Dagum L, Menon R (1998) OpenMP: an industry standard API for shared memory programming. IEEE Comp Sci Eng 5(1):46–55
4. Gordon MI, Thies W, Amarasinghe S (2006) Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. SIGOPS Oper Syst Rev 40(5):151–162
5. Kudlur M, Mahlke S (2008) Orchestrating the execution of stream programs on multicore platforms. In: PLDI '08 proceedings of

- the 2008 ACM SIGPLAN conference on programming language design and implementation. ACM, New York, pp 114–124
6. Kwok Y-K, Ahmad I (1999) Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput Surv* 31(4):406–471
  7. Leiserson CE (2009) The *cilk ++* concurrency platform. In: DAC '09 proceedings of the 46th annual design automation conference. ACM, New York, pp 522–527
  8. Liu H, Cheng E, Hudak P (2009) Causal commutative arrows and their optimization. In: ICFP '09 proceedings of the 14th ACM SIGPLAN international conference on functional programming. ACM, New York, pp 35–46
  9. Newton RR, Girod LD, Craig MB, Madden SR, Morrisett JG (2008) Design and evaluation of a compiler for embedded stream programs. In: LCTES '08 proceedings of the 2008 ACM SIGPLAN-SIGBED conference on languages, compilers, and tools for embedded systems. ACM, New York, pp 131–140
  10. Pillai PS, Mummert LS, Schlosser SW, Sukthankar R, Helfrich CJ (2009) Slipstream: scalable low-latency interactive perception on streaming data. In: NOSSDAV '09 proceedings of the 18th international workshop on network and operating systems support for digital audio and video. ACM, New York, pp 43–48
  11. Taylor MB, Lee W, Miller J, Wentzlaff D, Bratt I, Greenwald B, Hoffmann H, Johnson P, Kim J, Psota J, Saraf A, Shnidman N, Strumpen V, Frank M, Amarasinghe S, Agarwal A (2004) Evaluation of the raw microprocessor: an exposed-wire-delay architecture for ilp and streams. In: ISCA '04 proceedings of the 31st annual international symposium on Computer architecture. IEEE Computer Society, Washington, DC, p 2
  12. Travis J, Kring J (2006) LabVIEW for everyone: graphical programming made easy and fun, 3rd edn. Prentice Hall, Upper Saddle River
  13. Wiggers MH (2009) Aperiodic multiprocessor scheduling for real-time stream processing applications. Ph.D. thesis, University of Twente, Enschede, The Netherlands

## Strong Scaling

► [Amdahl's Law](#)

## Suffix Trees

AMOL GHOTING, KONSTANTIN MAKARYCHEV  
IBM Thomas. J. Watson Research Center, Yorktown Heights, NY, USA

## Synonyms

[Position tree](#)

## Definition

The suffix tree is a data structure that stores all the suffixes of a given string in a compact tree-based structure. Its design allows for a particularly fast implementation of many important string operations.

## Discussion

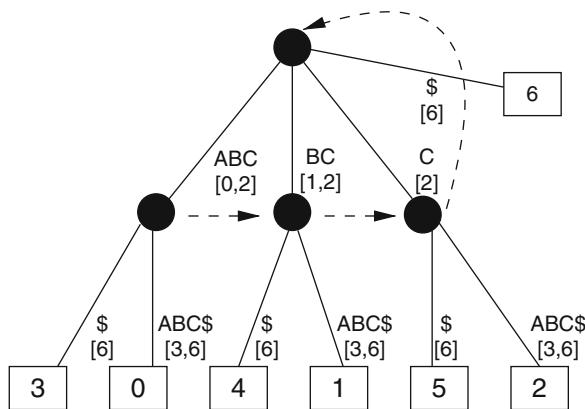
### Introduction

The suffix tree is a fundamental data structure in string processing. It exposes the internal structure of a string in a way that facilitates the efficient implementation of a myriad of string operations. Examples of these operations include string matching (both exact and approximate), exact set matching, all-pairs suffix-prefix matching, finding repetitive structures, and finding the longest common substring across multiple strings [12].

Let  $A$  denote a set of characters. Let  $S = s_0, s_1, \dots, s_{n-1}, \$$ , where  $s_i \in A$  and  $\$ \notin A$ , denote a \$ terminated input string of length  $n + 1$ . The  $i$ th suffix of  $S$  is the substring  $s_i, s_{i+1}, \dots, s_{n-1}, \$$ . The suffix tree for  $S$ , denoted as  $T$ , stores all the suffixes of  $S$  in a tree structure. The tree has the following properties:

1. Paths from the root node to the leaf nodes have a one-to-one relationship with the suffixes of  $S$ . The terminal character  $\$$  is unique and ensures that no suffix is a proper prefix of any other suffix. Therefore, there are as many leaf nodes as there are suffixes.
2. Edges spell nonempty strings.
3. All internal nodes, except the root node, have at least two children. The edge for each child node begins with a character that is different from the starting character of its sibling nodes.
4. For an internal node  $v$ , let  $l(v)$  denote the substring obtained by traversing the path from the root node to  $v$ . For every internal node  $v$ , with  $l(v) = x\alpha$ , where  $x \in A$  and  $\alpha \in A^*$ , we have a pointer known as a suffix link to an internal node  $u$  such that  $l(u) = \alpha$ .

An instance of a suffix tree for a string  $S = ABCABC\$$  is presented in Fig. 1. Each edge in a suffix tree is represented using the start and end indices of the corresponding substring in  $S$ . Therefore, even though a suffix tree represents  $n$  suffixes (each with at most  $n$  characters) for a total of  $\Omega(n^2)$  characters, it only requires  $O(n)$  space [31].



**Suffix Trees. Fig. 1** Suffix tree for  $S = ABCABC\$$  [0123456].

Internal nodes are represented using circles and leaf nodes are represented using rectangles. Each leaf node is labeled with the index of the suffix it represents. The *dashed arrows* represent the suffix links. Each edge is labeled with the substring it represents and its corresponding edge encoding

## Applications

Over the past few decades, the suffix tree has been used for a spectrum of tasks ranging from data clustering [33] to data compression [3]. The quintessential usage of suffix trees is seen in the bioinformatics domain [2, 5, 6, 12, 17, 19, 24] where it is used to effectively evaluate queries on biological sequence data sets. A hallmark of suffix trees is that in many cases it allows one to process queries in time proportional to the size of the query rather than the size of string.

## Suffix Tree Construction

### Serial Suffix Tree Construction Algorithms

Algorithms due to Weiner [32], McCreight [22], and Ukkonen [31] have shown that suffix trees can be built in linear space and time. These algorithms afford a linear time construction by employing suffix links. Ukkonen's algorithm is more recent and popular because it is easier to implement than the other algorithms. It is an  $O(n)$ , in-memory construction algorithm. The algorithm is based on the simple, but elegant observation that the suffixes of a string  $S_i = s_0, s_1, \dots, s_i$  can be obtained from the suffixes of string  $S_{i-1} = s_0, s_1, \dots, s_{i-1}$  by concatenating symbol  $s_i$  at the end of each suffix of  $S_{i-1}$  and by adding the empty suffix. The suffixes of the whole

string  $S = S_n = s_0, s_1, \dots, s_n$  can then be obtained by first expanding the suffixes of  $S_0$  into the suffixes of  $S_1$  and so on, until the suffixes of  $S_n$  are obtained from the suffixes of  $S_{n-1}$ . This translates into a suffix tree construction that can be performed by iteratively expanding the leaves of a partially constructed suffix tree. Through the use of suffix links, which provide a mechanism for quickly locating suffixes, the suffix tree can be expanded by simply adding the  $(i+1)$ th character to the leaves of the suffix tree built on the previous  $i$  characters. The algorithm thus relies on suffix links to traverse through all of the sub-trees in the main tree, expanding the outer edges for each input character.

### Parallel Suffix Tree Construction

Parallel suffix tree construction has been extensively studied in theoretical computer science. Apostolico et al. [1], Sühleyman et al. [28], and Hariharan [14] proposed theoretically efficient parallel algorithms for the problem based on the PRAM model. For example, Hariharan [14] showed how to execute McCreight's [22] algorithm concurrently on many processors. These algorithms, however, are designed for the case when the string and the suffix tree fit in main memory. Since the memory accesses of these algorithms exhibit poor locality of reference [8], these algorithms are inefficient when either the string or the suffix tree do not fit in main memory. On many real-world data sets, this is often the case.

The aforementioned problem has been theoretically resolved by Farach-Colton et al. [8]. Farach-Colton et al. [8] proposed a divide-and-conquer algorithm that operates as follows. First, construct a suffix tree for suffixes starting at odd positions. To do so, sort all pairs of characters at positions  $2i-1$  and  $2i$  and replace each pair with its index in the sorted list. This gives us a string of length  $[n/2]$  with characters in a bigger alphabet. Recursively construct the suffix tree for this string. The obtained tree is essentially the suffix tree for suffixes starting at odd positions in the original string. Then, given the “odd” suffix tree construct an “even” suffix tree of suffixes starting at even positions. Finally, merge the trees. The details of the Farach-Colton et al. [8] algorithms are very complex, and there have not been any successful implementations. However, the doubling approach has been successfully used in practice for constructing suffix arrays

(see [4, 7], and section Suffix Arrays). While the authors do not explicitly provide a parallel algorithm, they indicate that the sort and merge phases do lend themselves to a parallel implementation. While the aforementioned algorithms provide theoretically optimal performance, for most algorithms, memory accesses exhibit poor locality of reference. As a consequence, these algorithms are grossly inefficient when either the tree or the string does not fit in main memory. To tackle this problem, this past decade has seen several research efforts that target large suffix tree construction. Algorithms that have been developed in these efforts can be placed in two categories: ones that require the input string to fit in main memory and ones that do not have this requirement. Many of these efforts only provide and evaluate serial algorithms for suffix tree construction. However, by design, as will be discussed, they do indeed lend themselves to a parallel implementation.

### Practical Parallel Algorithms for In-Core Strings

Hunt et al. [13] presented the very first approach to efficiently build suffix trees that do not fit in main memory. The approach drops the use of suffix links in favor of better locality of reference. Typically, the suffix tree is an order of magnitude larger than the string being indexed. As a result, for large input strings, the suffix tree cannot be accommodated in main memory. The method first finds a set of prefixes so as to partition the suffix tree into sub-trees (each prefix corresponds to a sub-tree) that can be built in main memory. The number of times a prefix occurs in a string is used to bound the size of the suffix sub-tree. The approach iteratively increases the size of the prefix until a length is reached where all the suffix sub-trees will fit in memory. Next, for each of these prefixes, the approach builds the associated suffix sub-tree using a scan of the data set. Essentially, for each suffix with the prefix, during insertion, one finds a path in the partially constructed suffix sub-tree that shares the longest common prefix (lcp) with this suffix, and branches from this path when no more matching characters are found. The worst case complexity of the approach is  $O(n^2)$ , but it exhibits  $O(n \log n)$  average case complexity. This algorithm can be parallelized by distributing the prefixes to the different processors in a round-robin fashion and having each processor build one or more suffix sub-trees of the

suffix tree. Kalyanaraman et al. [18] presented a parallel generalized suffix tree construction algorithm that in some ways builds upon Hunt's approach to realize a scalable parallel suffix tree construction. A generalized suffix tree is a suffix tree for not one but a group of strings. While the algorithm was presented in the context of the genome assembly problem, the proposed approach is not tied to genome assembly and can be applied elsewhere. The approach first sorts all suffixes based on their  $w$ -length prefixes, where (like Hunt et al.'s approach)  $w$  is picked to ensure that the associated suffix sub-trees will fit in main memory. Suffixes with the same prefix are then assigned to the a single bucket. The buckets are then partitioned across the processors such that load is approximately balanced. Each processor then builds a sub-tree of the final suffix tree using a depth-first tree building approach. This is accomplished by sorting all the suffixes in the local bucket and then inserting them in sorted order. The end result in a distributed representation of the generalized suffix tree as a collection of sub-trees.

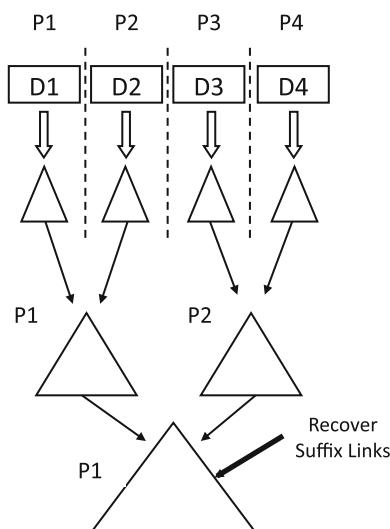
Japp introduced top-compressed suffix trees [15] that improves upon Hunt's approach by introducing a pre-processing stage to remove the need for repeated scans of the input sequence. Moreover, the author employed partitioning and optimized the insertion of suffixes into partitions using suffix links. The method is based on a new linear time construction algorithm for "sparse" suffix trees, which are sub-trees of the whole suffix tree. The new data structures are called the paged suffix tree (PST) and the distributed suffix tree (DST), respectively. Both tackle the memory bottleneck by constructing sub-trees of the full suffix tree independently and are designed for single processor and distributed memory parallel computing environments, respectively. The standard operations on suffix trees of biological importance are shown to be easily translatable to these new data structures.

S

### Practical Parallel Algorithms for Out-of-Core Strings

The above mentioned parallel algorithms scale well when the input string fits in main memory. Many real-world input strings (like the human genome), however, do not fit in main memory. Researchers have developed a variety of algorithms to handle this situation. The earlier solutions for this problem follow what is known as

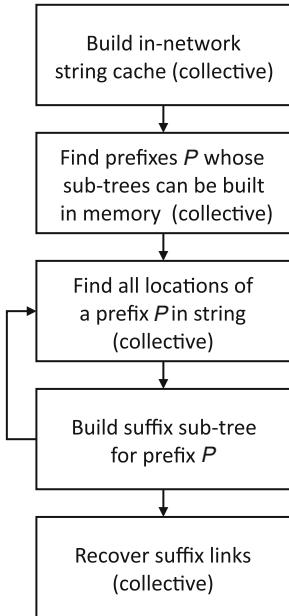
the partition-and-merge methodology. The approach is illustrated in Fig. 2. ST-MERGE [30], proposed by Tian et al. partitions the input string and constructs a suffix tree for each of these partitions in main memory. These suffix trees are then merged to create the final suffix tree. Merging two suffix trees involves finding matching suffixes in the two trees that share the longest common prefix, reusing this path in the new merged tree, and splitting this path when no more matching characters are found. Suffix link recovery can then be accomplished using a postprocessing step. The approach can be parallelized as follows. During the partition phase, each processor can be assigned block of the input string and it can build a suffix tree for the partition. During the merge phase, a processor can be assigned a pair of suffix trees where it is responsible for merging a pair of trees. One can realize a binary merge tree to build the final suffix tree. TRELLIS [26], due to Phoophakdee and Zaki, is similar in flavor but differs in the following regards. First, the approach finds a set of variable-length prefixes such that the corresponding suffix sub-trees will fit in main memory. Second, it partitions the input string and constructs a suffix tree for each partition in main memory (like ST-MERGE) and stores the sub-trees for each prefix determined in the first step, separately, on disk. Finally, it merges all the sub-trees associated with each prefix to realize the final set of suffix sub-trees. By



**Suffix Trees.** Fig. 2 Illustration of the partition and merge approach

design, TRELLIS ensures that each of the suffix sub-trees (built using the merge operation) will fit in main memory. TRELLIS can be parallelized using an approach similar to the parallelization of ST-MERGE.

Ghoting and Makarychev [10] studied the performance of the “partition-and-merge” approach for very large input strings and showed that the working set for these algorithms scales linearly with the size of the input string. This results in a lot of *random disk I/O* during the merge phase (the authors of TRELLIS do mention this as well [26]) when indexing strings that do not fit in main memory. One can even argue that for very large strings, the performance of the “partition-and-merge” way will converge to that of Hunt’s approach [13], as merging sub-trees in tantamount to inserting suffixes into a partially constructed suffix tree. To address this challenge, Ghoting and Makarychev presented a serial algorithm WAVEFRONT and its parallelization P-WAVEFRONT [10, 11]. P-WAVEFRONT is the first parallel algorithm that can build a suffix tree for an input string that does not fit in main memory. P-WAVEFRONT diverges from the partition-and-merge methodology to suffix tree construction. Leveraging the structure of suffix trees, the algorithm builds a suffix tree by simultaneously tiling accesses to both the input string and the partially constructed suffix tree. Steps for P-WAVEFRONT are presented in Fig. 3. First, an in-network string cache is built by distributing the input strings across all processors in a round-robin fashion. The algorithm assumes that the input string can fit in collective main memory. This step uses collective I/O to ensure efficient reading of the input string. Next, like TRELLIS, a set of variable length prefixes are found in parallel such that the associated suffix sub-trees fit in memory. These prefixes are then distributed across all processors in a round-robin fashion. The following prefix location discovery phase is used to find the location of each prefix being processed using a collective procedure. During this step, each processor is responsible for finding locations of all prefixes (not just its own) in a partition of the input string and these are collectively exchanged with other processors such that each processor has locations of its prefix in the entire input string. Finally, the suffix sub-tree for each prefix is built in a tiled and iterative manner by processing a pair of blocks of the input string at a time. The end result is an algorithm that can index very large input strings and at the same time maintain a



**Suffix Trees.** Fig. 3 Steps of P-wavefront

bounded working set size and a fixed memory footprint. The proposed methodology was applied to the suffix link recovery process as well, realizing an end-to-end I/O-efficient solution.

### Suffix Arrays

Closely related to suffix trees is the suffix array [23]. Suffix array is the alphabetically sorted list of all suffixes of a given string  $S$ . For example, the suffix array of the string  $S = ABCABC\$\$$  is as follows

| Index | Suffix   | lcp |
|-------|----------|-----|
| 6     | \$       | 0   |
| 3     | ABC\$    | 0   |
| 0     | ABCABC\$ | 3   |
| 4     | BC\$     | 0   |
| 1     | BCABC\$  | 2   |
| 5     | C\$      | 0   |
| 2     | CABC\$   | 1   |

The suffix array only contains pointers to the suffixes in the original string  $S$ . Thus, in the example above,

the suffix array is the first column of the table, that is, the array  $\{6, 3, 0, 4, 1, 5, 2\}$ . Given a suffix tree one can obtain a suffix array in linear time by performing a depth-first search (DFS) (see Fig. 1). Suffix arrays are often used instead of suffix trees. The main advantage of suffix arrays is that they have a more compact representation in the memory. Besides the pointers to the suffixes, suffix arrays can also be augmented to contain lengths of the longest common prefixes (lcp) between adjacent strings (see the third column in the example above). In which case, given a suffix array it is easy to reconstruct the suffix tree. Essentially, the longest common prefix corresponds to the lowest common ancestor (lca) in the tree. Relative to suffix trees, the main drawback is that query processing using suffix arrays can be more time consuming as most queries are processed in time that is function of the size of the input string and not the size of the query (as was the case with suffix trees). Researchers have developed several linear time algorithms for directly building suffix arrays without pre-building a suffix tree [16, 21].

Futamura et al. [9] presented the first algorithm for parallel suffix array construction. The approach is similar to the one by Kalyanaraman et al. [18] that was described in section Practical Parallel Algorithms for In-Core Strings. The approach first partitions the suffixes into buckets based on their  $w$ -length prefixes. These buckets are then distributed across the processors, where the suffixes in each bucket are sorted locally to obtain a portion of the suffix array. The final suffix array can then be realized by concatenating these distributed suffix arrays.

Algorithms for constructing suffix arrays in external memory have been extensively studied and compared by Crauser and Ferragina [4] and by Dementiev et al. [7]. Kärkkäinen et al. [21] proposed the DC3 algorithm, which is based on a similar approach as the Farach et al. [8] divide-and-conquer algorithm for suffix trees (see section Parallel Suffix Tree Construction). However, instead of dividing suffixes in the input string into suffixes starting at odd and even positions, it divides them in three groups depending on the remainder of the suffix position modulo 3. Surprisingly, this change significantly simplifies the algorithm. A variant of DC3, called pDC3, has been implemented by Kulla and Sanders [20]. Their study as well as the study of Dementiev et al. [7] indicate that DC3 and pDC3 are currently

the most efficient algorithms for constructing suffix arrays.

## Related Entries

- Bioinformatics
- Genome Assembly

## Bibliographic Notes and Further Reading

Research on parallel indexing for sequence/string data sets is still in its infancy. The primary reason for this is that until 2005, sequence data sets were not growing at a rapid pace and most problems could be handled in main memory. However, over the past few years, with advances in sequencing technologies, sequence databases have reached gigantic proportions. For instance, aggressive DNA sequencing efforts have resulted in the GenBank sequence database surpassing the 100 Gbp (one bp (base pair) is one character in the sequence) mark [25], with sequences from over 165,000 organisms. Further complicating the issue is the fact that researchers not only need the ability to index a single large genome, but a group of large genomes. For example, consider the area of comparative genomics [27] where one is interested in comparing different genomes, be they from the same or different species. Here researchers may be interested in comparing the genomes of individuals that are prone to a specific type of cancer to those that are not susceptible. In this case, we need to efficiently build a suffix tree for a group of large genomes, as and when needed. These trends suggest the parallel indexing technology for sequence data sets will be extremely important in the coming decade.

Much of this entry focused on parallel construction of suffix trees and suffix arrays. We would like to point the reader to Dan Gusfield's book [12] for a larger overview of applications. There has not been much work on parallelization of query processing using suffix trees and suffix arrays.

Another important direction is the design of parallel cache-conscious algorithms for multi-core processors so as to effectively utilize the memory hierarchy on modern processors. Tsirogiannis and Koudas looked at the problem of parallelizing the partition-and-merge approach on chip-multiprocessor architectures [29] and developed a cache-conscious algorithm for suffix tree

construction. Such directions will become increasingly important in the near future given the tendency of packing many cores on a single processor.

## Bibliography

1. Apostolico A, Iliopoulos C, Landau G, Schieber B, Vishkin U (1988) Parallel construction of a suffix tree with applications. *Algorithmica* 3(1-4):347–365
2. Bray N, Dubchak I, Pachter L (2003) AVID: a global alignment program. *Genome research* 13(1):97–102
3. Burrows M, Wheeler D (1994) A block sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation. Palo Alto, California
4. Crauser A, Ferragina P (2008) A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica* 52(1):1–35
5. Delcher A, Kasif S, Fleischmann R, Peterson J, White O, Salzberg S (1999) Alignment of whole genomes. *Nucleic Acids Res* 27(11):2369–2376
6. Delcher A, Phillippy A, Carlton J, Salzberg S (2002) Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Res* 30(1)
7. Dementiev R, Kärkkäinen J, Mehnert J, Sanders P (2008) Better external memory suffix array construction. *J Exp Algorithms* (JEA) 12:3–4
8. Farach-Colton M, Ferragina P, Muthukrishnan S (2000) On the sorting-complexity of suffix tree construction. *J ACM* 47(6): 987–1011
9. Futamura N, Aluru S, Kurtz S (2001) Parallel suffix sorting. In: Proceedings 9th international conference on advanced computing and communications. Citeseer, pp 76–81
10. Ghosh A, Makarychev K (2009) Indexing genomic sequences on the IBM Blue Gene. In: SC '09: proceedings of the conference on high performance computing networking, storage and analysis. ACM, New York, pp 1–11
11. Ghosh A, Makarychev K (2009) Serial and parallel methods for I/O efficient suffix tree construction. In: Proceedings of the ACM international conference on management of data. ACM, New York
12. Gusfield D (1997) Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge University Press, Cambridge
13. Hariharan R (1994) Optimal parallel suffix tree construction. In: Proceedings of the symposium on theory of computing. ACM, New York
14. Hunt E, Atkinson M, Irving R (2001) A database index to large biological sequences. In: Proceedings of 27th international conference on very large databases. Morgan Kaufmann, San Francisco
15. Japp R (2004) The top-compressed suffix tree: a disk resident index for large sequences. In: Proceedings of the bioinformatics workshop at the 21st annual british national conference on databases

16. Kalyanaraman A, Emrich S, Schnable P, Aluru S (2007) Assembling genomes on largescale parallel computers. *J Parallel Distrib Comput* 67(12):1240–1255
17. Kärkkäinen J, Sanders P, Burkhardt S (2006) Linear work suffix array construction. *J ACM* 53(6):918–936
18. Ko P, Aluru S (2005) Space efficient linear time construction of suffix arrays. *J Discret Algorithms* 3(2–4):143–156
19. Kulla F, Sanders P (2006) Scalable parallel suffix array construction. In: Recent advances in parallel virtual machine and message passing interface: 13th European PVM/MPI User's Group Meeting, Bonn, Germany, 17–20 September, 2006: proceedings. Springer, New York, p 22
20. Kurtz S, Choudhuri J, Ohlebusch E, Schleiermacher C, Stoye J, Giegerich R (2001) Reputer: the manifold applications of repeat analysis on a genome scale. *Nucleic Acids Res* 29(22):4633–4642
21. Kurtz S, Phillippy A, Delcher A, Smoot M, Shumway M, Antonescu C, Salzberg S (2004) Versatile and open software for comparing large genomes. *Genome Bio* 5(R12)
22. Manber U, Myers G (1990) Suffix arrays: a new method for on-line string searches. In: Proceedings of the first annual ACM-SIAM symposium on discrete algorithms. Society for Industrial and Applied Mathematics, Philadelphia, pp 319–327
23. McCreight E (1976) A space-economical suffix tree construction algorithm. *J ACM* 23(2)
24. Meek C, Patel J, Kasety S (2003) Oasis: an online and accurate technique for localalignment searches on biological sequences. In: Proceedings of 29th international conference on very large databases
25. NCBI. Public collections of DNA and RNA sequence reach 100 gigabases, 2005. [http://www.nlm.nih.gov/news/press\\_releases/dna\\_rna\\_100\\_gig.html](http://www.nlm.nih.gov/news/press_releases/dna_rna_100_gig.html).
26. Phoophakdee B, Zaki M (2007) Genome-scale disk-based suffix tree indexing. In: Proceedings of the ACM international conference on management of data. ACM, New York
27. Rubin GM, Yandell MD, Wortman JR, Gabor Miklos GL, Nelson CR, Hariharan IK, Fortini ME, Li PW, Apweiler R, Fleischmann W, Cherry JM, Henikoff S, Skupski MP, Misra S, Ashburner M, Birney E, Boguski MS, Brody T, Brokstein P, Celniker SE, Chervitz SA, Coates D, Cravchik A, Gabrilian A, Galle RF, Gelbart WM, George RA, Goldstein LS, Gong F, Guan P, Harris NL, Hay BA, Hoskins RA, Li J, Li Z, Hynes RO, Jones SJ, Kuehl PM, Lemaitre B, Littleton JT, Morrison DK, Mungall C, O'Farrell PH, Pickeral OK, Shue C, Vosshall LB, Zhang J, Zhao Q, Zheng XH, Zhong F, Zhong W, Gibbs R, Venter JC, Adams MD, Lewis S (2000) Comparative genomics of the eukaryotes. *Science* 287(5461):2204–2215
28. Sahinalp SC, Vishkin U (1994) Symmetry breaking for suffix tree construction. In: STOC '94: proceedings of the twenty-sixth annual ACM symposium on Theory of computing ACM, New York, pp 300–309
29. Tian Y, Tata S, Hankins R, Patel J (2005) Practical methods for constructing suffix trees. *VLDB J* 14(3):281–299
30. Tsirougiannis D, Koudas N (2010) Suffix tree construction algorithms on modern hardware. In: EDBT '10: Proceedings of the 13th international conference on extending database Technology. ACM, New York, pp 263–274
31. Ukkonen E (1992) Constructing suffix trees on-line in linear time. In: Proceedings of the IFIP 12th work computer congress on algorithms, software, architecture: information processing. North Holland Publishing Co., Amsterdam
32. Weiner P (1973) Linear pattern matching algorithms. In: Proceedings of 14th annual symposium on switch and automata theory. IEEE Computer Society, Washington, DC
33. Zamir O, Etzioni O (1998) Web document clustering: a feasibility demonstration. In: Proceedings of 21st international conference on research and development in information retrieval. ACM, New York

## Superlinear Speedup

### ► Metrics

## SuperLU

XIAOYE SHERRY LI<sup>1</sup>, JAMES DEMMEL<sup>2</sup>, JOHN GILBERT<sup>3</sup>, LAURA GRIGORI<sup>4</sup>, MEIYUE SHAO<sup>5</sup>

<sup>1</sup>Lawarence Berkeley National Laboratory, Berkeley, CA, USA

<sup>2</sup>University of California at Berkeley, Berkeley, CA, USA

<sup>3</sup>University of California, Santa Barbara, CA, USA

<sup>4</sup>Laboratoire de Recherche en Informatique Universite Paris-Sud 11, Paris, France

<sup>5</sup>Umeå University, Umeå, Sweden

## Synonyms

Sparse gaussian elimination

## Definition

SuperLU is a general-purpose library for the solution of large, sparse, nonsymmetric systems of linear equations using direct methods. The routines perform LU decomposition with numerical pivoting and solve the triangular systems through forward and back substitution. Iterative refinement routines are provided for improved backward stability. Routines are also provided to equilibrate the system, to reorder the columns to preserve sparsity of the factored matrices, to estimate the condition number, to calculate the relative backward error, to estimate error bounds for the refined solutions,

and to perform threshold-based incomplete LU factorization (ILU), which can be used as a preconditioner for iterative solvers. The algorithms are carefully designed and implemented so that they achieve excellent performance on modern high-performance machines, including shared-memory and distributed-memory multiprocessors.

## Discussion

### Introduction

SuperLU consists of a collection of three related ANSI C subroutine libraries for solving sparse linear systems of equations  $AX = B$ . Here  $A$  is a square, nonsingular,  $n \times n$  sparse matrix, and  $X$  and  $B$  are dense  $n \times nrhs$  matrices, where  $nrhs$  is the number of right-hand sides and solution vectors. The LU factorization routines can handle non-square matrices. Matrix  $A$  need not be symmetric or definite; indeed, SuperLU is particularly appropriate for matrices with very unsymmetric structure. All three libraries use variations of Gaussian elimination optimized to take advantage of both sparsity and the computer architecture, in particular memory hierarchies (caches) and parallelism [5, 17]. All three libraries can be obtained from the following URL:

<http://crd.lbl.gov/~xiaoye/SuperLU/>

The three libraries within SuperLU are as follows:

- *Sequential SuperLU* (SuperLU) is designed for sequential processors with one or more levels of caches [3]. Routines for both complete and threshold-based incomplete LU factorizations are provided [20].
- *Multithreaded SuperLU* (SuperLU\_MT) is designed for shared-memory multiprocessors, such as multicore, and can effectively use 16–32 parallel processors on sufficiently large matrices in order to speed up the computation [4].
- *Distributed SuperLU* (SuperLU\_DIST) is designed for distributed-memory parallel machines, using MPI for interprocess communication. It can effectively use hundreds of parallel processors on sufficiently large matrices [19].

Table 1 summarizes the current status of the software. All the routines are implemented in C, with parallel extensions using Pthreads or OpenMP for

**SuperLU. Table 1** SuperLU software status

|                                         | Sequential<br>SuperLU                 | SuperLU_<br>MT                        | SuperLU_<br>DIST           |
|-----------------------------------------|---------------------------------------|---------------------------------------|----------------------------|
| Platform                                | Serial                                | Shared-<br>memory                     | distributed-<br>memory     |
| Language<br>(with Fortran<br>interface) | C                                     | C + Pthreads<br>or OpenMP             | C + MPI                    |
| Data type                               | Real/<br>complex<br>Single/<br>double | Real/<br>complex<br>Single/<br>double | Real/<br>complex<br>Double |

shared-memory programming, or MPI for distributed-memory programming. Fortran interfaces are provided in all three libraries.

### Overall Algorithm

The kernel algorithm in SuperLU is sparse Gaussian elimination, which can be summarized as follows:

1. Compute a *triangular factorization*  $P_r D_r A D_c P_c = LU$ . Here  $D_r$  and  $D_c$  are diagonal matrices to equilibrate the system, and  $P_r$  and  $P_c$  are *permutation matrices*. Premultiplying  $A$  by  $P_r$  reorders the rows of  $A$ , and postmultiplying  $A$  by  $P_c$  reorders the columns of  $A$ .  $P_r$  and  $P_c$  are chosen to enhance sparsity, numerical stability, and parallelism.  $L$  is a unit lower triangular matrix ( $L_{ii} = 1$ ) and  $U$  is an upper triangular matrix. The factorization can also be applied to non-square matrices.
2. Solve  $AX=B$  by evaluating  $X=A^{-1}B=(D_r^{-1}P_r^{-1}LUP_c^{-1}D_c^{-1})^{-1}B=D_c(P_c(U^{-1}(L^{-1}(P_r(D_rB)))))$ . This is done efficiently by multiplying from right to left in the last expression: Scale the rows of  $B$  by  $D_r$ . Multiplying  $P_r B$  means permuting the rows of  $D_r B$ . Multiplying  $L^{-1}(P_r D_r B)$  means solving  $nrhs$  triangular systems of equations with matrix  $L$  by substitution. Similarly, multiplying  $U^{-1}(L^{-1}(P_r D_r B))$  means solving triangular systems with  $U$ .

In addition to exact, or complete, factorization, SuperLU also contains routines to perform incomplete factorization (ILU), in which sparser and so cheaper approximations to  $L$  and  $U$  are computed, which can be

used as a general-purpose preconditioner in an iterative solver.

The simplest implementation, used by the *simple driver* routines in SuperLU and SuperLU\_MT, consists of the following steps:

### Simple Driver (assumes $D_r = D_c = I$ )

1. Choose  $P_c$  to order the columns of  $A$  to increase the sparsity of the computed  $L$  and  $U$  factors, and hopefully increase parallelism (for SuperLU\_MT). Built-in choices are described later.
2. Compute the LU factorization of  $AP_c$ . SuperLU and SuperLU\_MT can perform dynamic pivoting with row interchanges during factorization for numerical stability, computing  $P_r$ ,  $L$ , and  $U$  at the same time.
3. Solve the system using  $P_r$ ,  $P_c$ ,  $L$ , and  $U$  as described above (where  $D_r = D_c = I$ ).

The simple driver subroutines for double precision real data are called `dgsvv` and `pdgsvv` for SuperLU and SuperLU\_MT, respectively. The letter `d` in the subroutine names means double precision real; other options are `s` for single precision real, `c` for single precision complex, and `z` for double precision complex. SuperLU\_DIST does not include this simple driver.

There is also an *expert driver* subroutine that can provide more accurate solutions, compute error bounds, and solve a sequence of related linear systems more economically. It is available in all three libraries.

### Expert Driver

1. Equilibrate the matrix  $A$ , that is, compute diagonal matrices  $D_r$  and  $D_c$  so that  $\hat{A} = D_r A D_c$  is “better conditioned” than  $A$ , that is,  $\hat{A}^{-1}$  is less sensitive to perturbations in  $\hat{A}$  than  $A^{-1}$  is to perturbations in  $A$ .
2. Preorder the rows of  $\hat{A}$  (SuperLU\_DIST only), that is, replace  $\hat{A}$  by  $P_r \hat{A}$  where  $P_r$  is a permutation matrix. This step is called “static pivoting,” and is only done in the distributed-memory algorithm, which allows scaling to more processors.
3. Order the columns of  $\hat{A}$ , to increase the sparsity of the computed  $L$  and  $U$  factors, and increase parallelism (for SuperLU\_MT and SuperLU\_DIST). In other words, replace  $\hat{A}$  by  $\hat{A} P_c^T$  in SuperLU and SuperLU\_MT, or replace  $\hat{A}$  by  $P_c \hat{A} P_c^T$  in SuperLU\_DIST, where  $P_c$  is a permutation matrix.

4. Compute the LU factorization of  $\hat{A}$ . SuperLU and SuperLU\_MT can perform dynamic pivoting with row interchanges for numerical stability. In contrast, SuperLU\_DIST uses the order computed by the preordering step (Step 2), and replaces tiny pivots by larger values for stability; this is corrected by Step 6.
5. Solve the system using the computed triangular factors.
6. Iteratively refine the solution, again using the computed triangular factors. This is equivalent to Newton’s method.
7. Compute error bounds. Both forward and backward error bounds are computed.

The expert driver subroutines for double precision real data are called `dgsvvx`, `pdgsvvx`, and `pdgsvvx` for SuperLU, SuperLU\_MT, and SuperLU\_DIST, respectively.

The driver routines are composed of several lower level computational routines for computing permutations, computing LU factorization, solving triangular systems, and so on. For large matrices, the LU factorization step takes most of the time, although choosing  $P_c$  to order the columns can also be time consuming.

## Common Features of the Three Libraries

### Supernodes in the Factors

The factorization algorithms in all three libraries use unsymmetric supernodes [3], which enable the use of higher level BLAS routines with higher flops-to-byte ratios, and so higher speed. A supernode is a range ( $r : s$ ) of columns of  $L$  with the triangular block just below the diagonal being full, and the same nonzero structure below the triangular block. Matrix  $U$  is partitioned rowwise by the same supernodal boundaries. But due to the lack of symmetry, the nonzero pattern of  $U$  consists of dense column segments of different lengths.

S

### Sparse Matrix Data Structure

The principal data structure for a matrix is SuperMatrix, defined as a C structure. This structure contains two levels of fields. The first level defines the three storage-independent properties of a matrix: mathematical type, data type, and storage type. The second level points to the actual storage used to store the compressed matrix. Specifically, matrix  $A$  is stored

in either column-compressed format (aka Harwell-Boeing format), or row-compressed format (i.e.,  $A^T$  stored in column-compressed format) [1]. Matrices  $B$  and  $X$  are stored as a single dense matrix of dimension  $n \times nrhs$  in column-major order, with output  $X$  overwriting input  $B$ . In SuperLU\_DIST,  $A$  and  $B$  can be either replicated or distributed across all processes. The factored matrices  $L$  and  $U$  are stored differently in SuperLU/SuperLU\_MT and SuperLU\_DIST, to be described later.

### Options Input Argument

The options is an input argument to control the behaviour of the libraries. The user can tell the solvers how the linear systems should be solved based on some known characteristics of the system. For example, for diagonally dominant matrices, choosing the diagonal pivots ensures stability; there is no need for numerical pivoting (i.e.,  $P_r$  can be an identity matrix). In another situation where a sequence of matrices with the same sparsity pattern needs to be factorized, the column permutation  $P_c$  (and also the row permutation  $P_r$ , if the numerical values are similar) needs to be computed only once, and reused thereafter. In these cases, the solvers' performance can be much improved over using the default parameter settings.

### Performance-Tuning Parameters

All three libraries depend on having an optimized BLAS library to achieve high performance [7, 8]. In particular, they depend on matrix–vector multiplication or matrix–matrix multiplication of relatively small dense matrices arising from the supernodal structure. The block size of these small dense matrices can be tuned to match the “sweet spot” of the BLAS performance on the underlying architecture. These parameters can be altered in the inquiry routine `sp_ienv`.

### Example Programs

In the source code distribution, the EXAMPLE/ directory contains several examples of how to use the driver routines, illustrating the following usages:

- Solve a system once
- Solve different systems with the same  $A$ , but different right-hand sides
- Solve different systems with the same sparsity pattern of  $A$

- Solve different systems with the same sparsity pattern and similar numerical values of  $A$

Except for the case of one-time solution, all the other examples can reuse some of the data structures obtained from a previous factorization, hence, save some time compared to factorizing  $A$  from scratch. The users can easily modify these examples to fit their needs.

### Differences Between SuperLU/ SuperLU\_MT and SuperLU\_DIST

#### Numerical Pivoting

Both sequential SuperLU and SuperLU\_MT use *partial pivoting with diagonal threshold*. The row permutation  $P_r$  is determined dynamically during factorization. At the  $j$ th column, let  $a_{mj}$  be a largest entry in magnitude on or below the diagonal of the partially factored  $A$ :  $|a_{mj}| = \max_{i \geq j} |a_{ij}|$ . Depending on a threshold  $u$  ( $0.0 \leq u \leq 1.0$ ) selected by the user, the code may use the diagonal entry  $a_{jj}$  as the pivot in column  $j$  as long as  $|a_{jj}| \geq u |a_{mj}|$  and  $a_{jj} \neq 0$ , or else use  $a_{mj}$ . If the user sets  $u = 1.0$ ,  $a_{mj}$  (or an equally large entry) will be used as the pivot; this corresponds to the classical partial pivoting. If the user has ordered the matrix so that choosing diagonal pivots is particularly good for sparsity or parallelism, then smaller values of  $u$  tend to choose those diagonal pivots, at the risk of less numerical stability. Selecting  $u = 0.0$  guarantees that the pivot on the diagonal will be chosen, unless it is zero. The code can also use a user-input  $P_r$  to choose pivots, as long as each pivot satisfies the threshold for each column. The backward error bound *BERR* measures how much stability is actually lost.

It is hard to get satisfactory execution speed with partial pivoting on distributed-memory machines, because of the fine-grained communication and the dynamic data structures required. SuperLU\_DIST uses a *static pivoting* strategy, in which  $P_r$  is chosen before factorization and based solely on the values of the original  $A$ , and remains fixed during factorization. A maximum weighted matching algorithm and the code MC64 developed by Duff and Koster [6] is currently employed. The algorithm chooses  $P_r$  to maximize the product of the diagonal entries, and chooses  $D_r$  and  $D_c$  simultaneously so that each diagonal entry of  $P_r D_r A D_c$  is  $\pm 1$  and each off-diagonal entry is bounded by 1 in magnitude. On the basis of empirical evidence, when

this strategy is combined with diagonal scaling, setting very tiny pivots to larger values, and iterative refinement, the algorithm is as stable as partial pivoting for most matrices that have occurred in the actual applications. The detailed numerical experiments can be found in [19].

### Sparsity-Preserving Reordering

For unsymmetric factorizations, preordering for sparsity is less understood than that for Cholesky factorization. Many unsymmetric ordering methods use symmetric ordering techniques, either minimum degree or nested dissection, applied to a symmetrized matrix (e.g.,  $A^T A$  or  $A^T + A$ ). This attempts to minimize certain upper bounds on the actual fills. Which symmetrized matrix to use strongly depends on how the numerical pivoting is performed.

In sequential SuperLU and SuperLU\_MT, where partial pivoting is used, an  $A^T A$ -based ordering algorithm is preferable. This is because the nonzero pattern of the Cholesky factor  $R$  in  $A^T A = R^T R$  is a superset of the nonzero pattern of the  $L^T$  and  $U$  in  $P_r A = LU$ , for any with row interchanges [9]. Therefore, a good symmetric ordering  $P_c$  on  $A^T A$  that preserves the sparsity of  $R$  can be applied to the columns of  $A$ , forming  $AP_c^T$ , so that the LU factorization of  $AP_c^T$  is likely to be sparser than that of the original  $A$ .

In SuperLU\_DIST, an a priori row permutation  $P_r$  is computed to form  $P_r A$ . With fixed  $P_r$ , an  $(A^T + A)$ -based ordering algorithm is preferable. This is because the symbolic Cholesky factor of  $A^T + A$  is a much tighter upper bound on the structures of  $L$  and  $U$  than that of  $A^T A$  when the pivots are chosen on the diagonal. Note that after  $P_c$  is chosen, a symmetric permutation  $P_c(P_r A)P_c^T$  is performed so that the diagonal entries of the permuted matrix remain the same as those in  $P_r A$ , and they are larger in magnitude than the off-diagonal entries. Now the final row permutation is  $P_c P_r$ .

In all three libraries, the user can choose one of the following ordering methods by setting the options .ColPerm option:

- NATURAL: Natural ordering
- MMD\_ATA: Multiple Minimum Degree [21] applied to the structure of  $A^T A$
- MMD\_AT\_PLUS\_A: Multiple Minimum Degree applied to the structure of  $A^T + A$

- METIS\_ATA: MeTiS [14] applied to the structure of  $A^T A$
- METIS\_AT\_PLUS\_A: MeTiS applied to the structure of  $A^T + A$
- PARMETIS: ParMeTiS [15] applied to the structure of  $A^T + A$
- COLAMD: Column Approximate Minimum Degree [2]
- MY\_PERMC: Use a permutation  $P_c$  supplied by the user as input

COLAMD is designed particularly for unsymmetric matrices when partial pivoting is needed, and does not require explicitly forming  $A^T A$ . It usually gives comparable orderings as MMD on  $A^T A$ , and is faster.

The purpose of the last option MY\_PERMC is to be able to reap the results of active research in the ordering methods. Recently, there is much research on the orderings based on graph partitioning. The user can invoke those ordering algorithms separately, and then input the ordering in the permutation vector for  $P_c$ . The user may apply them to the structures of  $A^T A$  or  $A^T + A$ . The routines getata() and at\_plus\_a() in the file get\_perm\_c.c can be used to form  $A^T A$  or  $A^T + A$ .

### Task Ordering

The Gaussian elimination algorithm can be organized in different ways, such as left-looking (fan-in) or right-looking (fan-out). These variants are mathematically equivalent under the assumption that the floating-point operations are associative (approximately true), but they have very different memory access and communication patterns. The pseudo-code for the left-looking blocking algorithm is given in Algorithm 1.

---

#### Algorithm 1 Left-looking Gaussian elimination

---

```

for block $K = 1$ to N do
 (1) Compute $U(1 : K - 1, K)$
 (via a sequence of triangular solves)
 (2) Update $A(K : N, K) \leftarrow A(K:N, K)$
 $-L(1 : N, 1 : K - 1) \cdot U(1 : K - 1, K)$
 (via a sequence of calls to GEMM)
 (3) Factorize $A(K : N, K) \rightarrow L(K : N, K)$
 (may involve pivoting)
end for

```

---

SuperLU and SuperLU\_MT use the left-looking algorithm, which has the following advantages:

- In each step, the sparsity changes are restricted within the  $K$ th block column instead of the entire trailing submatrix, which makes it relatively easy to accommodate dynamic compressed data structures due to partial pivoting.
- There are more memory *read* operations than *write* operations in [Algorithm 1](#). This is better for most modern cache-based computer architectures, because write tends to be more expensive in order to maintain cache coherence.

The pseudo-code for the right-looking blocking algorithm is given in [Algorithm 2](#).

---

**Algorithm 2** *Right-looking Gaussian elimination*


---

```

for block $K = 1$ to N do
 (1) Factorize $A(K : N, K) \rightarrow L(K : N, K)$
 (may involve pivoting)
 (2) Compute $U(K, K + 1 : N)$
 (via a sequence of triangular solves)
 (3) Update $A(K + 1 : N, K + 1 : N) \leftarrow$
 $A(K + 1 : N, K + 1 : N) - L(K + 1 : N, K) \cdot$
 $U(K, K + 1 : N)$
 (via a sequence of calls to GEMM)
end for

```

---

SuperLU\_DIST uses right-looking algorithm mainly for scalability consideration.

- The sparsity pattern and data structure can be determined before numerical factorization because of static pivoting.
- The right-looking algorithm fundamentally has more parallelism: at step (3) of [Algorithm 2](#), all the GEMM updates to the trailing submatrix are independent and so can be done in parallel. On the other hand, each step of the left-looking algorithm involves operations that need to be carefully sequenced, which requires a sophisticated pipelining mechanism to exploit parallelism across multiple loop steps.

## Parallelization and Performance

SuperLU\_MT is designed for parallel machines with shared address space, thus there is no need to partition the matrices. Matrices  $A$ ,  $L$ , and  $U$  are stored in separate compressed formats. The parallel elimination uses an asynchronous and barrier-free scheduling algorithm to schedule two types of parallel tasks to achieve a high degree of concurrency. One such task is factorizing the independent panels in the disjoint subtrees of the column elimination tree [10]. Another task is updating a panel by previously computed supernodes. The scheduler facilitates the smooth transition between the two types of tasks, and maintains load balance dynamically. In symbolic factorization, a non-blocking algorithm is used to perform depth-first search and symmetric pruning in parallel. The code achieved over tenfold speedups on a number of earlier SMP machines with 16 processors [4]. Recent evaluation shows that SuperLU\_MT performs very well on current multi-threaded, multicore machines; it achieved over 20-fold speedup on a 16 core, 128 thread Sun VictoriaFalls [16].

The design of SuperLU\_DIST is drastically different from SuperLU/SuperLU\_MT. Many design choices were made by the need for scaling to a large process count. The input sparse matrix  $A$  is divided by block rows, with each process having one block row represented in a local row-compressed format. This format is user-friendly and is compatible with the input interface of much other distributed-memory sparse matrix software. The factored  $L$  and  $U$  matrices, on the other hand, are distributed by a two-dimensional block cyclic layout using supernodal structure for block partition. This distribution ensures that most (if not all) processors are engaged in the right-looking update at each block elimination step, and also ensures that interprocess communication is restricted among process row set or column set. The right-looking factorizations use elimination DAGs to identify task and data dependencies, and a one step look-ahead scheme to overlap communication with computation. A distributed parallel symbolic factorization algorithm is designed so that there is no need to gather the entire graph on a single node, which largely increases memory scalability [12]. SuperLU\_DIST has achieved 50- to 100-fold speedups with sufficiently large matrices, and over half a Teraflops factorization rate [18].

## Future Directions

There are still many open problems in the development of high performance algorithm and software for sparse direct methods. The current architecture trend shows that the Chip Multiprocessor (CMP) will be the basic building block for computer systems ranging from laptops to extreme high-end supercomputers. The core count per chip will quickly increase from four to eight today to tens in the near future. Given the limited per-chip memory and memory bandwidth, the standard parallelization procedure based on MPI would suffer from serious resource contention. It becomes essential to consider a hybrid model of parallelism at the algorithm level as well as the programming level. The quantitative multicore evaluation of SuperLU shows that the left-looking algorithm in SuperLU\_MT consistently outperforms the right-looking algorithm in SuperLU\_DIST on a single node of the recent CMP systems, mainly because the former incurs much less memory traffic [16]. One new design is to combine the two algorithms – partitioning the matrix into larger panels, performing left-looking intra-chip elimination and right-looking inter-chip elimination.

A second new direction is to exploit the property of low numerical rank in many discretized operators so that a specialized Gaussian elimination algorithm can be designed. For example, it has been shown recently that *semi-separable structures* occur in the pivoting block and the corresponding Schur complement of each block factorization step. Thus, using the compressed, semi-separable representation throughout the entire factorization leads to an approximate factorization algorithm that has nearly linear time and space complexity [13, 22]. For many discretized elliptic PDEs, this approximation is sufficiently accurate and can be used as an optimal direct solver. For more general problems, the factorization can be used as an effective preconditioner. Because of its asymptotically lower data volume compared with conventional algorithms, the amount of memory-to-processor and inter-processor communication is smaller, making it more amenable to a scalable implementation.

Another promising area is to extend the new communication avoiding dense LU and QR factorizations to sparse factorizations [11]. In conventional algorithms, the panel factorization of a tall-skinny

submatrix requires a sequence of fine-grained message transfers, and often lies on the critical path of parallel execution. The new method employs a divide-and-conquer scheme for this phase, which has asymptotically less communication. This method should also work for sparse matrices. In particular, the new pivoting strategy for LU can replace the static pivoting currently used in SuperLU, leading to a more stable and scalable solver.

## Related Entries

- [Chaco](#)
- [LAPACK](#)
- [METIS and ParMETIS](#)
- [Mumps](#)
- [PARDISO](#)
- [PETSc \(Portable, Extensible Toolkit for Scientific Computation\)](#)
- [Preconditioners for Sparse Iterative Methods](#)
- [Reordering](#)
- [ScaLAPACK](#)
- [Sparse Direct Methods](#)

## Bibliography

1. Barrett R, Berry M, Chan TF, Demmel J, Donato J, Dongarra J, Eijkhout V, Pozo R, Romine C, van der Vorst H (1994) Templates for the solution of linear systems: building blocks for the iterative methods. SIAM, Philadelphia, PA
2. Davis TA, Gilbert JR, Larimore S, Ng E (2004) A column approximate minimum degree ordering algorithm. ACM Trans Math Softw 30(3):353–376
3. Demmel JW, Eisenstat SC, Gilbert JR, Li XS, Liu JWH (1999) A supernodal approach to sparse partial pivoting. SIAM J Matrix Anal Appl 20(3):720–755
4. Demmel JW, Gilbert JR, Li XS (1999) An asynchronous parallel supernodal algorithm for sparse gaussian elimination. SIAM J Matrix Anal Appl 20(4):915–952
5. Demmel JW, Gilbert JR, Li XS (1999) SuperLU users' guide. technical report LBNL-44289, Lawrence Berkeley National Laboratory, September 1999. <http://crd.lbl.gov/~xiaoye/SuperLU/>. Last update: September 2007
6. Duff IS, Koster J (1999) The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. SIAM J Matrix Anal Appl 20(4):889–901
7. BLAS Technical Forum (2002) Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard I. Int J High Perform Comput Appl 16:1–111

8. BLAS Technical Forum (2002) Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard II. *Int J High Perform Comput Appl* 16:115–199
9. George A, Liu J, Ng E (1988) A data structure for sparse QR and LU factorizations. *SIAM J Sci Stat Comput* 9:100–121
10. Gilbert JR, Ng E (1993) Predicting structure in nonsymmetric sparse matrix factorizations. In: George A, Gilbert JR, Liu JWH (eds) *Graph theory and sparse matrix computation*. Springer-Verlag, New York, pp 107–139
11. Grigori L, Demmel J, Xiang H (2008) Communication-avoiding Gaussian elimination. In: *Supercomputing 08*, Austin, TX, November 15–21, 2008.
12. Grigori L, Demmel JW, Li XS (2007) Parallel symbolic factorization for sparse LU with static pivoting. *SIAM J Sci Comput* 29(3):1289–1314
13. Gu M, Li XS, Vassilevski P (2010) Direction-preserving and schur-monotonic semi-separable approximations of symmetric positive definite matrices. *SIAM J Matrix Anal Appl* 31(5):2650–2664
14. Karypis G, Kumar V (1998) MeTiS – a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices – version 4.0. University of Minnesota, September 1998. <http://www-users.cs.umn.edu/~karypis/metis/>. Accessed 2010
15. Karypis G, Schloegel K, Kumar V (2003) ParMeTiS: Parallel graph partitioning and sparse matrix ordering library – version 3.1. University of Minnesota. <http://www-users.cs.umn.edu/~karypis/metis/parmetis/>. Accessed 2010
16. Li XS (2008) Evaluation of sparse factorization and triangular solution on multicore architectures. In: *Proceedings of VEPAR08 8th international meeting high performance computing for computational science*, Toulouse, France, June 24–27, 2008
17. Li XS (Sept 2005) An overview of SuperLU: algorithms, implementation, and user interface. *ACM Trans Math Softw* 31(3):302–325
18. Li XS (2009) Sparse direct methods on high performance computers. University of California, Berkeley, CS267 Lecture Notes
19. Li XS, Demmel JW (June 2003) SuperLU DIST: a scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans Math Softw* 29(2):110–140
20. Li XS, Shao M (2011) A supernodal approach to incomplete LU factorization with partial pivoting. *ACM Trans Math Softw* 37(4)
21. Liu JWH (1985) Modification of the minimum degree algorithm by multiple elimination. *ACM Trans Math Softw* 11:141–153
22. Xia J, Chandrasekaran S, Gu M, Li XS (2009) Superfast multifrontal method for large structured linear systems of equations. *SIAM J Matrix Anal Appl* 2008, 31(3):1382–1411

## Supernode Partitioning

► [Tiling](#)

## Superscalar Processors

WEN-MEI HWU

University of Illinois at Urbana-Champaign, Urbana, IL, USA

### Synonyms

[Multiple-instruction issue](#); [Out-of-order execution processors](#)

### Definition

A superscalar processor is designed to achieve an execution rate of more than one instruction per clock cycle for a single sequential program.

### Discussion

#### Introduction

Superscalar processor design typically refers to a set of techniques that allow the central processing unit (CPU) of a computer to achieve a throughput of more than one instruction per cycle while executing a single sequential program. While there is not a universal agreement on the definition, superscalar design techniques typically include parallel instruction decoding, parallel register renaming, speculative execution, and out-of-order execution. These techniques are typically employed along with complementing design techniques such as pipelining, caching, branch prediction, and multi-core in modern microprocessor designs.

A typical superscalar processor today is the Intel Core i7 processor based on the Nehalem microarchitecture. There are multiple processor cores in a Core i7 design, where each processor core is a superscalar processor. The processor performs parallel decoding on IA (X86) instructions, performs parallel register renaming to map the X86 registers used by these instructions to a larger set of physical registers, performs speculative execution of instructions beyond conditional branch instruction and potential exception causing instructions, and allows instructions to execute out of their program specified order while maintaining the appearance of in-order completion. These techniques are accompanied and supported by pipelining, instruction caching, data caching, and branch prediction in the design of each processor core.

## Instruction-Level Parallelism and Dependences

Superscalar processor design assumes the existence of instruction-level parallelism, a phenomenon that multiple instructions can be executed independently of each other at each point in time. Instruction-level parallelism arises due to the fact that instructions in an execution phase of a program often read and write different data, thus their executions do not affect each other.

In Fig. 1, Instruction A is a memory load instruction that forms its address from the contents of register 1 (r1), accesses the data in the address location, and deposits the accessed data into register 2 (r2). Instruction B is a memory load instruction that forms its address by adding value 4 to the contents of register 1 (r1), accesses the data in the address location, and deposits the accessed data into register 3 (r3). Instructions A and B can be executed independently of each other. Neither of their execution results is affected by the other. With sufficient execution resources, a superscalar processor can execute A and B together and achieve an execution rate of more than one instruction per clock cycle.

Note that Instruction C cannot be executed independently of Instruction A or Instruction B. This is because Instruction C uses the data in register 2 (r2) and register 3 (r3) as its input. These data are loaded from memory by Instruction A and Instruction B. That is, the execution of Instruction C *depends* on that of instruction A and Instruction B. Such dependences limit the amount of instruction-level parallelism that exists in a program.

In general, *data dependences* arise in programs due to the way instructions read from and write into register and memory storage locations. Data dependencies occur between instructions in three forms. We use the

|   |                                       |
|---|---------------------------------------|
| A | $r2 \leftarrow \text{Load MEM}[r1+0]$ |
| B | $r3 \leftarrow \text{Load MEM}[r1+4]$ |
| C | $r2 \leftarrow r2+r3$                 |
| D | $r3 \leftarrow \text{Load MEM}[r1+8]$ |

Superscalar Processors. Fig. 1 Code example for register access data dependences

code example in Fig. 1 to illustrate these forms of data dependences:

1. Data flow dependency: the destination register of Instruction A is the same as one of the source registers of Instruction C, and C follows A in the sequential program order. In this case, a subsequent instruction consumes the execution result of a previous instruction.
2. Data antidependency: one of the source operands of Instruction C is the same as the destination register of Instruction D, and D follows C in the sequential program order. In this case, Instruction C should receive result of Instruction B. However, if Instruction D is executed too soon, C may receive execution result of Instruction D, which is too new. In this case, a subsequent instruction overwrites one of the source registers of a previous instruction.
3. Data output dependency: the destination register of Instruction B is the same as the destination register of Instruction D, and Instruction D follows B in the sequential program order. In this case, a subsequent instruction overwrites the destination register of a previous instruction. If D is executed too soon, its result could be overwritten by Instruction B, leaving a result which is too old in the destination register. Subsequent instructions would be using stale results.

A superscalar processor uses register renaming and out-of-order execution techniques to detect and enhance the amount of instruction-level parallelism between instructions so that it can execute multiple instructions per clock cycle. These techniques ensure that all instructions acquire the appropriate input value in the presence of all the parallel execution activities and data dependences. They also make sure that the output values of instructions are reflected correctly in the processor registers and memory locations.

## Register Renaming

Register renaming is a technique that eliminates register antidependences and output dependences in order to increase instruction parallelism. A register renaming mechanism provides a physical or implementation register file that is larger than the architectural register files. For example, the IA (X86) architecture specifies 8 general-purpose registers whereas the register

renaming mechanism of a superscalar processor typically provides 32 or more physical registers. At any time, each architectural register is mapped to one or more of the physical registers. By mapping, an architectural register is mapped to multiple physical registers, one can eliminate apparent antidependences and output dependences between instructions.

For example, a register renaming mechanism may map architectural register r3 to physical register pr103 for the destination operand of Instruction B and source of Instruction C in [Fig. 1](#). That is, Instruction B will deposit its result to pr103 and Instruction C will fetch its second input operand from pr103. As long as the producer of the data and all the consumers of the data are redirected to the same physical register, the execution results will not be affected.

Let's further assume that architecture register r3 is mapped to physical register pr105 for the destination operand of Instruction D. That is, Instruction D will deposit its execution results to pr105. As long as all subsequent uses of the data produced by Instruction D are redirected to pr105 for their input, their execution results will remain the same.

The reader should see that the antidependence between Instruction C and Instruction D has been eliminated by the register renaming mechanism. Since Instruction C will go to pr103 for its input whereas Instruction D will deposit its result into pr105, Instruction D will no longer overwrite the input for Instruction C no matter how soon it is executed. As a result, we have eliminated a dependence constraint between Instruction C and Instruction D.

The reader should also notice that the output dependence between Instruction B and Instruction D has been eliminated by the register renaming mechanism. Since Instruction B deposits its result into pr103 and Instruction D deposits its result into pr105, B will no longer overwrite the result of D no matter how soon D is executed. As long as subsequent Instructions that use architecture register r3 are also directed to pr105, they will see the updated results rather than stale results even though B could be executed after D.

By eliminating the register antidependence between C and D and output dependence between B and D, Instruction D can now be executed in parallel with Instructions A and B. This increases the level of

instruction-level parallelism. This is the reason why register renaming has become an essential mechanism for superscalar processor design.

## Speculative Execution

A major limitation on the amount of instruction-level parallelism is uncertainties in program execution sequence. There are two major sources of such uncertainty. The first is conditional and indirect branch instructions. Conditional branches are used to implement control constructs such as if-then-else statements and loops in high-level language programs. Depending on the condition values, the instructions to be executed after a conditional branch can be either from the next sequential location or from a target location specified by the branch offset. Indirect branches use the contents of a register or memory location as the address of the next instruction to execute. Indirect branches are used to implement procedure return statements, case statements, and table-based procedural calls in high-level language programs.

While conditional and indirect branch instructions serve essential purposes in implementing high-level languages, they introduce uncertainties for execution. The classic approach to addressing this problem is branch prediction, where a mechanism is used to predict the next instructions to execute when conditional and indirect branches are encountered during program execution. However, prediction alone is not sufficient. One needs a means to speculatively execute the instructions on the predicted path of execution and recover from any incorrect predictions.

The second source of uncertainty in program execution is exception conditions. Modern computers use exceptions to support the implementation of virtual memory management, memory protection, and rare execution condition handling. For example, the Instruction A in [Fig. 1](#) may trigger a page fault in its execution and require operating system service to bring in its missing load data. The execution needs to be able to resume cleanly after the page fault is handled by the operating system. This requirement means that the instructions after a load instruction, or any instruction that can potentially cause exceptions, cannot change the execution state in a way that prevents the execution from restarting at the exception causing instruction.

Like in the case of branch prediction, one can “predict” that the exception conditions do not occur and assume that the execution will simply continue down the current path. However, one needs a means to recover the state when an exception indeed occurs so that the execution can correctly restart from the exception-causing instruction.

Speculative execution is a mechanism that allows processors to fetch and execute instructions down a predicted path and to recover if the prediction is incorrect. In general, these mechanisms use buffers to keep both the original state and the recent updates to the state. The updated state is used during the speculative execution. The original state is used if the processor needs to recover from an incorrect prediction. Since conditional and indirect branches occur frequently, one in every four to five instructions on average, the level of instruction level parallelism a superscalar processor can exploit would be extremely low without speculative execution. This is why speculative execution has become an essential mechanism in superscalar processor design. The most popular methods for recovering from incorrect branch predictions are reorder buffer (ROB) and checkpointed register file. The most popular method for recovering from exception causing instructions is reorder buffer.

A retirement mechanism in speculative execution attempts to make the effects of instruction execution permanent. An instruction is eligible for retirement when all instructions before it have retired. An eligible instruction is then checked if it has caused any exceptions or incorrect branch prediction. If the instruction does not incur any exception or incorrect branch prediction, it can commit its execution result into an *architectural state*. Otherwise, the instruction triggers a recovery and the state of the processor is restored to a previous architectural state.

The capacity of reorder buffers and checkpoint buffers defines the notion of *instruction window*, a collection of consecutive instructions that are actively processed by a superscalar processor. At any point in time, the instruction window starts with the oldest instruction that has not completed execution and ends with the youngest instruction that has started execution. The larger the instruction window, the more hardware is needed to keep track of the execution status

of all instructions in the window and the information needed to recover the processor state if anything goes wrong with the execution of the instructions in the window.

## Out-of-Order Execution

In a superscalar processor, instructions are fetched according to their sequential program order. However, this may not be best order of execution. For example, in Fig. 1, Instruction D is fetched after Instruction C. However, Instruction C cannot execute until Instructions B and Instruction D completes their execution. On the other hand, with register renaming, Instruction D does not have any dependence on Instructions A, Instruction B, or Instruction C. Therefore, a superscalar processor “reorders” the execution of Instruction C and Instruction D so that Instruction D can produce results for its consumers as soon as possible.

An out-of-order execution mechanism provides buffering for Instructions that need to wait for their input data so that some of their subsequent instructions can proceed with execution. A popular method used in out-of-order execution is the Tomasulo’s Algorithm that was originally used in the IBM 360/91. The method maintains *reservation stations*, hardware buffers that allow the instructions to wait for their input operands.

## Brief Early History

The most popular out-of-order execution mechanism in modern superscalar processors is Tomasulo’s algorithm [1] designed by Bob Tomasulo and used in the IBM 360/91 floating point unit in 1967. The out-of-order-execution mechanism was abandoned in later IBM machines partly due to the concern of the problems it introduces to virtual memory management.

The Intel Pentium processor [2] is an early superscalar design that fetches and executes multiple instructions at each clock cycle. It did not employ register renaming or speculative execution and was able to exploit only a very limited amount of instruction-level parallelism.

Register renaming in superscalar processor design started with the Register Alias Table (RAT) by Patt et al. [3]. To this day, the register renaming structure used in Intel superscalar processors are still called RAT.

Smith and Pleszkun proposed reorder buffers and history buffers for recovering from exceptions in highly pipelined processors [4]. Hwu et al. [5] extended the concepts with checkpointing and incorporated these speculative execution mechanisms with Tomasulo's algorithm.

In 1985, Patt et al. [3] proposed a comprehensive superscalar design that incorporates register renaming, speculative execution, out-of-order execution, along with parallel instruction decode and branch prediction. This is the first comprehensive academic design of superscalar processors. In 1987, Sohi and Vijapeyam [6] proposed a unified reservation station design. These designs were later adopted and refined by Intel to create the Pentium Pro Processor, the first commercially successful superscalar processor design [7].

Recent superscalar processors include MIPS R10000, Intel Pentium 4, IBM Power 6, AMD Athlon, and ARM Cortex. Interested readers should refer to Hennessy and Patterson [8] for more detailed treatment and more recent history on superscalar processor design.

## Bibliography

1. Tomasulo R (1967) An efficient algorithm for exploiting multiple arithmetic units. *IBM J Res Dev* 11(1):8–24
2. Case B (1993) Intel reveals pentium implementation details. *Microprocessor Report* 29 Mar 1993
3. Patt Y, Hwu W-M, Shebanow M (1985) HPS, a new microarchitecture: rationale and introduction. In: Proceedings of the 18th annual workshop on microprogramming, Pacific Grove, pp 103–108
4. Smith J, Pleszkun A (1985) Implementation of precise interrupts in pipelined processors. In: Proceedings of the 12th international symposium on computer architecture, Boston
5. Hwu W-M, Patt Y (1987) Checkpoint repair for out-of-order execution machines. In: Proceedings of the 14th international symposium on computer architecture, Pittsburgh
6. Sohi G, Vajapeyam S (1987) Instruction issue logic for high-performance, interruptable pipelined processors. In: Proceedings of the 14th international symposium on computer architecture, New York
7. Colwell R (2005) Pentium chronicles – the people, passion, and politics behind Intel's Lanmark chips. Wiley-IEEE Computer Society, ISBN 978-0-47-173617-2
8. Hennessy J, Patterson D (2007) Computer architecture – a quantitative approach, 4th edn. Morgan Kauffman, San Francisco, ISBN 978-0-12-370490-0

## SWARM: A Parallel Programming Framework for Multicore Processors

DAVID A. BADER<sup>1</sup>, GUOJING CONG<sup>2</sup>

<sup>1</sup>Georgia Institute of Technology, Atlanta, GA, USA

<sup>2</sup>IBM, Yorktown Heights, NY, USA

## Definition

SoftWare and Algorithms for Running on Multi-core (SWARM) is a portable open-source parallel library of basic primitives for programming multicore processors. SWARM is built on POSIX threads that allows the user to use either the already developed primitives or direct thread primitives. SWARM has constructs for parallelization, restricting control of threads, allocation and deallocation of shared memory, and communication primitives for synchronization, replication and broadcast. Built on these techniques, it contains a higher-level library of multicore-optimized parallel algorithms for list ranking, comparison-based sorting, radix sort, and spanning tree. In addition, SWARM application example codes include efficient implementations for solving combinatorial problems such as minimum spanning tree [3], graph decomposition [8], breadth-first-search [9], tree contraction [10], and maximum parsimony [7].

## Motivation

For the last few decades, software performance has improved at an exponential rate, primarily driven by the rapid growth in processing power. However, performance improvements can no longer rely solely on Moore's law. Fundamental physical limitations such as the size of the transistor and power constraints have now necessitated a radical change in commodity microprocessor architecture to multicore designs. Dual and quad-core processors from Intel [13] and AMD [2] are now ubiquitous in home computing. Also, several novel architectural ideas are being explored for high-end workstations and servers [15, 16]. Continued software performance improvements on such novel multicore systems now require the exploitation of concurrency at the algorithmic level. Automatic methods for detecting concurrency from sequential codes, for example

with parallelizing compilers, have had only limited success. SWARM was introduced to fully utilize multicore processors.

On multicore processors, caching, memory bandwidth, and synchronization constructs have a considerable effect on performance. In addition to time complexity, it is important to consider these factors for algorithm analysis. SWARM assumes the multicore model that can be used to explain performance on systems such as Sun Niagara, Intel, and AMD multicore chips. Different models [6] are required for modeling heterogeneous multicore systems such as the Cell architecture.

## Model for Multicore Architectures

Multicore systems have a number of processing cores integrated on to a single chip [2, 11, 13, 15, 16]. Typically, the processing cores have their own private  $L_1$  cache and share a common  $L_2$  cache [13, 16]. In such a design, the bandwidth between the  $L_2$  cache and main memory is shared by all the processing cores. Figure 1 shows the simplified architectural model.

### Multicore Model

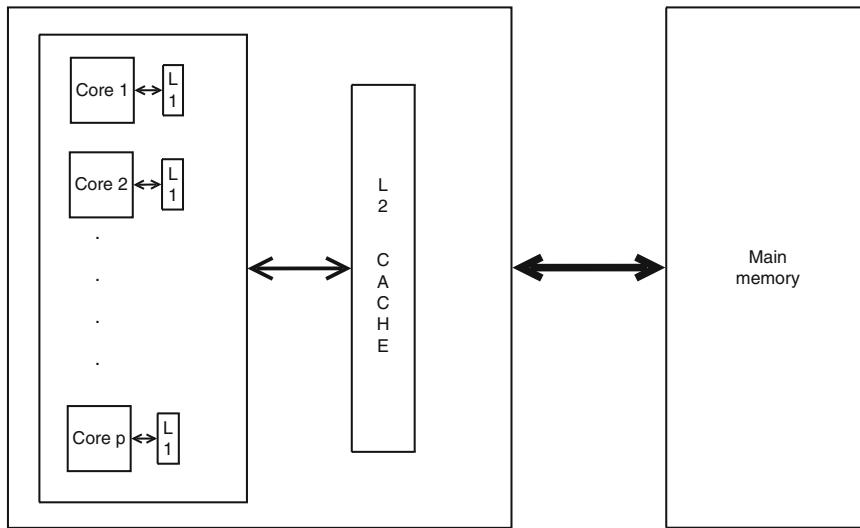
The multicore model (MCM) consists of  $p$  identical processing cores integrated onto a single chip. The

processing cores share an  $L_2$  cache of size  $C$ , and the memory bandwidth is  $\sigma$ .

1. Let  $T(i)$  denote the local time complexity of the core  $i$  for  $i = 1, \dots, p$ . Let  $T = \max_i T(i)$ .
2. Let  $B$  be the total number of blocks transferred between  $L_2$  cache and the main memory. The requests may arise out of any processing core.
3. Let  $L$  be the time required for synchronization between the cores. Let  $N_S(i)$  be the total number of synchronizations required on core  $i$  for  $i = 1, \dots, p$ . Let  $N_S = \max_i N_S(i)$ .

Then the complexity under the multicore model can be represented by a triple  $\langle T, B \cdot \sigma^{-1}, N_S \cdot L \rangle$ . The complexity of an algorithm will be represented by the dominant element in this triple.

The model proposed above is in many ways similar to the Helman-JáJá model for symmetric multiprocessor (SMP) systems [12], with a few important differences. In the case of SMPs, each processor typically has a large  $L_2$  cache and dedicated bandwidth to main memory, whereas in multicore systems, the shared memory bandwidth will be an important consideration. Thus, SWARM explicitly models the cache hierarchy, and count the number of block transfers between the cache and main memory in a manner similar to Aggarwal and Vitter's external memory model [1].



SWARM: A Parallel Programming Framework for Multicore Processors. Fig. 1 Architectural model for multicore systems

SWARM targets three primary issues that affect performance on multicore systems:

1. *Number of processing cores*: Current systems have two to eight cores integrated on a single chip. Cores typically support features such as simultaneous multithreading (SMT) or hardware multithreading, which allow for greater parallelism and throughput. In future designs, up to 100 cores can exist on a single chip.
2. *Caching and memory bandwidth*: Memory speeds have been historically increasing at a much slower rate than processor capacity [14]. Memory bandwidth and latency are important performance concerns for several scientific and engineering applications. Caching is known to drastically affect the efficiency of algorithms even on single processor systems [17, 18]. In multicore systems, this will be even more important due to the added bandwidth constraints.
3. *Synchronization*: Implementing algorithms using multiple processing cores will require synchronization between the cores from time to time, which is an expensive operation in shared memory architectures.

## Case Study: Merge Sort

### Algorithm 1

In the first algorithm, the input array of length  $N$  is equally divided among the  $p$  processing cores so that each core gets  $N/p$  elements to sort. Once the sorting phase is completed, there are  $p$  sorted sub-arrays, each of length  $N/p$ . Thereafter the merge phase takes place. A  $p$ -way merge over the runs will give the sorted array. Each processor individually sorts its elements using some *cache-friendly* algorithm. This approach does not try to minimize the number of blocks transferred between the  $L_2$  cache and main memory.

**Analysis.** Since the  $p$  processors are all sorting their respective elements at the same time, the  $L_2$  cache will be shared by all the cores during the sorting phase. Thus, if the size of the  $L_2$  cache is  $C$ , then effectively each core can use just a portion of the cache with size  $C/p$ . Assuming the input size is larger than the cache size, the cache misses will be  $p$  times that if only a single core were sorting. Also the bandwidth between the cache and shared

main memory is also shared by all the  $p$  cores, and this may be a bottleneck.

The time complexity of each processor is:

$$T_c(\text{sort}) = \frac{N}{p} \cdot \log\left(\frac{N}{p}\right)$$

During the merge phase:

$$T_c(\text{merge}) = N \cdot \log(p)$$

$$T_c(\text{total}) = \frac{N}{p} \log\left(\frac{N}{p}\right) + N \log(p)$$

### Algorithm 2

This algorithm divides the given array of length  $N$  into blocks of size  $M$  where  $M$  is less than  $C$ , the size of the  $L_2$  cache. Each of such  $N/M$  blocks is first sorted using all  $p$  cores. This is the sorting phase. When the sorting phase is completed, the array consists of  $N/M$  runs each of length  $M$ . During the merge phase,  $p$  blocks are merged at a time. This process is repeated till a single sorted array is arrived. Thus, the merge phase is carried out  $\log_p\left(\frac{N}{M}\right)$  times.

**Analysis.** This algorithm is very similar to the I/O model merge sort [1]. Thus, this algorithm is optimal in terms of transfers between main memory and  $L_2$  cache. However, it will have slightly higher-computational complexity. The  $p$  cores sort a total of  $N/M$  blocks of size  $M$ . Assuming the use of a split-and-merge sort for sorting the block of  $M$  elements, thus, during the sorting phase, the time per core is:

$$\begin{aligned} T_c(\text{sort}) &= \frac{N}{M} \cdot \frac{M}{p} \log\left(\frac{M}{p}\right) + \frac{N}{M} \cdot M \log(p) \\ &= \frac{N}{p} \log\left(\frac{M}{p}\right) + N \log(p) \end{aligned} \quad (1)$$

During any merge phase, if blocks of size  $S$  are being merged  $p$  at a time, the complexity per core is  $\frac{N}{Sp}$ .

$Sp \log(p) = N \log(p)$ . There are  $\log_p\left(\frac{N}{M}\right)$  merge phases, thus

$$T_c(\text{merge}) = N \log(p) \cdot \log_p\left(\frac{N}{M}\right)$$

$$T_c(\text{total}) = \frac{N}{p} \log\left(\frac{M}{p}\right) + N \log(p) \left(1 + \log_p\left(\frac{N}{M}\right)\right)$$

**Comparison.** Algorithm 1 clearly has better time complexity than algorithm 2. However, algorithm 2 is optimal in terms of transfers between  $L_2$  cache and shared

main memory. Algorithm analysis using this model captures computational complexity as well as memory performance.

## Programming in SWARM

A typical SWARM program is structured as follows:

```
int main (int argc, char **argv)
{
 SWARM_Init(&argc, &argv);
 /* sequential code */

 /* parallelize a routine using
 SWARM */
 SWARM_Run(routine);
 /* more sequential code */

 SWARM_Finalize();
}
```

In order to use the SWARM library, the programmer needs to make minimal modifications to existing sequential code. After identifying compute-intensive routines in the program, work can be assigned to each core using an efficient multicore algorithm. Independent operations such as those arising in *functional parallelism* or *loop parallelism* can be typically threaded. For functional parallelism, this means that each thread acts as a functional process for that function, and for loop parallelism, each thread computes its portion of the computation concurrently. Note that it might be necessary to apply loop transformations to reduce data dependencies between threads.

SWARM contains efficient implementations of commonly used primitives in parallel programming.

**Data parallel.** The SWARM library contains several basic “*pardo*” directives for executing loops concurrently on one or more processing cores. Typically, this is useful when an independent operation is to be applied to every location in an array, for example element-wise addition of two arrays. Pardo implicitly partitions the loop among the cores without the need for coordinating overheads such as synchronization of communication between the cores. By default, pardo uses block partitioning of the loop assignment values to the threads, which typically results in better cache utilization due to

the array locations on lefthand side of the assignment being owned by local caches more often than not. However, SWARM explicitly provides both block and cyclic partitioning interfaces for the *pardo* directive.

```
/* example: partitioning a "for"
 loop among the cores */
pardo(i, start, end, incr) {
 A[i] = B[i] + C[i];
}
```

**Control.** SWARM control primitives restrict which threads can participate in the context. For instance, the control may be given to a single thread on each core, all threads on one core, or a particular thread on a particular core.

```
THREADS: total number of execution
threads
MYTHREAD: the rank of a thread,
from 0 to THREADS-1
/* example: execute code on
 thread MYTHREAD */
on_thread(MYTHREAD) {

}
/* example: execute code on
 one thread */
on_one_thread {

}
```

**Memory management.** SWARM provides two directives *SWARM\_malloc* and *SWARM\_free* that, respectively, dynamically allocate a shared structure and release this memory back to the heap.

```
/* example: allocate a shared array
 of size n */
A=(int*) SWARM_malloc(n*sizeof(int),
TH);
/* example: free the array A */
SWARM_free(A);
```

**Barrier.** This construct provides a way to synchronize threads running on the different cores.

```
/* parallel code */
```

```

 ...
 ...
/* use the SWARM Barrier for
 synchronization */
SWARM_Barrier();
/* more parallel code */
 ...
 ...

```

**Replicate.** This primitive uniquely copies a data buffer for each core.

**Scan (reduce).** This performs a prefix (reduction) operation with a binary associative operator, such as addition, multiplication, maximum, minimum, bitwise-AND, and bitwise-OR. `allreduce` replicates the result from `reduce` for each core.

```

/* function signatures */
int SWARM_Reduce_i(int myval,
 reduce_t op,
 THREADED);
double SWARM_Reduce_d(double
 myval,
 reduce_t op,
 THREADED);

/* example: compute global sum,
using
partial local values from each
core */
sum = SWARM_Reduce_d(mySum, SUM,
TH);

```

**Broadcast.** This primitive supplies each processing core with the address of the shared buffer by replicating the memory address.

```

/* function signatures */
int SWARM_Bcast_i (int myval,
 THREADED);
int* SWARM_Bcast_ip (int* myval,
 THREADED);
char SWARM_Bcast_c (char myval,
 THREADED);

```

Apart from the primitives for computation and communication, the thread-safe parallel pseudo-random number generator SPRNG [19] is integrated in SWARM.

## Algorithm Design and Examples in SWARM

The SWARM library contains a number of techniques to demonstrate key methods for programming on multicore processors.

- The ***prefix-sum*** algorithm is one of the most useful parallel primitives and is at the heart of several other primitives, such as array compaction, sorting, segmented prefix-sums, and broadcasting; it also provides a simple use of balanced binary trees.
- ***Pointer-jumping*** (or path-doubling) iteratively halves distances in a list or graph. It is used in numerous parallel graph algorithms, and also as a sampling technique.
- Determining the root for each tree node in a rooted-directed forest is a crucial step in handling equivalence classes – such as detecting whether or not two nodes belong to the same component; when the input is a linked list, this algorithm also solves the parallel prefix problem.
- An entire family of techniques of major importance in parallel algorithms is loosely termed ***divide-and-conquer*** – such techniques decompose the instance into smaller pieces, solve these pieces independently (typically through recursion), and then merge the resulting solutions into a solution to the original instance. These techniques are used in sorting, in almost any tree-based problem, in a number of computational geometry problems (finding the closest pair, computing the convex hull, etc.), and are also at the heart of fast transform methods such as the FFT. The `pardo` primitive in SWARM can be used for implementing such a strategy.
- A variation of the above theme is the ***partitioning strategy***, in which one seeks to decompose the problem into independent subproblems – and thus avoid any significant work when recombining solutions; quicksort is a celebrated example, but numerous problems in computational geometry can be solved efficiently with this strategy (particularly problems involving the detection of a particular configuration in three- or higher-dimensional space).
- Another general technique for designing parallel algorithms is ***pipelining***. In this approach, waves of concurrent (independent) work are employed to achieve optimality.

Built on these techniques, SWARM contains a higher-level library of multicore-optimized parallel algorithms for list ranking, comparison-based sorting, radix sort and spanning tree. In addition, SWARM application example codes include efficient implementations for solving combinatorial problems such as minimum spanning tree [3], graph decomposition [8], breadth-first-search [9], tree contraction [10] and maximum parsimony [7].

## Related Entries

- ▶ [Cilk](#)
- ▶ [OpenMP](#)
- ▶ [Parallel Skeletons](#)

## History

The SWARM programming framework is a descendant of the symmetric multiprocessor (SMP) node library component of SIMPLE [4].

## Bibliography

1. Aggarwal A, Vitter J (1988) The input/output complexity of sorting and related problems. *Commun ACM* 31:1116–1127
2. AMD Multi-Core Products (2006), <http://multicore.amd.com/en/Products/>
3. Bader DA, Cong G (2004) A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). In: Proceedings of the international parallel and distributed processing symposium (IPDPS 2004), Santa Fe, NM, April 2004
4. Bader DA, JáJá J (1999) SIMPLE: a methodology for programming high performance algorithms on clusters of symmetric multiprocessors (SMPs). *J Parallel Distrib Comput* 58(1):92–108
5. Bader DA (2006) SWARM: a parallel programming framework for multicore processors, <https://sourceforge.net/projects/multicore-swarm>
6. Bader DA, Agarwal V, Madduri K (2007) On the design and analysis of irregular algorithms on the Cell processor: a case study of list ranking. In: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2007), Long Beach, CA
7. Bader DA, Chandu V, Yan M (2006) ExactMP: an efficient parallel exact solver for phylogenetic tree reconstruction using maximum parsimony. In: Proceedings of the 35th International Conference on Parallel Processing (ICPP), Columbus, OH, August 2006
8. Bader DA, Illendula AK, Moret BME, Weisse-Bernstein N (2001) Using PRAM algorithms on a uniform-memoryaccess shared-memory architecture. In: Brodal GS, Frigioni D, Marchetti-Spaccamela A (eds) Proceedings of the 5th international workshop on algorithm engineering (WAE 2001), volume 2141 of lecture notes in computer science. Springer-Verlag, Århus, Denmark, pp 129–144
9. Bader DA, Madduri K (2006) Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In: Proceedings of the 35th international conference on parallel processing (ICPP), IEEE Computer Society, Columbus, OH, August 2006
10. Bader DA, Sreshta S, Weisse-Bernstein N (2002) Evaluating arithmetic expressions using tree contraction: a fast and scalable parallel implementation for symmetric multiprocessors (SMPs). In: Sahni S, Prasanna VK, Shukla U (eds) Proceedings of the 9th international conference on high performance computing (HiPC 2002), volume 2552 of lecture notes in computer science. Bangalore, India, Springer-Verlag, December 2002, pp 63–75
11. Barroso LA, Gharachorloo K, McNamara R, Nowatzky A, Qadeer S, Sano B, Smith S, Stets R, Verghese B (2000) Piranha: a scalable architecture based on single-chip multi-processing. *SIGARCH Comput Archit News* 28(2):282–293
12. Helman DR, JáJá J (1999) Designing practical efficient algorithms for symmetric multiprocessors. In: Algorithm engineering and experimentation (ALENEX'99), volume 1619 of lecture notes in computer science, Springer-Verlag, Baltimore, MD, January 1999, pp 37–56
13. Multi-Core from Intel – Products and Platforms (2006) <http://www.intel.com/multi-core/products.htm>
14. International Technology Roadmap for Semiconductors (2004), <http://itrs.net>, 2004 update
15. Kahle JA, Day MN, Hofstee HP, Johns CR, Maeurer TR, Shippy D (2005) Introduction to the cell multiprocessor. *IBM J Res Dev* 49(4/5):589–604
16. Kongetira P, Aingaran K, Olukotun K (2005) Niagara: a 32-way multithreaded Sparc processor. *IEEE Micro* 25(2):21–29
17. Ladner R, Fix JD, LaMarca A (1999) The cache performance of traversals and random accesses. In: Proceedings of the 10th annual symposium discrete algorithms (SODA-99), ACM-SIAM, Baltimore, MD, pp 613–622
18. Ladner RE, Fortna R, Nguyen B-H (2002) A comparison of cache aware and cache oblivious static search trees using program instrumentation. In: Fleischer R, Meineche-Schmidt E, Moret BME (eds) Experimental algorithms, volume 2547 of lecture notes in computer science, Springer-Verlag, Berlin Heidelberg, pp 78–92
19. Mascagni M, Srinivasan A (2000) Algorithm 806: SPRNG: a scalable library for pseudorandom number generation. *ACM Trans Math Softw* 26(3):436–461

## Switch Architecture

JOSÉ FLICH  
Technical University of Valencia, Valencia, Spain

## Synonyms

[Router architecture](#)

## Definition

The switch architecture defines the internal organization and functionality of components in a switch. The switch is in charge of forwarding units of information from the input ports to the output ports.

## Discussion

High-performance computing systems, like clusters and massively parallel processors (MPPs), rely on the use of an efficient interconnection network. As the number of end nodes increases to thousands, or even larger sizes, the network becomes a key component since it must provide low latencies and high bandwidth at a moderate cost and power consumption. The basic components of a network are switches/routers, links, and network interfaces. The interconnection network efficiency largely depends on the switch design.

Prior to defining and describing the switch architecture concept, it is worth differentiating between router and switch. Typically, the router is the basic component in a network that forwards messages from a set of input ports to a set of output ports. The router usually negotiates paths and computes the output ports messages need to take. Therefore, the router has some intelligence and adapts to the varying conditions of the network. Indeed, the router concept comes from Wide Area Networks (WANs) where the switching devices must be smart enough to adapt to the varying topological conditions. In high-performance interconnection networks, typically found in cluster cabinets, connecting massively parallel processors, and nowadays even inside a chip, switching devices are also needed. However, differently from the WAN environment, these devices, although have some intelligence and can make critical decisions, they have no capabilities to negotiate the paths and to adapt to the varying conditions. Indeed, usually the topology is expected not to change, thus no need for such negotiation. This is the reason why these devices are also known as switches rather than routers. However, both terms are used with no clear differentiation by the community and thus, both become acceptable. This entry is related to switch architecture, although it can be seen also as *router architecture*.

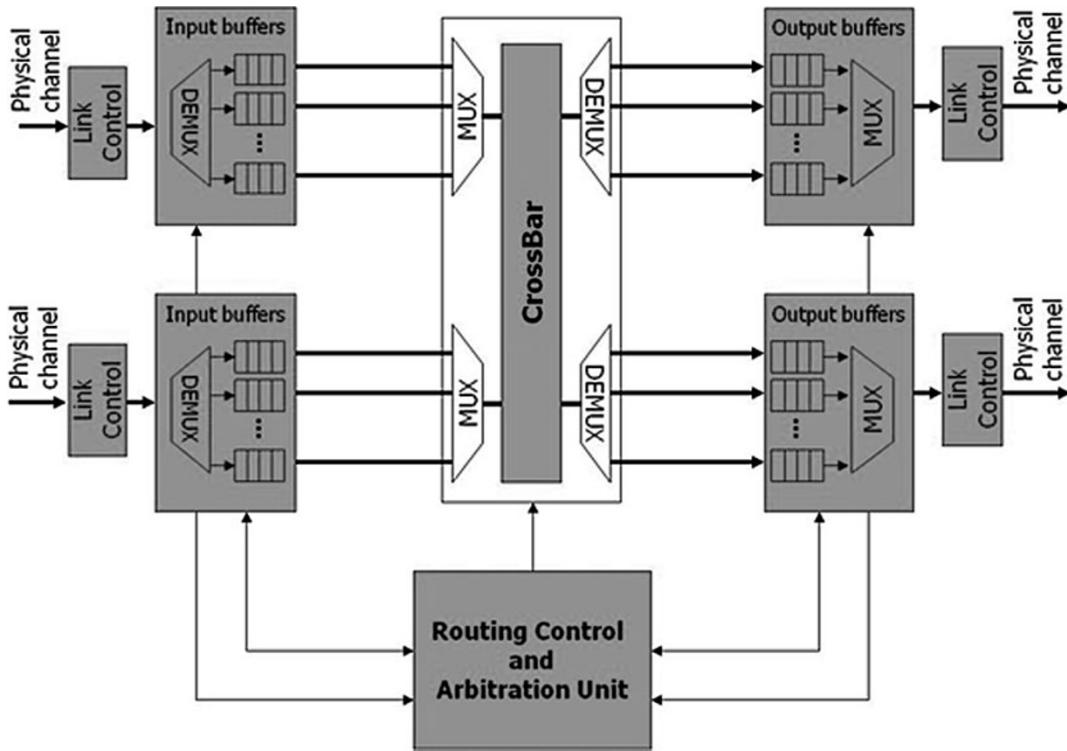
The switch architecture defines the internal organization and functionality of a switch or router. Basically, the switch has a set of input ports where data messages

come in and a set of output ports where data messages are delivered. The way internal components are used and connected between them is defined by the switch architecture.

The switch architecture largely depends on the switching technique (see ►[Switching Techniques](#)) used; thus, we may find switch architectures implementing *store and forward switching* that greatly vary from switches implementing *wormhole switching*. Most of the current modern routers and switches, used in high-performance networks, implement *cut-through switching*, or some variant. This entry focuses on such architectures, mainly in wormhole (WH) switches and virtual cut-through (VCT) switches. Although there are basic differences between them, that affect the switch architecture, there are commonalities as both rely on the same form of switching. In the next section, a canonical switch architecture including all the commonalities found in current switches is provided. Then, different components are reviewed and alternative switch organizations are described.

## Canonical Switch Architecture

[Figure 1](#) shows the organization of a basic switch architecture. The switch is made up of a set of identical input ports, connected to input physical channels. Each input port uses a link control module to adapt messages coming through the physical channel to the internal switch. Each message is then stored in a buffer at the input port. The buffer can be made of different queues, each one usually associated to a virtual channel. Each message is then routed (computing the appropriate output port to take) at the Routing control unit. Once the output port is computed, the message may cross the internal crossbar of the switch, thus reaching the output port. To do so, the message has to win the access to the output port since different messages may compete for the same port. In addition, the message has to compete to get access to an available virtual channel. Both are resolved by the arbitration unit. Once the access is granted, the message crosses the crossbar and is written at the corresponding queue at the output port. Each message, again, needs to compete with other messages in order to win the access to the physical channel. Prior to reaching the physical channel the message passes through the link controller logic.



**Switch Architecture. Fig. 1** Basic switch architecture

As an alternative view, the resources found in the switch can be divided into two separate blocks: the *data plane* (or *datapath*) and the *control plane*. Basically, the data plane is the set of resources messages use while being forwarded (storage and movement) through the switch. The control plane is made of the resources used to make decisions at the switch, like the routing control unit and the arbitration unit.

### Alternative Switch Architectures

Taking this basic switch architecture as the starting point, several alternative architectures can be derived, some of them better suited for a particular switching mechanism. One of the key issues in the switch architecture is the location of the buffering resources. Regarding this, switch architectures can be classified as:

**Output-Queued (OQ) switch architecture.** In this architecture buffers only exist at the output ports and thus no buffering exists at the input port. Thus, whenever a message arrives at the switch it must be sent directly to the appropriate output port. An  $N \times N$  OQ switch requires only  $N$  memories, one per output

port. As packets are mapped to the memory associated with the requested output port, HOL blocking is totally eliminated (see ►Congestion Management Entry); thus, this organization achieves maximum switch efficiency. However, internal speedup is required to handle the worst-case scenario without dropping messages, allowing all the input ports to transmit packets to the same output port at the same time. In particular, output queues must either implement multiple write ports or use a higher clock frequency. A speedup of  $N$  is required in OQ switches in the worst case. Unfortunately, providing such internal speedup is not always viable.

**Input-Queued (IQ) switch architecture.** Buffers only exist at input ports and not at output ports. In this architecture the required internal bandwidth does not increase with the number of ports and the switch can be designed with the same bandwidth as the link. However, a switch designed with this organization may face low performance due to contention/congestion at the output ports. It is well known that such switches achieve only 58% of maximum efficiency under uniformly distributed requests [3]. This is mainly

due to the HOL blocking problem. One solution to eliminate the HOL blocking issue in IQ switches is the use of  $N$  queues at every input port, mapping the incoming message to a queue associated with the requested output port. This technique is known as Virtual Output Queuing (VOQ) [4]. However, it increases the queue requirements quadratically with the number of ports. Therefore, as the number of ports increases, this solution becomes too expensive.

**Combined-Input-Output-Queued (CIOQ) switch architecture.** With this organization, the contention/congestion problem found in IQ switches is alleviated or even eliminated, since messages can be also stored at the output side of the switch. In this architecture some moderate internal speedup is used, as the internal bandwidth is higher than the aggregate link bandwidth. A speedup of two is usually enough to compensate the performance drop produced by HOL blocking. Speedup can be implemented by using internal datapaths with higher transmission frequencies or wider transmission paths. However, as the external link bandwidth increases, sustaining the speedup may become difficult.

**Buffered-Crossbar (BC) switch architecture.** This organization uses a memory at every crossbar cross-point. An input link is connected to  $N$  memories, each one connected to a different output port. By design, the BC organization implements internal speedup, as many inputs can forward a packet to the same output at the same time. Additionally, such memory organization eliminates the HOL blocking (every packet is mapped to the memory associated with the requested output port). As a consequence, the BC organization requires low-cost arbiters per output port. However, the problem with such organization is that the number of memories increases quadratically with the number of ports ( $N^2$ ), thus limiting scalability.

When focusing on the switching device (the crossbar) different alternatives exist. The solution used in the basic example is the most frequently implemented one, where a crossbar connecting every input to every possible output is used. The crossbar has inherited bandwidth as it is able to connect an input to every output (broadcast communication). There are, however, other solutions, like using a centralized buffer and using a bus.

In addition, the connection of the input ports to the crossbar, and the crossbar to the output ports can be implemented in different ways. In the example

multiplexers and demultiplexers are used at both sides of the crossbar. This is done to reduce the crossbar complexity (that increases quadratically with the number of ports). Different alternative configurations arise when these devices are removed from one or both sides of the crossbar. In the case multiplexers are removed, the input speedup of the switch is increased, since different messages can be forwarded through the crossbar from the same input port. If demultiplexers are removed then the output speedup of the switch is increased, since different messages can be written to the same output port at the same time. Obviously, output buffers are required when implementing output speedup.

## Input Buffer Organization

An important component of the switch architecture is the buffer organization, mostly when virtual channels are implemented. The way memory resources are assigned to queues will impact on the performance of the switch. This issue is highly related to the flow control protocol (see ►Flow Control). Buffer partitioning can be designed in several ways. The first one is to combine all the buffers across the entire switch (a single memory). In that case, there is no need for a switching element. The benefit of this approach is the flexibility in dynamically allocating memory across the input ports. The problem, however, is the high memory bandwidth required.

The second way to organize buffers is by partitioning them by physical input ports and dynamically assigning them to virtual channels of the same input port. The third way is to partition the buffers by virtual channel, providing a separate memory for each virtual channel at each port. This, however, may lead to high cost and poor memory utilization.

Another important aspect of buffer organization is the way flits are stored. Such memories require data structures to keep track of where flits are located, and to manage free slots. Two buffer organizations are mostly used: circular buffers and linked lists. Circular buffers are used when memory is statically assigned to a queue (virtual channel) and have low implementation overheads. However, a linked list is used when memory is assigned dynamically and has, in turn, higher implementation overhead.

## Pipelined Organization

It is common to design a switch as a pipelined datapath. Such designs allow for high clock frequencies of the switch and, thus, high throughput. On every cycle, a different flit (see ►Flow Control Entry) from the same message may be processed at each stage. Typically, five stages are conceived for the switches:

- Input Buffer (IB) stage, where the flit is stored at the input port of the switch.
- Routing computation (RC) stage, where the output port is computed for the message.
- Switch allocation (SA) stage, where the access to the output port is granted among all the requesting messages. Also, selection of the virtual channel to use is performed.
- Switch transversal (ST) stage, where the flit crosses the crossbar and reaches the output.
- Output Buffer (OB) stage, where the flit is stored at the output port of the switch.

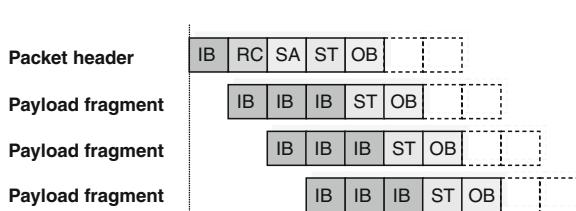
In the ideal case (no contention experienced within the switch) the flit advances through the pipelined switch architecture as shown in Fig. 2. The figure shows the case for a switch implementing virtual cut-through switching where arbitration is performed once per packet.

The first flit is the header flit of the message. At the first cycle the header flit is stored at the input port (IB stage) in a virtual channel. The header includes the identifier of the virtual channel to use. At the second cycle, the header flit is processed at the RC stage so the output port for the message is computed. The output port identifier is associated with the input virtual channel, as this output port will be used by all the flits of the message. At the same cycle the second flit of the message (payload flit) is stored at the input port (IB stage). At

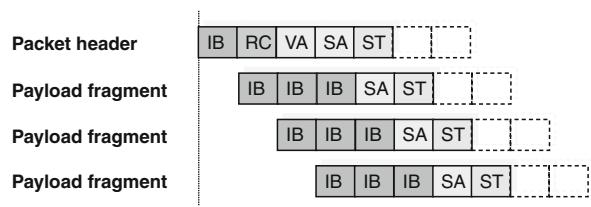
the third cycle the SA stage is executed to get access to the requested output port. A valid virtual channel for the message at the next switch is also requested. On success (there is an available virtual channel and the output port is granted) the header flit crosses the crossbar at the next cycle, followed in the following cycles by the rest of payload flits. All the flits use the same virtual channel and output port. Notice also that resources (access to the output port) are granted per message, so flit multiplexing in the crossbar may not occur. Upon crossing the crossbar, the flits are stored at the output port in the corresponding queue (OB stage). The tail flit of a message will have a different treatment since the connection of the input port to the output port will be broken. Notice that the RC and SA stages are performed only once per packet.

A typical wormhole switch architecture differs in some stages. First, it is not common to have a buffer at the output port, thus using an IQ switch approach. Second, virtual channel allocation and port allocation are usually performed in separate stages. Therefore, a new stage appears, referred to as Virtual channel allocation (VA) used only for header flits, and the SA stage only intended for requesting the output port. Also, output port usually is granted flit by flit. Figure 3 shows an example of a five-stage pipelined wormhole switch.

The pipeline design may experience stalls. Stalls may happen due to different reasons (a virtual channel is not available in the SA stage, the RC stage does not find a free output port, ...). In all these cases, the switch must be designed accordingly. Also, the flow control (see ►Flow Control) must be aware of the stalls and thus backpressure the previous switch to avoid buffer overflows. For a detailed analysis of pipelined switches and stall treatments, the reader is referred to [1] (Chapter 16).



**Switch Architecture. Fig. 2** Typical five-stage pipelined switch design for a virtual cut-through switch

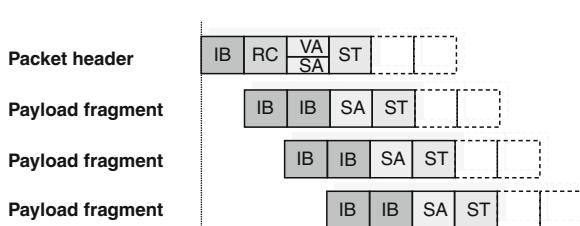


**Switch Architecture. Fig. 3** Typical five-stage pipelined switch architecture for a wormhole switch

This basic pipelined architecture can be enhanced with different techniques, most of them trying to reduce the number of stages, thus also reducing the delay of traversing a switch. Routers with three or even two stages can be found in the literature. Also, single-cycle switches are common in some environments, like networks-on-chip (NoCs) for low-end systems-on-chip (SoCs). Obviously, such reduction in the number of stages should be achieved by not incurring in an excessive increase of the cycle time. Two basic procedures exist to reduce the number of stages, the first one by using speculation and performing actions in parallel, and the second one by performing computations ahead of time to remove the operation from the critical path.

Virtual channel allocation can be performed speculatively and in parallel with switch allocation in wormhole switching. Notice that this is done only for header flits. If both resources are obtained (the virtual channel is successfully assigned and the output port is granted) then the wormhole switch saves one cycle. If any of the two fails (or both fail) then the pipeline stalls and at the next cycle the operation is repeated. [Figure 4](#) shows the case.

A further reduction in cycles would be to speculatively send the flit through the crossbar to the output port (ST) at the same time. To achieve this, internal speedup is required. Finally, the output port computation can be performed at the previous switch and stored at the head flit. So, when the flit reaches the next switch the output port to take is already computed and thus, there is no need for the RC stage. In that case, the output port computed for the next switch can be done in parallel with the VA stage, thus further reducing the pipeline depth.



**Switch Architecture. Fig. 4** Four-stage pipelined wormhole switch design

## High-Radix Switch Architectures

As identified in [\[5\]](#), during the last decades, the pin bandwidth of a switch has increased exponentially (from 64 Mb/s of the Torus routing chip in 1986 to the 1Tb/s of the recent Velio 3003). This is due to the increase in the signaling speed and the increase in the number of signals. In this sense, high-radix switches with narrow channels are able to achieve lower packet latencies than low-radix switches with wide channels. The explanation is simple. With high-radix switches the hop count in the network decreases. Additional benefits from high-radix switches are a lower cost and lower power consumption (as the total number of switches and links to build a network is reduced). Following this trend, there are some proposals for high-radix switch architectures, [\[5–7\]](#).

However, designing high-radix switches presents major challenges. The most important one is to keep a high switch efficiency with an affordable cost. The cost of a high-radix switch will largely depend on three key components: memory resources, arbiter logic, and internal connection logic. Depending on the location of memories in the switch, different switch organizations (memory and crossbar capabilities and their interconnects) have been used. In some of them, the number of memories increases quadratically with the number of ports. Also, arbiters and crossbars must cope with more candidates and connections, and for that reason become expensive. As an example, in on-chip networks, the use of high-radix switches is not appropriate due to the increase in power consumption and reduction in switch-operating frequency [\[8\]](#).

## Related Entries

- ▶ [Flow Control](#)
- ▶ [Switching Techniques](#)

## Bibliographic Notes and Further Reading

Two basic books exist for interconnection networks. Both describe in detail the concept of switch architecture. The first one, [\[2\]](#) describes a wide range of routers, some with wormhole switching (Intel Teraflops router,

Cray 3TD and 3TE routers, Reliable router and SGI spider), others with virtual cut-through switching (Chaos router, Arctic router, R2 router, Alpha 21364 router), and others with circuit switching (Intel iPSC Direct Connect Module). Also, the book describes the Myrinet switch.

The other book [1], provides as a case study the Alpha 21364 router and the IBM Colony router.

In on-chip networks building an efficient switch is critical. In addition to delay constraints, designing an on-chip switch has also power consumption and area limitations. In [9] basic design rules for building an on-chip router are provided. Also, literature is populated with many switch/router architectures for on-chip networks.

## Bibliography

1. Dally W, Towles B (2004) Principles and practices of interconnection networks. Morgan Kaufmann, San Francisco, CA
2. Duato J, Yalamanchili S, Ni N (2002) Interconnection networks: an engineering approach. Morgan Kaufmann, San Francisco, CA
3. Karol MJ et al (1987) Input versus output queueing on a space-division packet switch. IEEE Trans Commun COM-35(12):1347–1356
4. Tamir Y, Frazier GL (1988) High-performance multi-queue buffers for vlsi communications switches. SIGARCH Comput Archit News 16(2):343–354
5. Kim J, Dally WJ, Towles B, Gupta AK (2005) Microarchitecture of a high-radix router. In: 32nd Annual International Symposium on Computer Architecture (ISCA '05), Madison, WI, pp 420–431
6. Scott S, Abts D, Kim J, Dally WJ (2006) The blackwidow high-radix clos network. In: Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA), The Washington, DC, June 2006
7. Mora G, Flich J, Duato J, López P, Baydal E, Lysne O (2006) Towards and efficient switch architecture for high-radix switches. In: Proceedings of ANCS 2006, San Jose, CA
8. Pullini A, Angiolini F, Murali S, Atienza D, De Micheli G, Benini G (2007) Bringing NOCs to 65 nm. IEEE Micro 27(5):75–85
9. de Micheli G, Benini L (2006) Networks on chips: technology and tools. Morgan Kaufmann, San Francisco, CA

## Switched-Medium Network

### ► Buses and Crossbars

## Switching Techniques

SUDHAKAR YALAMANCHILI

Georgia Institute of Technology, Atlanta, GA, USA

### Definition

Switching techniques determine how messages are forwarded through the network. Specifically, these techniques determine how and when buffers and switch ports of individual routers are allocated and released and thereby the timing with which messages or message components can be forwarded to the next router on the destination path.

### Discussion

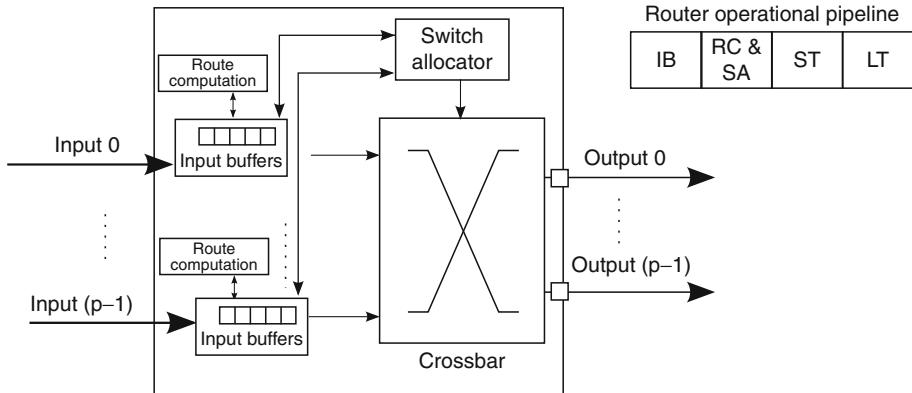
#### Introduction

This section introduces basic switching techniques used within the routers of multiprocessor interconnection networks. Switching techniques determine *when* and *how* messages are forwarded through the network. These techniques determine the granularity and timing with which resources such as buffers and switch ports are requested and released and consequently determine the blocking behavior of routing protocols that utilize them in different network topologies. As a result, they are key determinants of the deadlock properties of routing protocols. Further, their relationship to flow control protocols and traffic characteristics significantly impact the latency and bandwidth characteristics of the network.

#### A Generic Router Model

Switching techniques are understood in the context of routers used in multiprocessor interconnection networks. A simple generic router architecture is illustrated in Fig. 1. This router microarchitecture presents to messages a four stage pipeline comprised of the following stages.

- *Input Buffering (IB)*: Message data is received into the input buffer.
- *Route Computation (RC) and Switch Allocation (SA)*: Based on the message destination, a switch output port is computed, requested, and allocated.



**Switching Techniques.** Fig. 1 A generic router model

- *Switch Traversal (ST)*: Message data traverses the switch to the output buffer.
- *Link Traversal (LT)*: The message data traverses the link to the next router.

The end-to-end latency experienced by a message depends on how the switching techniques interact with this pipeline. This generic router architecture has been enhanced over the years with virtual channels [1], speculative operation [2, 3], flexible arbiters, effective channel and port allocators [4], deeper pipelines, and a host of buffer management and implementation optimizations (e.g., see [5, 6]). The result has been a range of router designs with different processing pipelines.

For the purposes of this discussion, here it is assumed that all of the pipeline stages take the same time – one cycle. This is adequate to define, distinguish, and compare properties of basic switching techniques. The following section addresses some basic concepts governing the operation and implementation of switching techniques based on the generic router model shown in Fig. 1. The remainder of the section is devoted to a detailed presentation of alternative switching techniques.

### Basic Concepts

The performance and behavior of switching techniques are enabled by the low-level flow control protocols used for the synchronized transfer of data between routers. Flow control determines the granularity with which data is moved through the network and consequently when routing decisions can be made, when switching

operations can be initiated, and how (at what granularity) data is transferred.

Flow control is the synchronized transfer of a unit of data between a sender and a receiver and ensures the availability of sufficient buffering at the receiver to avoid the loss of data. Selection of the unit of data transfer is based on a few simple concepts. A message is partitioned into fixed-length *packets*. Packets are individually routable units of data and are comprised of control bits packaged as the header and data bits packaged as the body. Packets are typically terminated with some bits for error detection such as a checksum. The header contains destination information used by the routers to select the onward path through the network. The packet as a whole is partitioned into fixed size units corresponding to the unit of transfer across a link or across a router and is referred to as a flow control digit or *flit* [7]. A flit becomes the unit of buffer management for transmission across a physical channel. Many schemes have been developed over the years for flow control including on-off, credit-based, sliding window, and flit reservation. The physical transfer of a flit across a link may in fact rely on synchronized transfers of smaller units of information. For example, consider flit sizes of 4 bytes and a physical channel width of 8 bits. The transfer of a flit across the physical link requires the synchronized transfer of 8-bit quantities referred to as a physical digit or *phit* [4] using phit-level flow control. In contrast to flits which represent units of buffer management, phits correspond to quantities reflecting a specific physical link implementation. Both flit-level and phit-level flow control are atomic in the sense that in the absence of

errors, all transfers will complete successfully and are not interleaved with other transfers. While phit sizes between chips or boards tend to be small, e.g., 8–32 bits, phit sizes on-chip can be much larger, e.g., 128 bits. Typical message sizes can range from 8–12 bytes (control messages) to 64–128 bytes (for example cache lines), to much larger sizes in message-passing parallel architectures. The preceding hierarchy of units is illustrated in Fig. 2 along with an example of a packet format [4, 8]. The actual content of a packet header will depend on the specifics of an implementation such as the routing protocol (e.g., destination address), flow control strategy (e.g., credits), use of virtual channels (e.g., virtual channel ID), and fault tolerance strategy (e.g., acknowledgements).

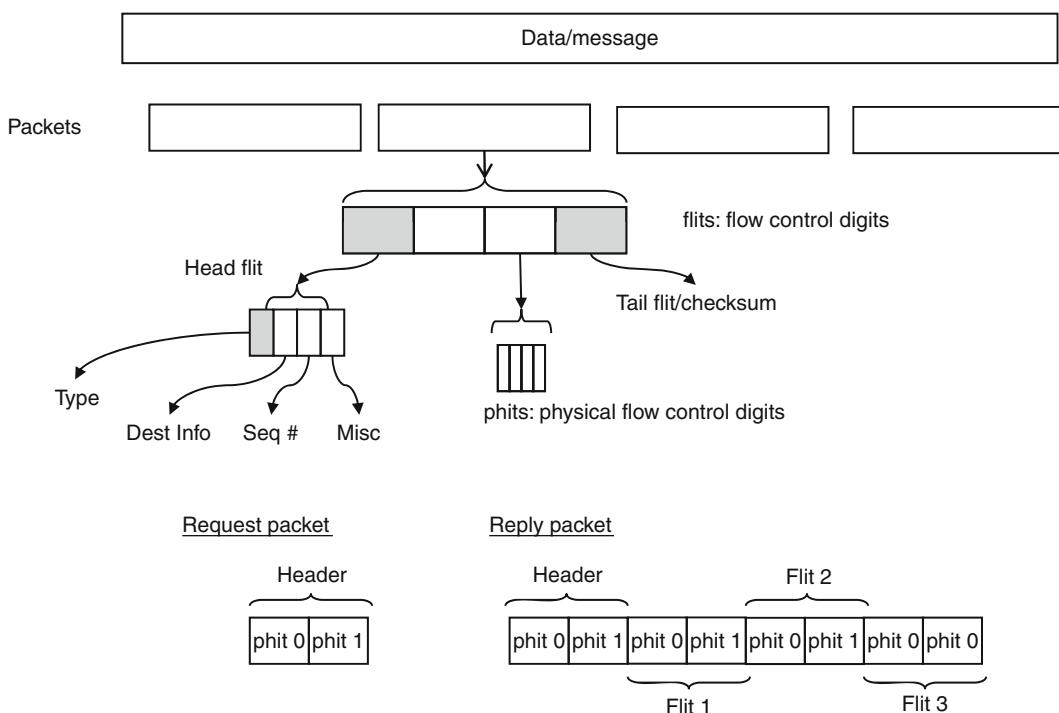
Switching techniques determine when messages are forwarded through the network and differ in the relative timing of flow control operations, route computation, and data transfer. High-performance switching techniques seek as much overlap as possible between these operations while reliable communication protocols may seek less concurrency between these operations in the interest of efficient error recovery. For example, one can

wait until the entire packet is received before a request is made for the output port of the router's switch (packet switching). Alternatively, the request can be made as soon as all flits corresponding to the header are received (cut-through switching) but before the rest of the packet has been received.

The remainder of this section will focus on the presentation and discussion of switching techniques under the assumption of a flit-level flow control protocol and a supporting phit-level flow control protocol. It is common to have the flit size the same as the phit size.

### Basic Switching Techniques

As one might expect, switching techniques have their roots in traditional digital communication and have evolved over the years to address the unique requirements of multiprocessor interconnection networks. For the purposes of comparison, the no-load latency is computed for an  $L$ -bit message. The phit size and flit size are assumed to be equivalent and equal to the physical data channel width of  $W$  bits, which is also the width of the internal datapath of the router. The routing header is assumed to be one flit, thus the message size is  $L + W$



Switching Techniques. Fig. 2 Basic concepts and an example packet format

bits. A router can make a routing decision in one cycle and a flit can traverse a switch or link in one cycle as described with respect to Fig. 1.

### Circuit Switching

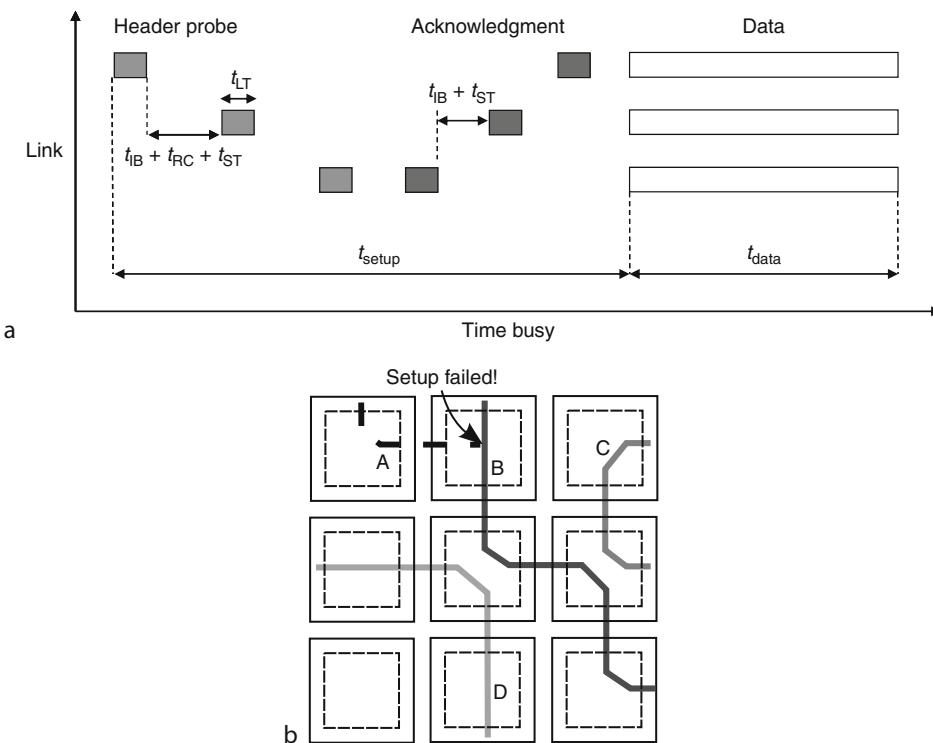
Circuit switching evolved from early implementations in telephone switching networks. In circuit switching, a physical path from the source network interface to the destination network interface is reserved prior to the transmission of the data. The source injects a routing packet commonly called a probe into the network. The probe is routed to the destination as specified by the routing protocol reserving the physical channels and switch ports along the way. An acknowledgement is returned to the source node to confirm path reservation. Figure 3b shows several circuits that have been set up and one in the process of being set up. While circuits B, C, D have been set up, circuit A is blocked from being set up by circuit B.

The base no-load latency of a circuit-switched message is determined by the sum of the time to set up a path and the time to transmit data. For a circuit that

traverses  $D$  routers to the destination and operates at  $B$  Hz, the no-load message latency can be represented as follows.

$$\begin{aligned} t_{circuit} &= t_{setup} + t_{data} \\ t_{setup} &= D[t_{RC} + 2(t_{IB} + t_{ST} + t_{LT})] \\ t_{data} &= \frac{1}{B} \left\lceil \frac{L}{W} \right\rceil \end{aligned} \quad (1)$$

The notation in the expression follows the generic router model shown in Fig. 1, and the expression does not include the time to inject the probe into the router at the source (a link traversal) or the time to inject the acknowledgement into the router at the destination. The subscripts correspond to the pipeline stages illustrated in Fig. 1. For example,  $t_{RC}$  is the time taken to compute the output port to be traversed by the message at a router. This computation is carried out in the same cycle as the process of requesting and allocating a switch port, and therefore the switch allocation (SA) time is hidden and does not appear in the expression. The terms  $t_{IB}$ ,  $t_{ST}$ , and  $t_{LT}$  represent the times for input buffering, switch traversal, and link traversal,



**Switching Techniques.** Fig. 3 Circuit switching. (a) Time-space utilization across two routers, (b) an example

respectively, experienced by the probe and the acknowledgement. The factor of 2 in the expression for path setup reflects the forward progress of the probe and the return progress of the acknowledgement – assuming the acknowledgement traverses the same path in the reverse direction. Note the acknowledgements do not have to be routed and therefore do not experience any routing delays at intermediate routers. A time-space diagram in Fig. 3a depicts the setup and operation of a circuit that crosses two routers.

Nominally, the routing probe contains the destination address and additional control bits used by the routing protocol and is buffered at intermediate routers where it is processed to reserve links and set router switch settings. On reaching the destination, an acknowledgement packet is transmitted back to the source. The hardware circuit has been set up and data can be transmitted at the full speed of the circuit. Flow control for data transfer is exercised end-to-end across the circuit and the flow control bandwidth (rate of signaling) should be at least as fast as the transmission speeds to avoid slowing down the hardware circuit. When transmission is complete, a few control bits traversing the circuit from the source release link and switch resources along the circuit. These bits may take the form of a small packet or with suitable channel design and message encoding, may be transmitted as part of the last few bits of the message data. Routing and data transmission functions are disjoint operations where all switches on the source-destination path are set prior to the transmission of any data.

Circuit switching is generally advantageous when messages are infrequent and long compared to the size of the routing probe. It is also advantageous when inter-node traffic exhibits a high degree of temporal locality. After a path has been set up, subsequent messages to the same destination can be transmitted without incurring the latency of path set-up times. The inter-node throughput is also maximized since once a path has been set up, messages to the destination are not routed and do not block in the network. The disadvantages are the same as those typically associated with most reservation-based protocols. When links and switch ports are reserved for a duration they prevent other traffic from making progress if they need to use any of the resources reserved by the established circuit. In particular, a probe can be blocked during circuit set up while

waiting for another circuit to be torn down. The links reserved by the probe up to that point can similarly prevent other circuits from being established.

Wire delays place a practical limit on the speed of circuit switching as a function of system size, motivating the development of techniques to mitigate or eliminate end-to-end wire delay dependencies. One set of techniques evolved around pipelining multiple bits on the wire essentially using a long latency wire as a deeply pipelined transmission medium. Such techniques have been referred to as *wave pipelining* or *wave switching* [9–11]. These techniques maximize wire bandwidth while reducing sensitivity to distance. However, when used in anything other than bit serial channels, pragmatic constraints of signal skew and stable operation across voltage and temperature remain challenges to widespread adoption. Alternatively, both the wire length issue and blocking are mitigated by combining circuit switching concepts with the use of virtual channels. Virtual channel buffers at each router are reserved by the routing probe setting up a pipelined virtual circuit from source to destination in a manner referred to as pipelined circuit switching [12]. This approach increases physical link utilization by enabling sharing across circuits although the available bandwidth to each circuit is now reduced. This approach was also used in the Intel iWarp chip that was designed to support systolic communication through *message pathways*: long-lived communication paths [13]. Rather than set up and remove network paths each time data are to be communicated, paths through the network persist for long periods of time. Special messages called *pathway begin markers* are used to reserve virtual channels (referred to as *logical channels* in iWarp) and set up interprocessor communication paths. On completion of the computation, the paths are explicitly removed by other control messages.

## Packet Switching

In circuit switching, routing and data transfer operations are separated. All switches in the path to the destination are set a priori, and all data is transmitted in a burst at the full bandwidth of the circuit. In packet switching, the message data is partitioned into fixed-sized packets. The first few bytes of a packet contain routing and control information and are collectively

referred to as the *packet header* while the data is contained in the packet body. Each packet can now be independently routed through the network. Flow control between routers is at the level of a complete packet. A packet cannot be forwarded unless buffer space for a complete packet is available at the next router. A packet is then transferred in its entirety across a link to the input buffer of the next router. Only then is the header information extracted and used by the routing and control unit to determine the candidate output port. The switch can now be set to enable the packet to be transferred to the output port and then stored at the next router before being forwarded. This switching technique is also known as *store-and-forward* (SAF) switching. There is no overlap between flow control, routing operations, and data transfer. Consequently, the

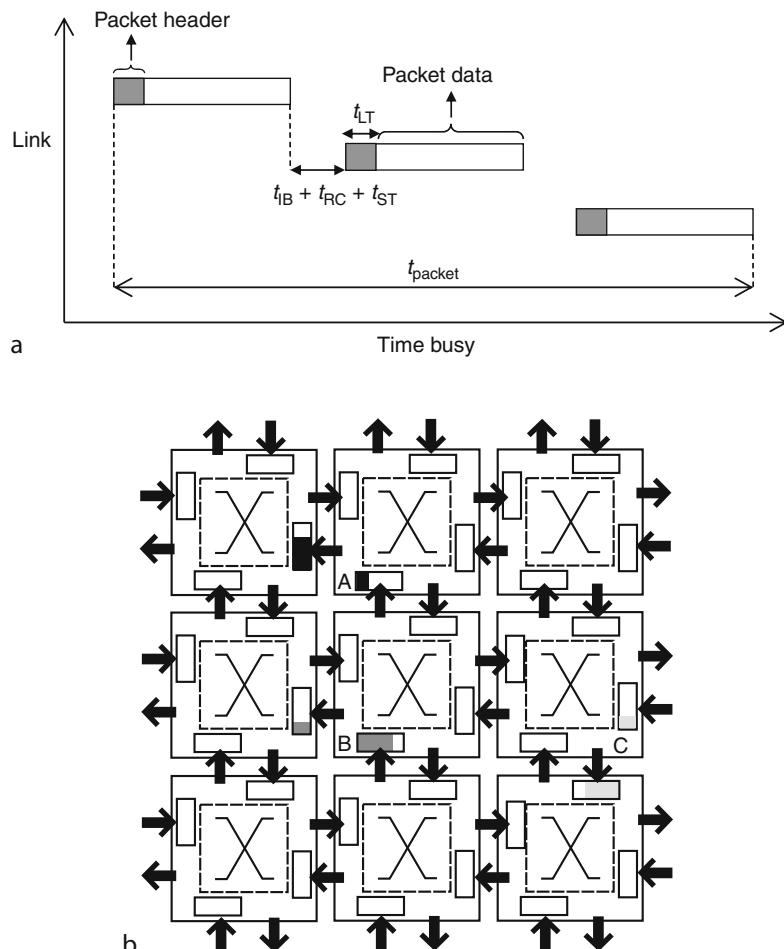
end-to-end latency of a packet is proportional to the distance between the source and destination nodes.

An example of packet switching is illustrated in Fig. 4b (links to local processors are omitted for brevity). Note how packets A, B, and C are in the process of being transferred across a link. Each input buffer is partially full pending receipt of the remainder of the packet before being forwarded.

The no-load latency for a packet transmission across  $D$  routers can be modeled as follows.

$$t_{\text{packet}} = D \left\{ t_{RC} + (t_{IB} + t_{ST} + t_{LT}) \left\lceil \frac{L + W}{W} \right\rceil \right\} \quad (2)$$

This expression does not include the time to inject the packet into the network at the source. The expression follows the router model in Fig. 1 where entire packets must traverse the link or switch at each router.



**Switching Techniques. Fig. 4** Packet switching. (a) Time-space utilization across three links, (b) an example

Therefore end-to-end latency is proportional to the distance between the source and destination.

Packet switching evolved from early implementations in data networks. Relative to circuit switching, packet switching is advantageous when the average message size is small. Since packets only hold resources as they are being used, this switching technique can achieve high link utilization and network throughput. Packets are also amenable to local techniques for error detection and recovery since all data and its associated routing information are encapsulated as a single locally available unit. However, the overhead per data bit is higher – each packet must invest in a header reducing energy efficiency as well as the proportion of physical bandwidth that is accessible to actual data transfer. If a message is partitioned into multiple packets and adaptive routing is employed, packets may arrive at the destination out of order necessitating investments in reordering mechanisms. Packet-switched routers have also been designed with dynamically allocated centralized queues rather than keeping the messages buffered at the router input and output resulting in both cost and power advantages.

### **Virtual Cut-Through (VCT) Switching**

Virtual cut-through (VCT) switching is an optimization for packet switching where in the absence of congestion, packet transfer is pipelined. Like the preceding techniques, VCT has its genesis in packet data networks [14]. Flow control is still at the packet level. However, packet transfer is overlapped with flow control and routing operations as follows. Routing can begin as soon as the header bytes of a packet have arrived at the input buffer and before the rest of the packet has been received. In the absence of congestion, switch allocation and switch traversal can proceed and the forwarding of the packet through the switch as well as flow control requests to the next router can begin. Thus, packet transfer can be pipelined through multiple routers. For example, consider a 128 byte packet with an 8 byte header. After the first 8 bytes have been received, routing decisions and switch allocation can be initiated. If the switch output port is available, then the router can begin forwarding bytes to the output port before the remainder of the packet has arrived and can *cut-through* to the next router. In the absence of blocking in the network, the

latency for the header to arrive at the destination network interface is proportional to the distance between the source and destination. Thereafter a phit can exit the network every cycle. If the header is blocked on a busy output channel at an intermediate router, the packet is buffered at the router – a consequence of the fact that flow control is at the level of a packet. Thus, at high network loads, VCT switching behaves like packet switching. An example of VCT at work is illustrated in Fig. 5b. Packet A is blocked by packet B. Note that packet A has enough buffer space to be fully buffered at the local router. Packet B can be seen to be spread across multiple routers as it is pipelined through the network.

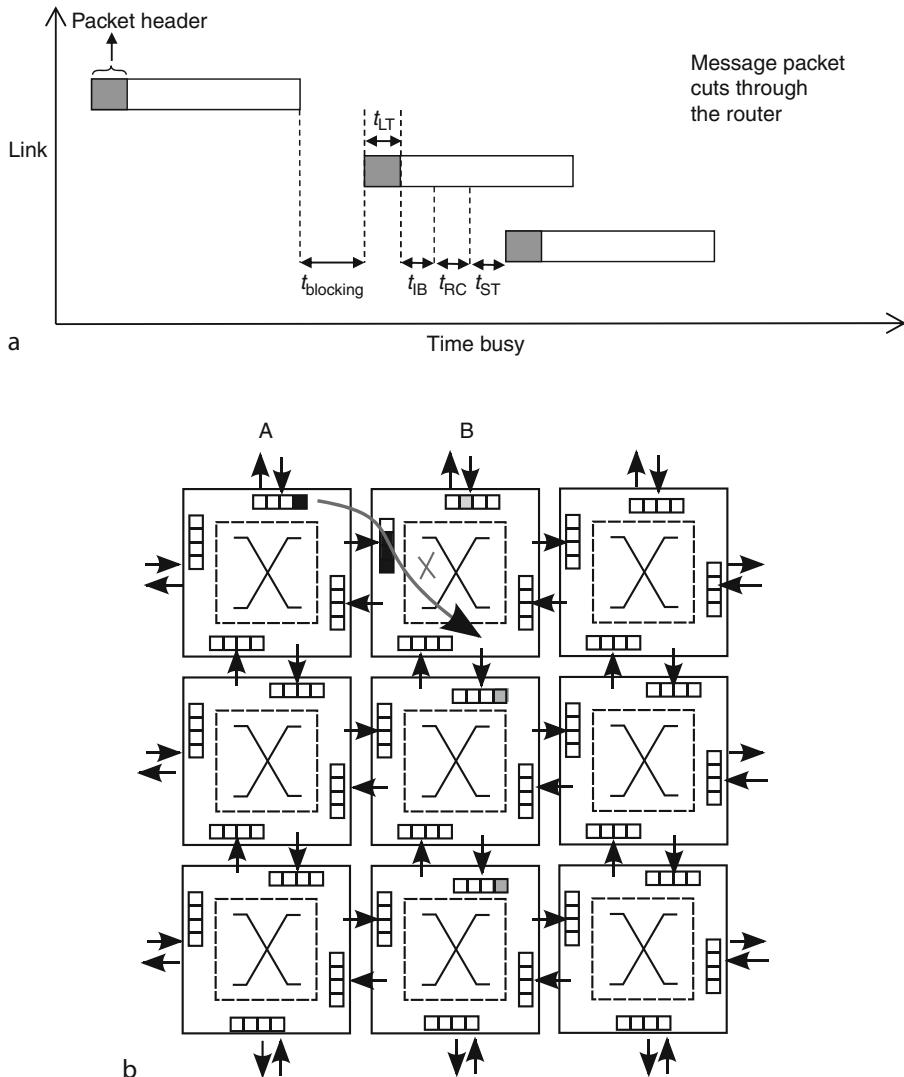
The no-load latency of a message that successfully cuts through  $D$  intermediate routers is captured in the following expression.

$$t_{VCT} = D(t_{IB} + t_{RC} + t_{ST} + t_{LT}) + t_{IB} \left[ \frac{L}{W} \right] \quad (3)$$

This expression does not include the time to inject the packet into the network at the source. The first term in the equation is much smaller than the second term (recall that each value of delay is one pipeline cycle). Therefore, under low load conditions, the message latency is approximately proportional to the packet size rather than distance between source and destination. However, when a packet is blocked, the packet is buffered at the router. Thus, at high network loads, the behavior of VCT approximates that of packet switching. At low loads, the performance improves considerably approaching that of wormhole switching which is described next.

### **Wormhole Switching**

The need to buffer complete packets within a router can make it difficult to construct small, compact, and fast routers. In the multiprocessor machines of the 1980s, interconnection networks employed local node memory as storage for buffering blocked packets. This ejection and re-injection of packets incurred significant latency penalties. It was desirable to keep packets in the network. However, there was insufficient buffering within individual routers. Wormhole switching evolved as a small buffer optimization of virtual cut-through where packets were pipelined through routers. The buffers in each router had enough storage for several flits. When a packet header blocks, the message occupies buffers in several routers. For example, consider



**Switching Techniques. Fig. 5** Virtual cut-through switching. (a) Time-space utilization across three links, (b) an example

the message pattern shown in Fig. 6b with routers with one flit buffers. Message A is blocked by message B and occupies buffers across multiple routers leading to secondary blocking across multiple routers. The time-space diagram illustrates how packets are pipelined across multiple routers significantly reducing the sensitivity of message latency to distance.

When it was introduced [7, 15], the pipelined behavior of wormhole switching led to relatively large reductions in message latency at low loads. Further gains derived from not having to eject messages from the network for storage. The use of small buffers also

has two physical consequences. First, smaller buffers lead to lower access latency and shorter pipeline stage time (see Fig. 1). The smaller pipeline delay enables higher clock rates and consequently high bandwidth routers, for example, 5 GHz in today's Intel TeraFlops router [16]. Second, the smaller buffers also reduce static energy consumption which is particularly important in the context of on-chip routers. However, as the offered communication load increases, messages block in place occupying buffers across multiple routers and the links between them. This leads to secondary blocking of messages that share any of these links which

in turn propagates congestion further. The disadvantage of wormhole switching is that blocked messages hold physical channel resources. Routing information is only associated with a few header flits. Data flits have no routing information associated with them. Consequently, when a packet is blocked in place, packet transmission cannot be interleaved over a physical link without additional support (such as virtual channels – see Section on Virtual Channels) and physical channels cannot be shared. The result is the rapid onset of saturation as offered load increases. Virtual channel flow control was introduced to alleviate this problem.

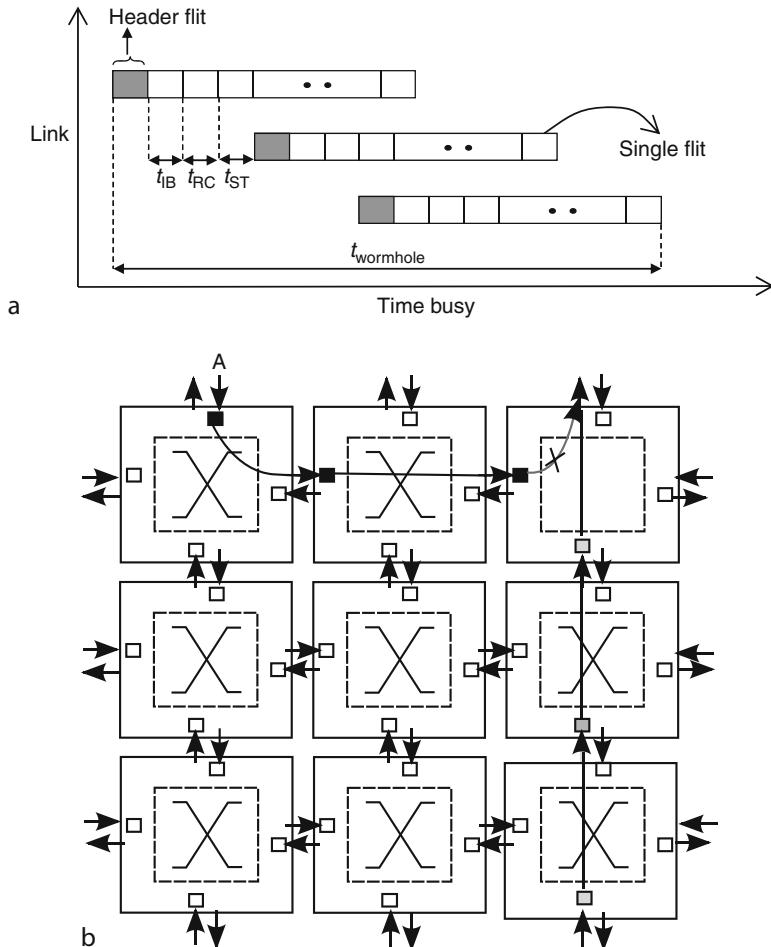
The key deadlock issue is that a single message produces dependencies between buffers across multiple routers. Routing protocols must be designed

to ensure that such dependencies are not composed across multiple messages to produce deadlocked configurations of messages. Deadlock freedom requirements for deterministic routing protocols are described in [7] while proofs for adaptive routing protocols are described in [17–19].

The base latency of a wormhole-switched message crossing  $D$  routers with the flit size equal to the phit size can be computed as follows.

$$t_{\text{wormhole}} = D(t_{IB} + t_{RC} + t_{ST} + t_{LT}) + t_{IB} \left[ \frac{L}{W} \right] \quad (4)$$

This expression does not include the time to inject flits into the network at the source. After the first flit arrives at the destination, each successive flit is delivered



**Switching Techniques. Fig. 6** Wormhole switching. (a) Time-space utilization across three links, (b) an example

in successive clock cycles. Thus, for message sizes that are large relative to the distance between sender and destination, the no-load latency is approximately a function of the message size rather than distance.

Several optimizations have been proposed to further improve the performance of wormhole switching. One example is flit reservation flow control was introduced to improve buffer turn-around time in wormhole switching [20]. Deeply pipelined, high speed routers can lead to low buffer occupancy as a consequence of propagation delays of flits over the link and the latency in receiving and processing credits before the buffer can be reused. Flit reservation is a technique that combines some elements of circuit switching (a priori reservations) to improve performance. A control flit advances ahead of the data flits of a message to reserve buffer and channel resources. Note that router pipeline for data flits is much shorter (they do not experience routing and switch allocation delays). As a result, reservations and transfers can be overlapped and buffer occupancy is significantly increased.

Another example that combines the advantages of wormhole switching and packet switching is *buffered wormhole switching (BWS)*. This was proposed and used in IBM's Power Parallel SP systems [21, 22]. In the absence of blocking, messages are routed through the network using wormhole switching. When messages block, 8-flit *chunks* are constructed at the input port of a switch and buffered in a dynamically allocated centralized router memory freeing up the input port for use by other messages. Subsequently, buffered chunks are transferred to an output port where they are converted to a flit stream for transmission across the physical channel. BWS differs from wormhole switching in that flits are not buffered in place. Rather flits are aggregated and buffered in a local memory within the switch and in this respect BWS is similar to packet switching. The no-load latency of a message routed using BWS is identical to that of wormhole-switched messages.

## Virtual Channels

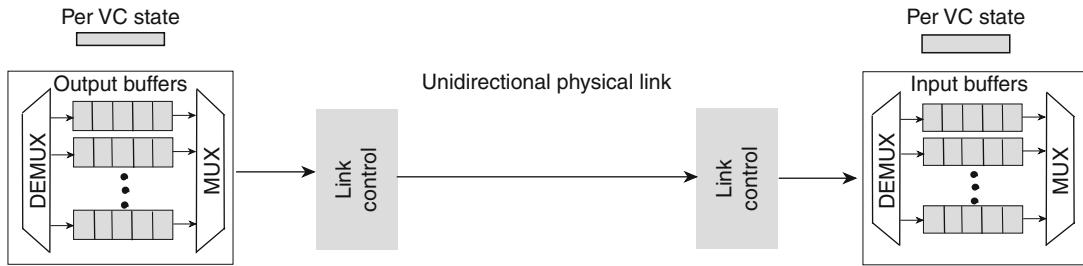
An important interconnection architecture function is the use of virtual channel flow control [1]. Each unidirectional virtual channel across a physical link is realized by an independently managed pair of message buffers. Multiple virtual channels are multiplexed across the physical link increasing link utilization

and network throughput. Importantly, routing constraints on the use of virtual channels are commonly used to ensure deadlock freedom. The use of virtual channels decouples message flows from the physical links and their use is orthogonal to the operation of switching techniques. Each switching technique is now employed to regulate the flow of packet data within a virtual channel while constraints on virtual channel usage may govern routing decisions at intermediate routers. The microarchitecture pipeline of the routers now includes an additional stage for virtual channel allocation. Virtual channels have been found to be particularly useful for optimizing the performance of wormhole-switched routers ameliorating the consequences of blocking and thus broadening the scope of application of wormhole switching. A simple example of the operation of virtual channel flow control is illustrated in Fig. 7.

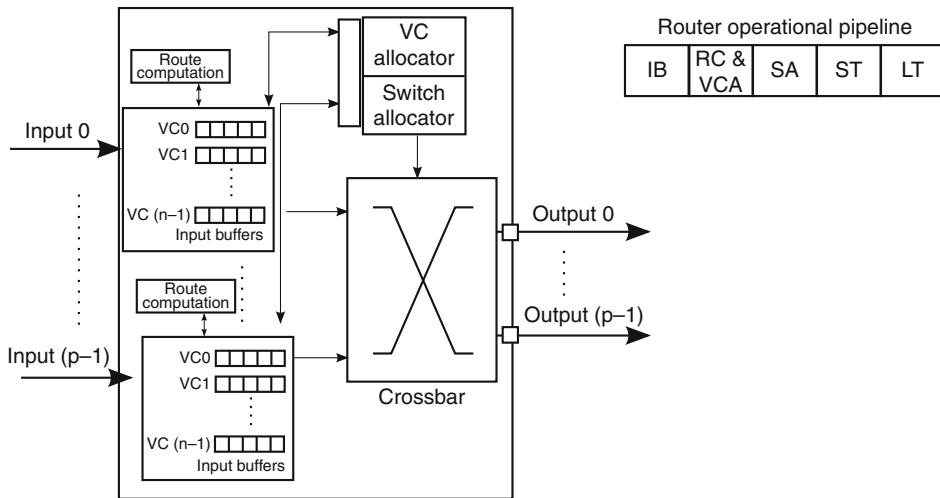
A modified router architecture reflecting the use of virtual channels is shown in Fig. 8. The route computation operation now returns a set of virtual channels that are candidates for forwarding the message (note that the router might be employing adaptive routing). A typical router pipeline is now extended by an extra stage as shown – *virtual channel allocation*. This is implemented prior to requesting an output port of the switch extending the router pipeline of Fig. 1 by one stage.

## A Comparison of Switching Techniques

Switching techniques have fundamental differences in their ability to utilize network bandwidth. In packet switching and VCT switching, messages are partitioned into fixed length packets each with its own header. Consequently, the overhead per transmitted data byte of a message is a fixed function of the message length. Wormhole switching supports variable sized messages and consequently overhead per data byte decreases with message size. However, as the network load increases when wormhole-switched messages block, they hold links and resources across multiple routers, wasting physical bandwidth, propagating congestion, and saturating the network at a fraction of the peak. The use of virtual channels in wormhole switching decouples the physical channel from blocked messages improving network utilization but increases the flow control latency across the physical link as well as the complexity of the channel controllers and intra-router switching.



**Switching Techniques. Fig. 7** Virtual channel flow control



**Switching Techniques. Fig. 8** The generic router with virtual channels

In VCT switching and packet switching, packets are fully buffered at each router and therefore traffic consumes network bandwidth in proportion to network load at the expense of increased amount of in-network buffering.

The latency behavior of packets using different switching techniques exhibits distinct behaviors. At low loads, the pipelined behavior of wormhole switching produces superior latency characteristics. However, saturation occurs at lower loads and the variance in packet latency is higher and accentuated with variance in packet length. The latency behavior of packets under packet switching tends to be more predictable since messages are fully buffered at each load. VCT packets will operate like wormhole switching at low loads and approximate packet switching at high loads where blocking will force packets to be buffered at an intermediate node. Consequently, approaches to provide Quality of Service guarantees (QoS) typically utilize VCT or packet switching. Attempting to control QoS when

blocked messages are spread across multiple nodes is by comparison much more difficult.

Reliability schemes are shaped by the switching techniques. Alternative topologies and routing algorithms affect the probability of encountering a failed component. The switching technique affects feasible detection and recovery algorithms. For example, packet switching is naturally suited to link level error detection and retransmission since each packet is an independently routable unit. For the same reason, packets may be adaptively routed around faulty regions of the network. However, when messages are pipelined over several links, error recovery and control becomes complicated. Recall that data flits have no routing information. Thus, errors that occur within a message that is spread across multiple nodes can lead to buffers and channel resources that are indefinitely occupied (e.g., link transceiver failures) and can lead to deadlocked message configurations. Thus, link level recovery must be accompanied by some higher

level layer recovery protocols that typically operate end-to-end.

Finally, it can be observed that the switching techniques exert a considerable influence on the architecture of the router, and as a result, the network performance. For example, flit level flow control enabled pipelined message transfers as well as the use of small buffers. The combination resulted in higher flow control signaling speeds and small compact router pipeline stages that could be clocked at higher speeds. The use of wormhole switching precluded the need for larger (slower) buffers or costly use of local storage at a node – the message could remain in the network. This is a critical design point if one considers that the link bandwidth can often exceed the memory bandwidth. Such architectural advances have amplified the performance gained via clock speed advances over several technology generations. For example, consider the difference in performance between the Cosmic cube network [23] that operated at 5 MHz and produced message latencies approaching hundreds of microseconds to milliseconds while the most recent TeraFlops chip from Intel operating at 5 GHz produces latencies on the order of nanoseconds. While performance has increased almost 5 orders of magnitude, the clock speeds have only increased by about 3 orders of magnitude. Much of this performance differential can be attributed to switching techniques and associated microarchitecture innovations that accompany their implementation.

## Related Entries

- [Collective Communication, Network Support for](#)
- [Congestion Management](#)
- [Flow Control](#)
- [Interconnection Networks](#)
- [Networks, Fault-Tolerant](#)
- [Routing \(Including Deadlock Avoidance\)](#)

## Bibliographic Notes and Further Reading

Related topics such as flow control and deadlock freedom are intimately related to switching techniques. A combined coverage of fundamental architectural, theoretical, and system concepts and a distillation of key concepts can also be found in two texts [4, 8]. More advanced treatments of these and related topics can

be found in papers in most major systems and computer architecture conferences with the preceding texts contributing references to many seminal papers in the field.

## Acknowledgments

Assistance and feedback from Mitchelle Rasquinha, Dhruv Choudhary, and Jeffrey Young are gratefully acknowledged.

## Bibliography

1. Dally WJ (1992) Virtual-channel flow control. *IEEE Trans Parallel Distrib Syst* 3(2):194–205
2. Peh L-S, Dally WJ (2001) A delay model for router microarchitectures. *IEEE Micro* 21:26–34
3. Peh L-S, Dally WJ (2001) A delay model and speculative architecture for pipelined routers. In: Proceedings of the 7th international symposium on high-performance computer architecture, Nuevo Leone
4. Dally WJ, Towles B (2004) Principles and practices of interconnection networks. Morgan Kaufman, San Francisco
5. Choi Y, Pinkston TM (2004) Evaluation of queue designs for true fully adaptive routers. *J Parallel Distrib Comput* 64(5):606–616
6. Mullins R, West A, Moore S (2004) Low-latency virtual-channel routers for on-chip networks. In: Proceedings of the 31st annual international symposium on computer architecture, Munchen
7. Dally WJ, Seitz CL (1987) Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans Comput C-36(5):547–553*
8. Duato J, Yalamanchili S, Ni L (2003) Interconnection networks: an engineering Approach. Morgan Kaufmann, San Francisco
9. Flynn M (1995) Computer architecture: pipelined and parallel processor design. Jones & Bartlett, Boston, pp 63–140
10. Duato J et al (1996) A high performance router architecture for interconnection networks. In: Proceedings of the 1996 international conference on parallel processing, Bloomington, vol I, August 1996, pp 61–68
11. Scott SL, Goodman JR (1994) The impact of pipelined channels on k-ary n-cube networks. *IEEE Trans Parallel Distrib Syst* 5(1):2–16
12. Gaughan PT et al (1996) Distributed, deadlock-free routing in faulty, pipelined, direct interconnection networks. *IEEE Trans Comput* 45(6):651–665
13. Borkar S et al (1988) iWarp: an integrated solution to high-speed parallel computing. In: Proceedings of supercomputing '88, Orlando, November 1988, pp 330–339
14. Kermani P, Kleinrock L (1979) Virtual cut-through: a new computer communication switching technique. *Comp Networks* 3(4):267–286
15. Dally WJ, Seitz CL (1986) The torus routing chip. *J Distrib Comput* 1(3):187–196
16. Hoskote Y, Vangal S, Singh A, Borkar N, Borkar S (2007) A 5-GHz mesh interconnect for a teraflops processor. *IEEE Micro* 27(5): 51–61

17. Duato J (1993) A new theory of deadlock-free adaptive routing in wormhole networks. *IEEE Trans Parallel Distrib Syst* 4(12): 1320–1331
18. Duato J (1995) A necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks. *IEEE Trans Parallel Distrib Syst* 6(10):1055–1067
19. Duato J (1996) A necessary and sufficient condition for deadlock-free routing in cut-through and store-and-forward networks. *IEEE Trans Parallel Distrib Syst* 7(8):841–854
20. Peh L-S, Dally WJ (2000) Flit reservation flow control. In: *Proceedings of the 6th international symposium on high-performance computer architecture*, Toulouse, France, January 2000, pp 73–84
21. Stunkel CB et al (1994) Architecture and implementation of vulcan. In: *Proceedings of the 8th international parallel processing symposium*, Cancun, Mexico, pp 266–274
22. Stunkel CB et al (1994) The SPI high-performance switch. In: *Proceedings of the scalable high performance computing conference*, Knoxville, pp 150–157
23. Seitz C (1985) The cosmic cube. *Commun ACM* 28(1):22–23

may produce incorrect results. As a trivial example, consider a global counter incremented by multiple threads. Each thread loads the counter into a register, increments the register, and writes the updated value back to memory. If two threads load the same value before either stores it back, updates may be lost:

```
c == 0
Thread 1: Thread 2:
r1 := c r1 := c
++r1 ++r1
c := r1 c := r1
c == 1
```

Synchronization serves to preclude invalid thread interleavings. It is commonly divided into the subtasks of *atomicity* and *condition synchronization*. Atomicity ensures that a given sequence of instructions, typically performed by a single thread, appears to all other threads as if it had executed indivisibly – not interleaved with anything else. In the example above, one would typically specify that the load-increment-store instruction sequence should execute atomically.

Condition synchronization forces a thread to wait, before performing an operation on shared data, until some desired precondition is true. In the example above, one might want to wait until all threads had performed their increments before reading the final count.

While it is tempting to suspect that condition synchronization subsumes atomicity (make the precondition be that no other thread is currently executing a conflicting operation), atomicity is in fact considerably harder, because it requires *consensus* among all competing threads: they must all agree as to which will proceed and which will wait. Put another way, condition synchronization delays a thread until some locally observable condition is seen to be true; atomicity is a property of the system as a whole.

Like many aspects of parallel computing, synchronization looks different in shared-memory and message-passing systems. In the latter, synchronization is generally subsumed in the message-passing methods; in a shared-memory system, it typically employs a separate set of methods.

## Symmetric Multiprocessors

► [Shared-Memory Multiprocessors](#)

## Synchronization

MICHAEL L. SCOTT

University of Rochester, Rochester, NY, USA

### Synonyms

[Fences](#); [Multiprocessor synchronization](#); [Mutual exclusion](#); [Process synchronization](#)

### Definition

Synchronization is the use of language or library mechanisms to constrain the ordering (interleaving) of instructions performed by separate threads, to preclude orderings that lead to incorrect or undesired results.

### Discussion

In a parallel program, the instructions of any given thread appear to occur in sequential order (at least from that thread's point of view), but if the threads run independently, their sequences of instructions may interleave arbitrarily, and many of the possible interleavings

Shared-memory implementations of synchronization can be categorized as *busy-wait (spinning)*, or *scheduler-based*. The former actively consume processor cycles until the running thread is able to proceed. The latter deschedule the current thread, allowing the processor to be used by other threads, with the expectation that future activity by one of those threads will make the original thread runnable again. Because it avoids the cost of two context switches, busy-wait synchronization is typically faster than scheduler-based synchronization when the expected wait time is short and when the processor is not needed for other purposes. Scheduler-based synchronization is typically faster when expected wait times are long; it is *necessary* when the number of threads exceeds the number of processors (else quantum-long delays or even deadlock can occur). In the typical implementation, busy-wait synchronization is built on top of whatever hardware instructions execute atomically. Scheduler-based synchronization, in turn, is built on top of busy-wait synchronization, which is used to protect the scheduler's own data structures (see entries on *Scheduling Algorithms* and on *Processes, Tasks, and Threads*).

## Hardware Primitives

In the earliest multiprocessors, *load* and *store* were the only memory-access instructions guaranteed to be atomic, and busy-wait synchronization was implemented using these. Modern machines provide a variety of atomic *read-modify-write* (RMW) instructions, which serve to *update* a memory location atomically. These significantly simplify the implementation of synchronization. Common RMW instructions include:

**Test-and-set ( $l$ )** sets the Boolean variable at location  $l$  to *true*, and returns the previous value.

**Swap ( $l, v$ )** stores the value  $v$  to location  $l$  and returns the previous value.

**Atomic- $\phi$  ( $l, v$ )** replaces the value  $o$  at location  $l$  with  $\phi(o, v)$  for some simple arithmetic function  $\phi$  (add, sub, and, etc.).

**Fetch-and- $\phi$  ( $l, v$ )** is like atomic- $\phi$ , but also returns the previous value.

**Compare-and-swap ( $l, o, n$ )** inspects the value  $v$  at location  $l$ , and if it is equal to  $o$ , replaces it with  $n$ .

In either case, it returns the previous value, from which one can deduce whether the replacement occurred.

**Load-linked ( $l$ ) and store-conditional ( $l, v$ )**. The first of these returns the value at location  $l$  and “remembers”  $l$ . The second stores  $v$  to  $l$  if  $l$  has not been modified by any other processor since a previous load-linked by the current processor.

These instructions differ in their expressive power. Herlihy has shown [9] that compare-and-swap (CAS) and load-linked / store-conditional (LL/SC) are *universal* primitives, meaning, informally, that they can be used to construct a *non-blocking* implementation of any other RMW operation. The following code provides a simple implementation of fetch-and- $\phi$  using CAS.

```
val old := *l;
loop
 val new := phi(old);
 val found := CAS(l, old, new);
 if (old == found) break;
 old := found;
```

If the test on line 5 of this code fails, it must be because some other thread successfully modified  $*l$ . The system as a whole has made forward progress, but the current thread must try again.

As discussed in the entry on Non-blocking Algorithms, this simple implementation is *lock-free* but not *wait-free*. There are stronger (but slower and more complex) non-blocking implementations in which each thread is guaranteed to make forward progress in a bounded number of its own instructions.

**NB:** In any distributed system, and in most modern shared memory systems, instructions executed by a given thread are not, in general, guaranteed to be seen in sequential order by other threads, and instructions of any two threads are not, in general, guaranteed to be seen in the same order by all of their peers. Modern processors typically provide so-called *fence* or *barrier* instructions (not to be confused with the barriers discussed under Condition Synchronization below) that force previous instructions of the current thread to be seen by other threads before subsequent instructions of the current thread. Implementations of synchronization

methods typically include sufficient fences that if synchronization method  $s_1$  in thread  $t_1$  occurs before synchronization method  $s_2$  in thread  $t_2$ , then all instructions that precede  $s_1$  in  $t_1$  will appear in  $t_2$  to have occurred before any of its own instructions that follow  $s_2$ . For more information, see the entry on Memory Models. The remainder of the discussion here assumes that memory is *sequentially consistent*, that is, that instructions appear to interleave in some global total order that is consistent with program order in every thread.

## Atomicity

A multi-instruction operation is said to be *atomic* if it appears to occur “all at once” from every other thread’s point of view. In a sequentially consistent system, this means that the program behaves as if the instructions of the atomic operation were contiguous in the global instruction interleaving. More specifically, in any system, intermediate states of the atomic operation should never be visible to other threads, and actions of other threads should never become visible to a given thread in the middle of one of its own atomic operations.

The most straightforward way to implement atomicity is with a *mutual-exclusion (mutex) lock* – an abstract object that can be *held* by at most one thread at a time. In standard usage, a thread invokes the *acquire* method of the lock when it wishes to begin an atomic operation and the *release* method when it is done. *Acquire* waits (by spinning or rescheduling) until it is safe for the operation to proceed. The code between the acquire and release (the body of the atomic operation) is known as a *critical section*.

Critical sections that conflict with one another (typically, that access some common location, with at least one section writing that location) must be protected by the same lock. Programming discipline commonly ensures this property by associating data with locks. A thread must then acquire locks for all the data accessed in a critical section. It may do so all at once, at the beginning of the critical section, or it may do so incrementally, as the need for data is encountered. Considerable care may be required to ensure that locks are acquired in the same order by all critical sections, to avoid deadlock. All locks are typically held until the end of the critical section. This *two-phase locking* (all acquires occur

before any releases) ensures that the global set of critical section executions remains *serializable*.

## Relaxations of Mutual Exclusion

So-called *reader-writer locks* increase concurrency by observing that it is safe for more than one thread to read a location concurrently, so long as no thread is modifying that location. Each critical section is classified as either a reader or a writer of the data associated with a given lock. The *reader\_acquire* method waits until there is no concurrent writer of the lock; the *writer\_acquire* method waits until there is no concurrent reader or writer.

In a standard reader-writer lock, a thread must know, when it first reads a location, whether it will ever need to write that location in the current critical section. In some contexts it may be possible to relax this restriction. The Linux kernel, for example, provides a *sequence lock* mechanism that allows a reader to *abort* its peers and upgrade to writer status. Programmers are required to follow a restrictive programming discipline that makes critical sections “restartable,” and checks, before any write or “dangerous” read, to see whether a peer’s upgrade has necessitated a restart.

For data structures that are almost always read, and very occasionally written, several operating system kernels provide some variant of a mechanism known as RCU (originally an abbreviation for read-copy update). RCU divides execution into so-called *epochs*. A writer creates a new copy of any data structure it needs to update. It replaces the old copy with the new, typically using a single CAS instruction. It then waits until the end of the current epoch to be sure that all readers that might have been using the old copy have completed their critical sections (at which point it can reclaim the old copy, or perform other actions that depend on the visibility of the update). The advantage of RCU, in comparison to locks, is that it imposes *zero overhead* in the read-only case.

For more general-purpose use, *transactional memory* (TM) allows arbitrary operations to be executed atomically, with an underlying implementation based on *speculation* and *rollback*. Originally proposed [10] as a hardware assist for lock-free data structures – sort of a multi-word generalization of LL/SC – TM has seen a flurry of activity in recent years, and several hardware

and software implementations are now widely available. Each keeps track of the memory locations accessed by transactions (would-be atomic operations). When two concurrent transactions are seen to conflict, at most one is allowed to *commit*; the others *abort*, “roll back,” and try again, using a fully automated, transparent analogue of the programming discipline required by sequence locks. For further details, see the separate entry on TM.

## Fairness

Because they sometimes force multiple threads to wait, synchronization mechanisms inevitably raise issues of *fairness*. When a lock is released by the current holder, which waiting thread should be allowed to acquire it? In a system with reader-writer locks, should a thread be allowed to join a group of already-active readers when writers are already waiting? When transactions conflict in a TM system, which should be permitted to proceed, and which should wait or abort?

Many answers are possible. The choice among conflicting threads may be arbitrary, random, first-come-first-served (FIFO), or based on some other notion of priority. From the point of view of an individual thread, the resulting behavior may range from potential *starvation* (no progress guarantees) to some sort of proportional share of system run time. Between these extremes, a thread may be guaranteed to run eventually if it is continuously ready, or if it is ready infinitely often. Even given the possibility of starvation, the system as a whole may be *livelock-free* (guaranteed to make forward progress) as a result of algorithmic guarantees or pseudo-random heuristics. (Actual livelock is generally considered unacceptable.) Any starvation-free system is clearly livelock free.

## Simple Busy-Wait Locks

Several early locking algorithms were based on only loads and stores, but these are mainly of historical interest today. All required  $\Omega(tn)$  space for  $t$  threads and  $n$  locks, and  $\omega(1)$  (more-than-constant) time to arbitrate among threads competing for a given lock.

In modern usage, the simplest constant-space, busy-wait mutual exclusion lock is the *test-and-set* (TAS)

*lock*, in which a thread acquires the lock by using a test-and-set instruction to change a Boolean flag from false to true. Unfortunately, spinning by waiting threads tends to induce extreme contention for the lock location, tying up bus and memory resources needed for productive work. On a cache-coherent machine, better performance can be achieved with a “test-and-test-and-set” (TATAS) lock, which reduces contention by using ordinary load instructions to spin on a value in the local cache so long as the lock remains held:

```
type lock = Boolean;
proc acquire(lock *l):
 while (test-and-set(l))
 while (*l) /* spin */ ;
proc release(lock *l):
 *l := false;
```

This lock works well on small machines (up to, say, four processors).

Which waiting thread acquires a TATAS lock at release time depends on vagaries of the hardware, and is essentially arbitrary. Strict FIFO ordering can be achieved with a *ticket lock*, which uses fetch-and-increment (FAI) and a pair of counters for constant space and (per-thread) time. To acquire the lock, a thread atomically performs an FAI on the “next available” counter and waits for the “now serving” counter to equal the value returned. To release the lock, a thread increments its own ticket, and stores the result to the “now serving” counter. While arguably fairer than a TATAS lock, the ticket lock is more prone to performance anomalies on a multiprogrammed system: if any waiting thread is preempted, all threads behind it in line will be delayed until it is scheduled back in.

## Scalable Busy-Wait Locks

On a machine with more than a handful of processors, TATAS and ticket locks scale poorly, with time per critical section growing linearly with the number of waiting threads. Anderson [1] showed that exponential backoff (reminiscent of the Ethernet contention-control algorithm) could substantially improve the performance of TATAS locks. Mellor-Crummey and Scott [17] showed similar results for linear backoff in ticket locks (where

a thread can easily deduce its distance from the head of the line).

To eliminate contention entirely, waiting threads can be linked into an explicit queue, with each thread spinning on a separate location that will be modified when the thread ahead of it in line completes its critical section. Mellor-Crummey and Scott showed how to implement such queues in total space  $O(t + n)$  for  $t$  threads and  $n$  locks; their *MCS lock* is widely used in large-scale systems. Craig [4] and, independently, Landin and Hagersten [16] developed an alternative *CLH lock* that links the queue in the opposite direction and performs slightly faster on some cache-coherent machines. Auslander et al. developed a variant of the MCS lock that is API-compatible with traditional TATAS locks [3]. Kontothanassis et al. [14] and He et al. [8] developed variants of the MCS and CLH locks that avoid performance anomalies due to preemption of threads waiting in line.

### Scheduler-Based Locks

A busy-wait lock wastes processor resources when expected wait times are long. It may also cause performance anomalies or deadlock in a multiprogrammed system. The simplest solution is to *yield* the processor in the body of the spin loop, effectively moving the current thread to the end of the scheduler's ready list and allowing other threads to run. More commonly, *scheduler-based locks* are designed to *deschedule* the waiting thread, moving it (atomically) from the ready list to a separate queue associated with the lock. The release method then moves one waiting thread from the lock queue to the ready list. To minimize overhead when waiting times are short, implementations of scheduler-based synchronization commonly spin for a small, bounded amount of time before invoking the scheduler and yielding the processor. This strategy is often known as *spin-then-wait*.

### Condition Synchronization

It is tempting to assume that busy-wait condition synchronization can be implemented trivially with a Boolean flag: a waiting thread spins until the flag is true; a thread that satisfies the condition sets the flag to true. On most modern machines, however, additional fence

instructions are required both in the satisfying thread, to ensure that its prior writes are visible to other threads, and in the waiting thread, to ensure that its subsequent reads do not occur until after the spin completes. And even on a sequentially consistent machine, special steps are required to ensure that the compiler does not violate the programmer's expectations by reordering instructions within threads.

In some programming languages and systems, a variable may be made suitable for condition synchronization by labeling it *volatile* (or, in C++'0X, *atomic<>*). The compiler will insert appropriate fences at reads and writes of *volatile* variables, and will refrain from reordering them with respect to other instructions.

Some other systems provide special *event* objects, with methods to set and await them. Semaphores and monitors, described in the following two subsections, can be used for both mutual exclusion and condition synchronization.

In systems with dynamically varying concurrency, the *fork* and *join* methods used to create threads and to verify their completion can be considered a form of condition synchronization. (These are, in fact, the principal form of synchronization in systems like Cilk and OpenMP.)

### Barriers

One form of condition synchronization is particularly common in data-parallel applications, where threads iterate together through a potentially large number of algorithmic phases. A *synchronization barrier*, used to separate phases, guarantees that no thread continues to phase  $n + 1$  until all threads have finished phase  $n$ .

In most (though not all) implementations, the barrier provides a single method, composed internally of an *arrival* phase that counts the number of threads that have reached the barrier (typically via a log-depth tree) and a *departure* phase in which permission to continue is broadcast back to all threads. In a so-called *fuzzy barrier* [6], these arrival and departure phases may be separate methods. In between, a thread may perform any instructions that neither depend on the arrival of other threads nor are required by other threads prior to their departure. Such instructions can serve to "smooth out" phase-by-phase imbalances in the work assigned

to different threads, thereby reducing overall wait time. Wait time may also be reduced by an *adaptive barrier* [7, 19], which completes the arrival phase in constant time after the arrival of the final thread.

Unfortunately, when  $t$  threads arrive more or less simultaneously, no barrier implementation using ordinary loads, stores, and RMW instructions can complete the arrival phase in less than  $\Omega(\log t)$  time. Given the importance of barriers in scientific applications, some supercomputers have provided special near-constant-time hardware barriers. In some cases the same hardware has supported a fast *eureka* method, in which one thread can announce an event to all others in constant time.

## Semaphores

First proposed by Dijkstra in 1965 [5] and still widely used today, *semaphores* support both mutual exclusion and condition synchronization. A *general semaphore* is a nonnegative counter with an initial value and two methods, known as **V** and **P**. The **V** method increases the value of the semaphore by one. The **P** method waits for the value to be positive and then decreases it by one. A *binary semaphore* has values restricted to zero and one (it is customarily initialized to one), and serves as a mutual exclusion lock. The **P** method acquires the lock; the **V** method releases the lock. Programming discipline is required to ensure that **P** and **V** methods occur in matching pairs.

The typical implementation of semaphores pairs the counter with a queue of waiting threads. The **V** method checks to see whether the counter is currently zero. If so, it checks to see whether any threads are waiting in the queue and, if there are, moves one of them to the ready list. If the counter is already positive (in which case the queue is guaranteed to be empty) or if the counter is zero but the queue is empty, **V** simply increments the counter. The **P** method also checks to see whether the counter is zero. If so, it places the current thread on the queue and calls the scheduler to yield the processor. Otherwise it decrements the counter.

General semaphores can be used to represent resources of which there is a limited number, but more than one. Examples include I/O devices, communication channels, or free or full slots in a fixed-length buffer. Most operating systems provide semaphores as part of the kernel API.

## Monitors

While semaphores remain the most widely used scheduler-based shared-memory synchronization mechanism, they suffer from several limitations. In particular, the association between a binary semaphore (mutex lock) and the data it protects is solely a matter of convention, as is the paired usage of **P** and **V** methods. Early experience with semaphores, combined with the development of language-level abstraction mechanisms in the 1970s, led several developers to suggest building higher-level synchronization abstractions into programming languages. These efforts culminated in the definition of *monitors* [12], variants of which appear in many languages and systems.

A monitor is a data abstraction (a module or class) with an implicit mutex lock and an optional set of *condition variables*. Each *entry* (method) of the monitor automatically acquires and releases the mutex lock; entry invocations thus exclude one another in time. Programmers typically devise, for each monitor, a program-specific *invariant* that captures the mutual consistency of the monitor's state (data members – fields). The invariant is assumed to be true at the beginning of each entry invocation, and must be true again at the end.

Condition variables support a pair of methods superficially analogous to **P** and **V**; in Hoare's original formulation, these were known as *wait* and *signal*. Unlike **P** and **V**, these methods are *memory-less*: a signal invocation is a no-op if no thread is currently waiting.

For each reason that a thread might need to wait within a monitor, the programmer declares a separate condition variable. When it waits on a condition, the thread releases exclusion on the monitor. The programmer must thus ensure that the invariant is true immediately prior to every wait invocation.

## Semantic Details

The details of monitors vary significantly from one language to another. The most significant issues, discussed in the paragraphs below, are commonly known as the *nested monitor problem* and the modeling of signals as *hints vs. absolutes*. More minor issues include language syntax, alternative names for signal and wait, the modeling of condition variables in the type system, and the prioritization of threads waiting for conditions or for access to the mutex lock.

The nested monitor problem arises when an entry of one monitor invokes an entry of another monitor, and the second entry waits on a condition variable. Should the wait method release exclusion on the outer monitor? If it does, there is no guarantee that the outer monitor will be available again when execution is ready to resume in the inner call. If it does not, the programmer must take care to ensure that the thread that will perform the matching signal invocation does not need to go through the outer monitor in order to reach the inner one. A variety of solutions to this problem have been proposed; the most common is to leave the outer monitor locked.

Signal methods in Hoare's original formulation were defined to transfer monitor exclusion directly from the signaler to the waiter, with no intervening execution. The purpose of this convention was to guarantee that the condition represented by the signal was still true when the waiter resumed. Unfortunately, the convention often has the side effect of inducing extra context switches, and requires that the monitor invariant be true immediately prior to every signal invocation. Most modern monitor variants follow the lead of Mesa [15] in declaring that a signal is merely a hint, and that a waiting process must double-check the condition before continuing execution. In effect, code that would be written

```
if (!condition)
 cond_var.wait();
```

in a Hoare monitor is written

```
while (!condition)
 cond_var.wait();
```

in a Mesa monitor. To make it easier to write programs in which a condition variable "covers" a set of possible conditions (particularly when signals are hints), many monitor variants provide a *signal-all* or *broadcast* method that awakens all threads waiting on a condition, rather than only one.

## Message Passing

In a system in which threads interact by exchanging messages, rather than by sharing variables, synchronization is generally implicit in the *send* and *receive* methods. A receive method typically blocks until an appropriate message is available (a matching send has

been performed). Blocking semantics for send methods vary from one system to another:

**Asynchronous send** – In some systems, a sender continues execution immediately after invoking a send method, and the underlying system takes responsibility for delivering the message. While often desirable, this behavior complicates the delivery of failure notifications, and may be limited by finite buffering capacity.

**Synchronous send** – In other systems – notably those based on Hoare's Communicating Sequential Processes (CSP) [13] – a sender waits until its message has been received.

**Remote-invocation send** – In yet other systems, a send method has both ingoing and outgoing parameters; the sender waits until a reply is received from its peer.

## Distributed Locking

Libraries, languages, and applications commonly implement higher-level distributed locks or transactions on top of message passing. The most common lock implementation is analogous to the MCS lock: acquired requests are sent to a *lock manager* thread. If the lock is available, the manager responds directly; otherwise it forwards the request to the last thread currently waiting in line. The release method sends a message to the manager or, if a forwarding request has already been received, to the next thread in line for the lock. Races in which the manager forwards a request at the same time the last lock holder sends it a release are trivially resolved by statically choosing one of the two (perhaps the lock holder) to inform the next thread in line. Distributed transaction systems are substantially more complex.

S

## Rendezvous and Remote Procedure Call

In some systems, a message must be received explicitly by an already existing thread. In other systems, a thread is created by the underlying system to handle each arriving message. Either of these options – *explicit* or *implicit receipt* – can be paired with any of the three send options described above. The combination of remote-invocation send with implicit receipt is often called *remote procedure call* (RPC). The combination of remote-invocation send with explicit receipt is known

as *rendezvous*. Interestingly, if all shared data is encapsulated in monitors, one can model – or implement – each monitor with a *manager* thread that executes entry calls one at a time. Each such call then constitutes a rendezvous between the sender and the monitor.

## Related Entries

- ▶ [Actors](#)
- ▶ [Cache Coherence](#)
- ▶ [Concurrent Collections Programming Model](#)
- ▶ [Deadlocks](#)
- ▶ [Memory Models](#)
- ▶ [Monitors, Axiomatic Verification of](#)
- ▶ [Non-Blocking Algorithms](#)
- ▶ [Path Expressions](#)
- ▶ [Processes, Tasks, and Threads](#)
- ▶ [Race Conditions](#)
- ▶ [Scheduling Algorithms](#)
- ▶ [Shared-Memory Multiprocessors](#)
- ▶ [Transactions, Nested](#)

## Bibliographic Notes

The study of synchronization began in earnest with Dijkstra's "Cooperating Sequential Processes" monograph of 1965 [5]. Andrews and Schneider provide an excellent survey of synchronization mechanisms circa 1983 [2]. Mellor-Crummey and Scott describe and compare a variety of busy-wait spin locks and barriers, and introduce the MCS lock [17]. More extensive coverage of synchronization can be found in Chapter 12 of Scott's programming languages text [18], or in the recent texts of Herlihy and Shavit [11] and Taubenfeld [20].

## Bibliography

1. Anderson TE (Jan 1990) The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans Parallel Distr Sys* 1(1):6–16
2. Andrews GR, Schneider FB (Mar 1983) Concepts and notations for concurrent programming. *ACM Comput Surv* 15(1):3–43
3. Auslander MA, Edelsohn DJ, Krieger OY, Rosenburg BS, Wisniewski RW (2003) Enhancement to the MCS lock for increased functionality and improved programmability. U.S. patent application 20030200457, submitted 23 Oct 2003
4. Craig TS (Feb 1993) Building FIFO and priority-queueing spin locks from atomic swap. Technical Report 93-02-02, University of Washington Computer Science Department

5. Dijkstra EW (Sept 1965) Cooperating sequential processes. Technical report, Technological University, Eindhoven, The Netherlands. Reprinted in Genyus F (ed) *Programming Languages*, Academic Press, New York, 1968, pp 43–112. Also available at [www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html](http://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html).
6. Gupta R (Apr 1989) The fuzzy barrier: a mechanism for high speed synchronization of processors. Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, MA, pp 54–63
7. Gupta R, Hill CR (June 1989) A scalable implementation of barrier synchronization using an adaptive combining tree. *Int J Parallel Progr* 18(3):161–180
8. He B, Scherer III WN, Scott ML (Dec 2005) Preemption adaptivity in time-published queuebased spin locks. Proceeding of the 2005 International Conference on High Performance Computing, Goa, India
9. Herlihy MP (Jan 1991) Wait-free synchronization. *ACM Trans Progr Lang Syst* 13(1):124–149
10. Herlihy MP, Moss JEB (1993) Transactional memory: architectural support for lock-free data structures. Proceedings of the 20th International Symposium on Computer Architecture, San Diego, CA, May 1993 pp 289–300
11. Herlihy MP, Shavit N (2008) *The Art of Multiprocessor Programming*. Morgan Kaufmann, Burlington, MA
12. Hoare CAR (Oct 1974) Monitors: an operating system structuring concept. *Commun ACM* 17(10):549–557
13. Hoare CAR (Aug 1978) Communicating sequential processes. *Commun ACM* 21(8):666–677
14. Kontothanassis LI, Wisniewski R, Scott ML (Feb 1997) Scheduler-conscious synchronization. *ACM Trans Comput Sys* 15(1):3–40
15. Lampson BW, Redell DD (Feb 1980) Experience with processes and monitors in Mesa. *Commun ACM* 23(2):105–117
16. Magnussen P, Landin A, Hagersten E (Apr 1994) Queue locks on cache coherent multiprocessors. Proceedings of the 8th International Parallel Processing Symposium, Cancun, Mexico, pp 165–171
17. Mellor-Crummey JM, Scott ML (Feb 1991) Algorithms for scalable synchronization on sharedmemory multiprocessors. *ACM Trans Comput Syst* 9(1):21–65
18. Scott ML (2009) *Programming Language Pragmatics*, 3rd edn. Morgan Kaufmann, Burlington, MA
19. Scott ML, Mellor-Crummey JM (Aug 1994) Fast, contention-free combining tree barriers. *Int J Parallel Progr* 22(4):449–481
20. Taubenfeld G (2006) *Synchronization Algorithms and Concurrent Programming*. Prentice Hall, Upper Saddle River

## System Integration

- ▶ [Terrestrial Ecosystem Carbon Modeling](#)

## System on Chip (SoC)

- [SoC \(System on Chip\)](#)
- [VLSI Computation](#)

## Systems Biology, Network Inference in

JAROSLAW ZOLA<sup>1</sup>, SRINIVAS ALURU<sup>1,2</sup>

<sup>1</sup>Iowa State University, Ames, IA, USA

<sup>2</sup>Indian Institute of Technology Bombay,  
Mumbai, India

### Synonyms

[Gene networks reconstruction](#); [Gene networks reverse-engineering](#)

### Definition

Inference of gene regulatory networks, also called reverse-engineering of gene regulatory networks, is a process of characterizing, either qualitatively or quantitatively, regulatory mechanisms in a cell or an organism from observed expression data.

### Discussion

#### Introduction

Biological processes in every living organism are governed by complex interactions between thousands of genes, gene products, and other molecules. Genes that are encoded in the DNA are transcribed and translated to form multiple copies of gene products including proteins and various types of RNAs. These gene products coordinate to execute cellular processes – sometimes by forming supramolecular complexes (e.g., ribosome), or by acting in a concerted fashion, e.g., in biochemical or metabolic pathways. They also regulate the expression of genes, often through binding to cis-regulatory sequences upstream of the coding region of the genes, to calibrate gene expression depending on the endogenous and exogenous stimuli carried by, e.g., small molecules.

Gene regulatory networks are conceptual representations of interactions between genes in a cell or an organism. They are depicted as graphs with vertices corresponding to genes and edges representing regulatory interactions between genes (see [Fig. 1](#)). Overall,

gene regulatory networks are mathematical models to explain the observed gene expression levels. Network inference, or reconstructing, is the process of identifying the underlying network from multiple observations of gene expressions (outputs of the network). To infer a gene network, one relies on experimental data from high-throughput technologies such as microarrays, quantitative polymerase chain reaction, or short-read sequencing, which measure a snapshot of all gene expression levels under a particular condition or in a time series.

#### Information Theoretic Approaches

Consider a set of  $n$  genes  $\{g_1, g_2, \dots, g_n\}$ , where for each gene a set of  $m$  expression measurements is given. One can represent expression of gene  $i$  ( $g_i$ ) as a random variable  $X_i \in \mathcal{X}$ ,  $\mathcal{X} = \{X_1, \dots, X_n\}$ , with marginal probability  $p_{X_i}$  derived from some unknown joint probability characterizing the entire system. This random variable is described by observations  $\{x_{i,1}, \dots, x_{i,m}\}$ , where  $x_{i,j}$  corresponds to the expression level of  $g_i$  under condition  $j$ . The vector  $\langle x_{i,1}, x_{i,2}, \dots, x_{i,m} \rangle$  is called profile of  $g_i$ . Given a profile matrix  $Y_{n \times m}$ ,  $Y[i,j] = x_{i,j}$ , one can formulate network inference problem as that of finding a model that best explains the data in  $Y$ .

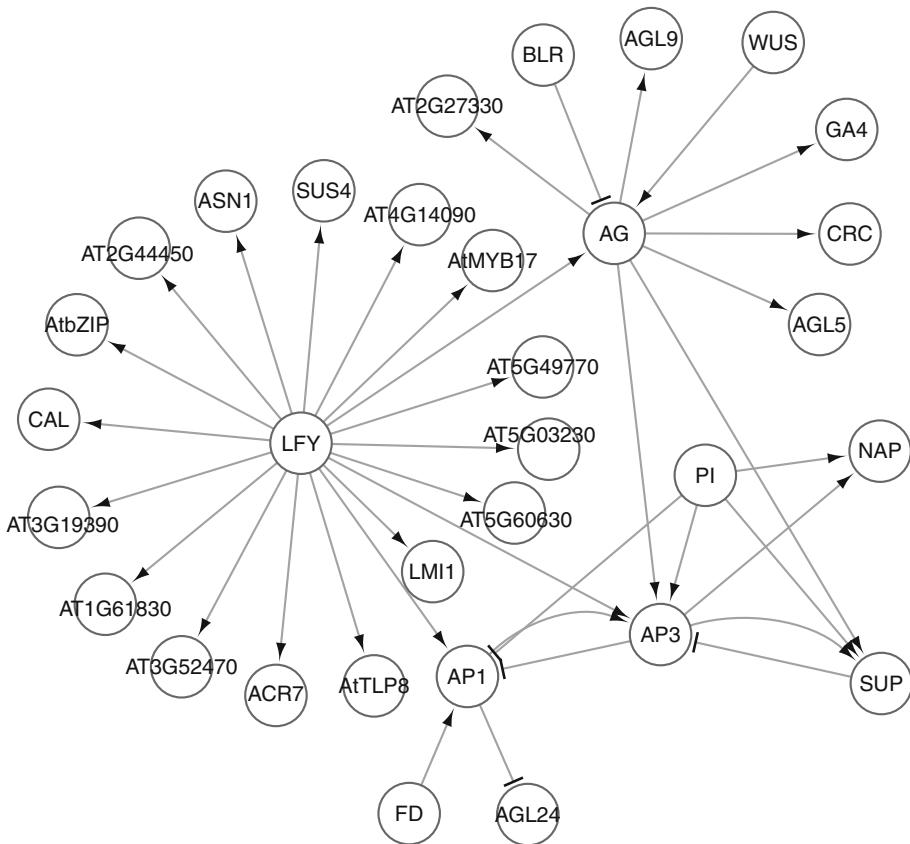
Such formulated problem can be approached using a variety of methods, including Bayesian networks [21] and Gaussian graphical models [20]; one class of methods that has been widely adopted uses the concept of mutual information. These methods [15] operate under the assumption that correlation of expression implies coregulation, and proceed in two main phases: First significant dependencies (connection between two genes) or independencies (lack of connections) are determined by means of computing mutual information for every pair of genes. Then, identification and removal of indirect interactions (e.g., when two genes are coregulated by a third) is performed.

Mutual information is arguably the best measure of correlation between two random variables, and is defined based on entropy  $\mathcal{H}$  in the following way:

$$\mathcal{I}(X_i; X_j) = \mathcal{H}(X_i) + \mathcal{H}(X_j) - \mathcal{H}(X_i, X_j),$$

where entropy  $\mathcal{H}$  is given by:

$$\mathcal{H}(X) = - \sum p_X(x) \log p_X(x),$$



**Systems Biology, Network Inference in.** Fig. 1 Example gene regulatory network. Nodes represent genes, “*T*-edges” denote regulation in which a source gene represses expression of the target gene. Arrow-edges denote regulation in which a source gene induces expression of the target gene

and  $p_X$  defines the probability distribution of  $X$ , and  $\sum$  is replaced by integral if  $X$  is continuous. Mutual information is a symmetric, nonnegative function, and is equal to zero if and only if two random variables are independent.

Application of mutual information for gene network inference poses two significant challenges. As the probability distribution of the random variable describing a gene is unknown, it has to be estimated from the expression profile. Consequently, gene comparison becomes more difficult because even independent expression profiles can result in mutual information greater than zero (owing to sampling and estimation errors). This in turn requires some mechanism to decide if the given mutual information estimate is statistically significant. The second challenge is due to the fact that a typical

genome-level network covers thousands of genes, and hence “all-pairs” comparison adds considerably to computational requirements. In practice, several mutual information estimators are available that offer different precision to complexity ratios (e.g., Gaussian kernel estimator, B-spline estimator), and complex statistical techniques are employed to decide if observed mutual information implies dependency.

Although all information theoretic approaches for reverse-engineering depend on “all-pairs” mutual information kernel executed in the first stage, they differ in how they identify indirect interactions in the second stage. For example, in relevance networks [4] the second stage is omitted, in ARACNe [3] and TINGe [22, 23] the Data Processing Inequality concept is used, while CLR algorithm [6] depends on estimates of a likelihood

of obtained mutual information values. Some other methods extend mutual information into conditional mutual information or augment it with feature selection techniques.

### Parallel Information Theoretic Approach

Reverse engineering of regulatory networks using mutual information is compute and memory intensive especially if whole-genome (i.e., covering all genes of an organism) networks are considered. Memory consumption arises from the  $\Theta(nm)$  size of input data, and from the  $\Theta(n^2)$  dense initial network generated in the first phase of the reconstruction algorithm. Adding to this is complexity of mutual information estimators, which for Gaussian kernel estimator for instance is  $O(m^2)$ . Taking into account that the number of genes typically considered is in the thousands, and at the same time genome-level inference requires that the number of observations  $m$  is large, the problem becomes prohibitive for sequential computers.

In [23], Zola et al. proposed a parallel information theory-based inference method that efficiently exploits multiple levels of parallelism inherent to mutual information computations and uses a generalized scheme for pairwise computation scheduling. The method has been implemented in the MPI-based software package called TINGe (Tool Inferring Networks of Genes), along with a version that supports the use of cell accelerators.

The algorithm proceeds in three stages. In the first stage, input expression profiles are rank-transformed and mutual information is computed for each of the  $\binom{n}{2}$  pairs of genes, and  $q$  randomly chosen permutations per pair. Rank transformation substitutes a gene expression profile with a permutation of  $\{1, \dots, m\}$  by replacing a gene expression with its rank among all gene expressions within the same expression profile. It has been shown that mutual information is invariant under this transformation [5]. By applying it the algorithm can reduce the total number of mutual information estimations between gene expression vectors and their random permutations [22]. In the second phase, the threshold value above which mutual information is considered to signify dependence is computed, and edges below this threshold are discarded. The threshold is computed by finding the element with rank  $(1 - \varepsilon) \cdot q \cdot \binom{n}{2}$  among  $q \cdot \binom{n}{2}$  values contributed by permutations generated

in the earlier stage, where  $\varepsilon$  is specifies the desired statistical significance of the corresponding permutation test. Finally, in the third stage data processing inequality is applied with the consequence that if  $g_i$  interacts with  $g_k$  via some other gene  $g_j$  then  $\mathcal{I}(X_i; X_k) \leq \min(\mathcal{I}(X_i; X_j), \mathcal{I}(X_j; X_k))$ .

The algorithm represents gene network using the standard adjacency matrix  $D_{n \times n}$ . The input and output data are distributed row-wise among  $p$  processors. Each processor stores up to  $\lceil \frac{n}{p} \rceil$  consecutive rows of matrix  $Y$  and the same number of consecutive rows from matrix  $D$ . Matrix  $D$  is then partitioned into  $p \times p$  blocks of submatrices which are computed in  $\lceil \frac{p+1}{2} \rceil$  iterations, where in iteration  $i$  processor with rank  $j$  computes submatrix  $D_{j,(j+i) \bmod p}$ . To implement the second stage a simple reduction operation is used to find the threshold value followed by pruning of matrix  $D$ . Finally, removing of indirect interactions is performed based on streaming of matrix  $D$  in  $p - 1$  communication rounds, where in iteration  $i$  only processors with ranks lower than  $p - i$  participate in communication and computation.

In their method, Zola et al. use B-spline mutual information estimator which is implemented to take advantage of SIMD extensions of modern processors. However, any mutual information estimator could be used. Furthermore, they report modification of the first stage of the algorithm that enables execution on cell heterogeneous processors. The method has been used to reconstruct a 15,222 network of the model plant *Arabidopsis thaliana* from 3,137 microarray experiments in 30 minutes on a 2,048 core IBM Blue Gene/L, and in 2 h and 25 min on a 8-node QS20 cell blade cluster.

S

### Approaches Based on Bayesian Networks

Bayesian networks are a class of graphical models that represent probabilistic relationships among random variables of a given domain. Formally, a Bayesian network is a pair  $(N, P)$ , where  $P$  is the joint probability distribution and  $N$  is a directed acyclic graph, with vertices representing random variables and edges corresponding to “parent – child” relationship between variables, which encodes the Markov assumption that a node is conditionally independent from its non-descendants, given its parents in  $N$ . Under this assumption one can

represent the joint probability as a product of conditional probabilities:

$$P(X_1, \dots, X_n) = \prod_i P(X_i | \pi_i),$$

where  $\pi_i$  is a set of parents of  $X_i$  in  $N$ . Given a set of realizations (observations) of random variables one can learn a structure of the Bayesian network that best fits the observed data.

Bayesian networks have been widely employed for reverse-engineering of gene regulatory networks [8, 18, 21]. Following the same formalization as for information theoretic approaches described above, the problem of gene network inference becomes that of learning the structure of the corresponding Bayesian network. Nodes of the network are random variables assigned to genes  $\mathcal{X} = \{X_1, \dots, X_n\}$ , expression profiles are realizations of those variables, and a Bayesian network learned from such data represents a gene regulatory network, where  $\pi_i$  is interpreted as a set of regulators of gene  $g_i$ . In order to learn the structure of a Bayesian network, a statistically motivated scoring function that evaluates the posterior probability of a network given the input data is typically assumed:  $Score(N) = \log P(N|Y)$ . To find the optimal network efficiently such a function should be decomposable into individual score contributions  $s(X_i, \pi_i)$ , i.e.:

$$Score(N) = \sum_i s(X_i, \pi_i).$$

A major difficulty in Bayesian network structure learning is the super exponential search space in the number of random variables – for a set of  $n$  variables there exist  $\frac{n!2^{\frac{n}{2}(n-1)}}{r \cdot z^n}$  possible directed acyclic graphs, where  $r \approx 0.57436$  and  $z \approx 1.4881$ .

### Parallel Exact Structure Learning

Even assuming that the cost of evaluating scoring function is negligible, exhaustive enumeration of all possible network structures remains prohibitive. Although heuristics have been proposed to tackle the problem, e.g., based on simulated annealing, oftentimes reconstructing the optimal network is advantageous as it enables more meaningful conclusions.

Nikolova et al. [17] proposed an elegant parallel exact algorithm for learning Bayesian networks that builds on top of earlier sequential methods [18]. In this

approach a network is represented as a permutation of nodes where each node is preceded by its parents. The algorithm identifies the optimal ordering, and a corresponding optimal network, using a dynamic programming approach that operates on the lattice formed on the power set of  $\mathcal{X}$  by the partial order “set inclusion.” The lattice is organized into  $n + 1$  levels, where level  $l \in [0, n]$  contains all subsets of size  $l$ , and a node at level  $l$  has  $l$  incoming and  $n - l$  outgoing edges. At each node of the lattice  $(n - l)$  evaluations of individual scores  $s$  have to be performed as a part of dynamic programming search, which next have to be communicated along outgoing edges.

The key component of the approach by Nikolova et al. is the observation that dynamic programming lattice forms an  $n$ -dimensional hypercube that can be decomposed on  $p = 2^k$  processors into  $2^{n-k}$   $k$ -dimensional hypercubes, each mapping to  $p$  processors. These hypercubes can be processed in a pipelined fashion that provides a work optimal algorithm.

The reported method has been used to reverse-engineer regulatory networks using synthetic data with up to 30 genes and 500 microarray observations, and applying Minimum Description Length principle [11] as a scoring function. To reconstruct a network for the largest data it took 1 h and 30 min on 1,024 processors of an IBM BlueGene/L.

### Approaches Based on Differential Equations

Under several simplifying assumptions the dynamics of a gene's expression can be modeled as a function of abundance of all other genes and the rate of degradation:

$$\dot{x}_i = f_i(\mathbf{x}) - \lambda(x_i),$$

where  $f_i$  is called input function of gene  $i$ ,  $x_i$  is expression of gene  $i$ , vector  $\mathbf{x}$  represents expression levels of all genes, and  $\lambda$  describes the rate of degradation. One can further assume that input functions are linear and in such cases the dynamics of the entire system can be represented as:

$$\dot{\mathbf{x}} = A \cdot \mathbf{x},$$

where  $A_{n \times n}$  is a matrix describing influences of genes on each other (including rates of degradation), i.e.,  $A[i, j]$  represents the influence of gene  $g_j$  on  $g_i$ . Consequently

by solving such system of equations one would obtain the underlying gene network represented by matrix  $A$ . Unfortunately finding the solution requires a large number of measurements of  $\mathbf{x}$  and  $\dot{\mathbf{x}}$  since otherwise the system is greatly underdetermined. At the same time obtaining representative measurements is experimentally very challenging and sometimes infeasible.

To overcome this limitation, and to enable linearization of systems with nonlinear gene input functions, Gardner et al. [9] proposed an approach in which  $m$  perturbation experiments (i.e., experiments in which expression of selected genes is affected in a controlled way) are performed, and resulting expression profiles are used to write the following system of differential equations:

$$\dot{Y} = AY + U,$$

which at steady state gives:

$$AY = -U.$$

Here  $Y_{n \times m}$  is a matrix describing expression of all genes in all experiments, i.e.,  $Y[i,j]$  describes expression of gene  $i$  under perturbation  $j$ , and matrix  $U_{n \times m}$  represents the effect of perturbations on every gene in all  $m$  experiments. Because in the majority of cases  $m < n$  and the resulting system remains underdetermined, Gardner et al. assumed that each gene can have at most  $k$  regulatory inputs (which is biologically plausible), and then applied multiple regression for every possible combination of  $k$  regulators, choosing the one that best fits the data to approximate  $A$ .

## Parallelization of Multiple Regression Algorithms

The approach of Gardner et al., named Network Identification by Multiple Regression, is computationally prohibitive for networks with more than a few dozen genes as it is infeasible to consider all  $\binom{n}{k}$  combinations of regulators, especially that for each gene and each combination of regulators the following expression must be evaluated

$$\hat{\mathbf{a}}_i = -\mathbf{u}_i Z^T (ZZ^T)^{-1},$$

to find the combination for which  $\hat{\mathbf{a}}_i$  minimizes the sum squared errors with respect to the observed data. Here,  $\mathbf{u}_i$  represents row of matrix  $U$  for gene  $i$ , and  $Z$  consists

of  $k$  rows selected from  $Y$  that correspond to  $k$  selected regulator genes.

In [10], Gregoretti et al. describe parallel implementation of the algorithm that replaces the exact enumeration of all combinations of regulators with a greedy search heuristic that starts with  $d$  best candidate regulators that are next iteratively combined with other genes to form the final set of regulators. Furthermore, they observe that it is possible to obtain  $\hat{\mathbf{a}}_i$  by solving the system of linear equations  $S\hat{\mathbf{a}}_i = -\mathbf{r}$ , where  $S$  is the symmetric submatrix of  $YY^T$  of  $k$  rows and columns that correspond to the considered regulators, and  $\mathbf{r}$  is the  $i$ -th row of  $Y^T$ . Because  $S$  is positive definite the Cholesky factorization can be used to efficiently solve this system. In practice, the implementation uses the PPSV routine, which has multiple parallel implementations. Finally, searching for regulators of a gene can be performed independently for every gene. Consequently, in the parallel version each of  $p$  processors is assigned at most  $\lceil \frac{n}{p} \rceil$  genes for which it executes the search heuristic.

The main limitation of the multiple regression algorithm is that it requires perturbation data as an input. Because in many cases such data is not available, Gregoretti et al. used synthetic data consisting of 2,500 genes with a single perturbation experiment for each gene. This data has been analyzed on a cluster with 100 nodes, each with dual core Itanium2 processor, and with Quadrics ELAN 4 interconnect in approximately 3 h and 25 min.

## Future Directions

The problem of reverse engineering gene regulatory networks is one of many in the broad area of computational systems biology, and parallel processing only recently attracted attention of systems biology researchers. Together with the rapid progress in high-throughput biological technologies one can expect accumulation of massive and diverse data, which will enable more complex and realistic models of regulation. Most likely these models will be evolving such as to enable *in silico* simulation of biological systems, which is one of the goals of the emerging field of synthetic biology. Consequently, parallel processing in its various flavors, ranging from accelerators and multicore

processors to clusters, grids, and clouds, will be necessary to tackle the resulting computational complexity.

## Bibliographic Notes and Further Reading

The textbook by Alon [1] and articles by Kitano [12, 13] and Murali and Aluru [16] provide a good general introduction to systems biology. A brief overview of existing approaches and challenges in gene networks inference can be found in [2, 14]. In [7] Friedman gives an introduction to using graphical models for network inference, and Meyer et al. review information theoretic approaches in [15]. The “Dialogue for Reverse Engineering Assessments and Methods” project [19] provides a robust set of benchmark data that can be used to assess the quality of inference methods, and it is a good source of information about developments in the area of networks reconstruction.

## Bibliography

1. Alon U (2006) An Introduction to Systems Biology: Design Principles of Biological Circuits. Chapman & Hall/CRC, Boca Raton
2. Bansal M, Belcastro V, Ambesi-Impiombato A, di Bernardo D (2007) How to infer gene networks from expression profiles. Mol Syst Biol 3:78
3. Basso K, Margolin AA, Stolovitzky G, Klein U, Dalla-Favera R, Califano A (2005) Reverse engineering of regulatory networks in human B cells. Nat Genet 37(4):382–390
4. Butte AJ, Kohane IS (2000) Mutual information relevance networks: functional genomic clustering using pairwise entropy measurements. In Pacific Symposium on Biocomputing, pp 418–429
5. Cover TM, Thomas JA (2006) Elements of Information Theory, 2nd edn. Wiley, New York
6. Faith JJ, Hayete B, Thaden JT, Mogno I, Wierzbowski J, Cottarel G, Kasif S, Collins JJ, Gardner TS (2007) Large-scale mapping and validation of *Escherichia coli* transcriptional regulation from a compendium of expression profiles. PLoS Biol 5(1):e8
7. Friedman N (2004) Inferring cellular networks using probabilistic graphical models. Science 303:799–805
8. Friedman N, Linial M, Nachman I, Pe'er D (2000) Using Bayesian networks to analyze expression data. J Comput Biol 7:601–620
9. Gardner TS, di Bernardo D, Lorenz D, Collins JJ (2003) Inferring genetic networks and identifying compound mode of action via expression profiling. Science 301(5629):102–105
10. Gregoretti F, Belcastro V, di Bernardo D, Oliva G (2010) A parallel implementation of the network identification by multiple regression (NIR) algorithm to reverse-engineer regulatory gene networks. PLoS One 5(4):e10179
11. Grunwald PD (2007) The Minimum Description Length Principle. MIT Press, Cambridge
12. Kitano H (2002) Computational systems biology. Nature 420(6912):206–210
13. Kitano H (2002) Systems biology: a brief overview. Science 295(5560):1662–1664
14. Margolin A, Califano A (2007) Theory and limitations of genetic network inference from microarray data. Ann N Y Acad Sci 1115:51–72
15. Meyer PE, Kontos K, Lafitte F, Bontempi G (2007) Information-theoretic inference of large transcriptional regulatory networks. EURASIP J Bioinform Syst Biol 2007:79879
16. Murali TM, Aluru S (2009) Algorithms and Theory of Computation Handbook, chapter Computational Systems Biology. Chapman & Hall/CRC, Boca Raton
17. Nikolova O, Zola J, Aluru S (2009) A parallel algorithm for exact Bayesian network inference. In IEEE Proceedings of the International Conference on High Performance Computing (HiPC 2009), pp 342–349
18. Ott S, Imoto S, Miyano S (2004) Finding optimal models for small gene networks. In Pacific Symposium on Biocomputing, pp 557–567
19. Prill RJ, Marbach D, Saez-Rodriguez J, Sorger PK, Alexopoulos LG, Xue X, Clarke ND, Altan-Bonnet G, Stolovitzky G (2010) Towards a rigorous assessment of systems biology models: the DREAM3 challenges. PLoS One 5(2):e9202
20. Schäfer J, Strimmer K (2005) An empirical Bayes approach to inferring large-scale gene association networks. Bioinformatics 21(6):754–764
21. Yu J, Smith V, Wang PP, Hartemink AJ, Jarvis ED (2004) Advances to Bayesian network inference for generating causal networks from observational biological data. Bioinformatics 20(18):3594–3603
22. Zola J, Aluru M, Aluru S (2008) Parallel information theory based construction of gene regulatory networks. In Proceedings of the International Conference on High Performance Computing (HiPC 2008). LNCS, vol 5375, pp 336–349
23. Zola J, Aluru M, Sarje A, Aluru S (2010) Parallel information-theory-based construction of genome-wide gene regulatory networks. IEEE Trans Parallel Distributed Syst 21(12):1721–1733

## Systolic Architecture

### ► Systolic Arrays

## Systolic Arrays

JAMES R. REINDERS  
Intel Corporation, Hillsboro, OR, USA

## Synonyms

Instruction systolic arrays; Processor arrays; Systolic architecture; Wavefront arrays

## Definition

A *Systolic Array* is a collection of processing elements, called cells, that implements an algorithm by rhythmically computing and transmitting data from cell to cell using only local communication. Cells of a Systolic Array are arranged and connected in a regular pattern with a design that emphasizes a balance between computational and communicational capabilities. Systolic Arrays have proven particularly effective for real-time applications in signal and image processing.

*Systolic Algorithms* are algorithms specifically designed to make effective use of Systolic Arrays. Systolic Algorithms have also been shown to make particularly efficient use of many general-purpose parallel computers.

The name *Systolic Arrays* derives from an analogy with the regular pumping of blood by the heart. Systolic, in medical terms, refers to the phase of blood circulation in which the pumping chambers of the heart, ventricles, are contracting forcefully and therefore blood pressure is at its highest.

## Discussion

### Background: Motivated by Emergence of VLSI

Systolic Arrays, first described in 1978 by H. T. Kung and Charles E. Leiserson, were originally described as a systematic approach to take advantage of rapid advances in VLSI technology and coping with difficulties present in designing VLSI systems. The simplicity and regularity of Systolic Arrays lead to a cheaper VLSI implementation as well as higher chip density.

Systolic Arrays were originally proposed for VLSI implementation of some matrix operations that were shown to have efficient solutions thereafter known as Systolic Algorithms. Systolic Algorithms support high degrees of concurrency while requiring only simple, regular communication and control, which in turn allows for efficient implementation in hardware.

Early VLSI offered high degrees of integration but no speed advantage over the decade-old TTL technology. While tens of thousands of gates could be integrated on a single chip, it appeared that computational speed would only come from the concurrent use of many processing elements. In 1982,

H. T. Kung wrote, "Since the technological trend clearly indicates a diminishing growth rate for component speed, any major improvement in computation speed must come from the concurrent use of many processing elements."

Furthermore, large-scale designs were pushing the limits of the methodologies of the day. It was observed at the time that the general practice of ad hoc designs for VLSI systems was not contributing to sufficient accumulation of experiences, and errors were often repeated. By providing general guidelines, the concept of a Systolic Array, a general methodology emerged for mapping high-level computations into hardware structures.

Cost-effectiveness emerged as a chief concern with special-purpose systems; costs need to be low enough to justify construction of any device with limited applicability. The cost of special-purpose systems can be greatly reduced if a limited number of simple substructures or building blocks, such as a cell, can be reused repeatedly for the overall design.

The VLSI implications of Systolic Array designs proved to be substantial. Spatial locality divides the time to design a chip by the degree of regularity. Locality avoids the need for long and therefore high capacitive wires, temporal regularity and synchrony reduces control issues, pipelinability yields performance, I/O closeness holds down I/O bandwidth, and modularity allows for parameterized designs.

Special-purpose systems pushed the limits of technology in order to achieve the performance needed to justify their cost. Systolic Arrays embraced VLSI circuit technology to achieve high performance via parallelism, honoring the scarcity of power and resistive delay by using a communication topology devoid of long inter-processor wires. Such communication topologies require chip area that is only linear in the number of processors. Systolic Arrays also embraced the design economics of special-purpose processors by reusing a limited number of cell designs.

Systolic Arrays proved especially well suited for processing data from sensor devices as an attached processor. Stringent time requirements of real-time signal processing and large-scale scientific computation strongly favor special-purpose devices that can be built in a cost-effective and reliable fashion because they deliver the performance needed.

## Concept

Unlike general-purpose processors, a Systolic Array is characterized by its regular data flow. Typically, two or more data streams flow through a Systolic Array in various speeds and directions. The crux of the Systolic Array approach is that once a stream of data is formed, it can be used effectively by each processing element it passes. A higher computational throughput is therefore achieved as compared with a general-purpose processor in which its computational speed may be limited by the I/O bandwidth. When a complex algorithm can be decomposed to fine-grained, regular operations, each operation will then be simpler to implement.

A Systolic Array consists of a set of interconnected cells, each capable of performing at least simple computational operations, and communicating only with cells in close proximity. Simple, regular communication and control structures offer substantial advantages over complicated ones in both design and implementation. Many shapes for an “array” are possible, and have been proposed including triangular, but simple two-dimensional meshes, or tori, have dominated as Systolic Arrays have tended to become more general because of the flexibility and simplicity of meshes and tori (Fig. 1).

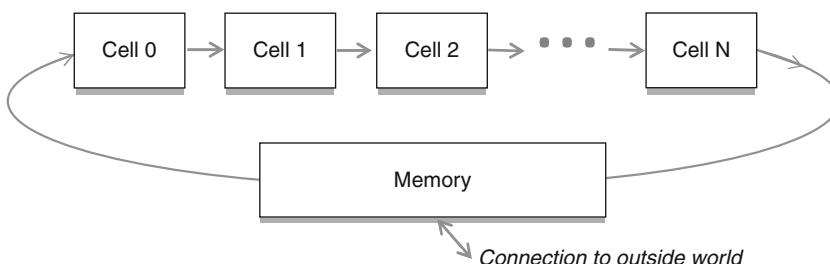
Systolic Arrays are specifically designed to address the communication requirements of parallel computer systems by placing an emphasis on strong connections between computation and communication in order to achieve both balance and scaling. Systolic Arrays directly address the importance of the communication system for scalable parallel systems by providing direct paths between the communication system and the computational units. Systolic Algorithms address the need for “balanced algorithms” to best utilize parallelism.

In a Systolic Array, data flows from the computer memory in a rhythmic fashion, passing through many

processing cells before it returns to memory, much as blood circulates into and out of the heart. The system works like an assembly line where many people work on the same automobile at different times and many cars are assembled simultaneously. The network for the flow of data can offer different degrees of parallelism, and data flow itself may be at different speeds and multiple directions. Traditional pipelined systems flow only results, whereas a Systolic Array flow includes inputs and partial results.

The essential characteristic of a Systolic Array is an emphasis on balance between computational and communicational capabilities, and the scalability of practical parallel systems. The features of Systolic Arrays, in pursuit of this emphasis on balance and scalability, are generally as follows:

- Spatial regularity and locality: the variety of processing cells is limited, and connections are limited to nearby processors. Cells are not connected via shared busses, which would not scale due to contention. There is neither global broadcasting nor global memory.
- Temporal regularity and synchrony: each cell acts as a finite state transducer. Cells do not need to execute the same program.
- Pipelinability: a design of  $N$  cells will exhibit a linear speedup  $O(N)$ .
- I/O closeness: only cells on the boundaries of the array have access to “the outside world” to perform I/O.
- Modularity/scaling: a larger array can handle a larger instance of a problem than the smaller version of which it is an extension. Replacing a processing cell with an array of cells offers a higher computation throughput without increasing memory bandwidth. This realization of parallelism offers the advantages



**Systolic Arrays. Fig. 1** Simple linear systolic array configuration

of both increased performance from increased concurrency and increased designer productivity from component reuse.

Synchronous global clocking is not a requirement of Systolic Arrays despite it being a property of many early implementations. Systolic Arrays are distinguished by their pursuit of both balance and scaling with a design emphasis on strong connections between computation and communication in order to achieve both.

### Importance of Interconnect Design

A central issue for every parallel system is how the computational nodes of a parallel computer communicate with other nodes. There are problems that have been dubbed “embarrassingly parallel,” where little or no communication is necessary between the multiple processors performing subsets of the problem. For those applications, the computation speed alone will determine the speed of execution on a parallel system. There are numerous important problems that are not embarrassingly parallel, in which communication plays a critical role in determining the effective speed of execution as well as the degree to which the problem can scale to utilize parallelism.

The exploration of “balanced algorithms” looked to find algorithms that scale without bounds. Such algorithms are marked by a constant ratio of computation to communication steps. These algorithms became known as “Systolic Algorithms.” While Systolic Algorithms can be mapped to a wide range of hardware, it was found that these algorithms tended to rely on finer and finer-grained communication as machine sizes increased. To efficiently deal with this fine-grained parallelism, communication with little to no overhead is needed.

Systolic Arrays accomplish this goal by providing a method to directly couple computation to communication. In such machines, the design will seek to match the communicational capabilities to the computational capabilities of the machine.

Systolic Arrays are one approach to addressing the communication requirements of parallel systems. Systolic Arrays acknowledge the importance of the communication system for scalable parallel systems and provide direct paths between the communication system and the computational units. The key concepts

behind a Systolic Array are the balance between computational and communicational capabilities, and the scalability of practical parallel systems. A balanced design minimizes inefficiencies as measured by underutilization of portions of a system due to stalls and bottlenecks. A scalable design allows for expanding performance by increasing the number of computational nodes in a system. Regularity, an often-noted characteristic of a Systolic Array, is a consequence of the scalability goal and not itself part of the definition of a Systolic Array. Another advantage of regularity is that the system can be scaled by repeating a common subsystem design, a benefit to designers that has reinforced use of regularity as a way to accomplish scalability.

The Systolic Array approach initially led to very rigidly synchronous hardware designs that, while elegant, proved overly constraining for many programming solutions. While balance is desirable, the tight coupling of systolic systems is not always desirable. Designs of more programmable Systolic Arrays worked to preserve the benefits of systolic communication while generalizing the framework to allow for more application diversity.

With tight coupling, any stall in communication will stall computation and vice versa. For many applications, a looser coupling that allowed coupling at a level of data blocks instead of individual data elements was advantageous. Looser coupling also leads to increased ability to overlap communication and computation in practice. As a result, the idea of Systolic Arrays evolved from an academic concept to realization in microprocessors such as the CMU/Intel iWarp.

### Variations

Increasing the degree of independence of individual processors in an array adds complexity for flexibility and affects the efficiency and performance of an array. One design consideration is whether individual processors have a local control store or a design where instructions were delivered via the processor interconnects to be executed upon arrival. The latter was sometimes referred to as an Instruction Systolic Array (ISA). Synchronous broadcasting of instructions to an array, as opposed to the flow of instructions via the interconnect, would be inconsistent with the goals of Systolic Array designs because it would introduce long paths and the associated delays. Processors with individual control

stores have been referred to as *programmable Systolic Arrays*, and therefore incur some overhead for the initial loading of the control stores.

As a design methodology for VLSI, silicon implementations were generally hard coded to a large degree once a design was determined. The resulting array performed a fixed function and offered no ability to be loaded with a new algorithm for other functions. Such designs could be optimized to offer the most efficient implementations with the least flexibility for future changes by customizing the cells and the connection networks. More flexible designs offered more design reuse, and thereby offered the opportunity to amortize development costs over multiple uses in exchange for some loss of efficiency, increase in silicon size and generally some loss of array performance.

The most flexible implementations of Systolic Arrays were machines developed principally for academic and research purposes to host the exploration and demonstration of Systolic Algorithms. The Warp and iWarp machines, developed under the guidance of H. T. Kung at Carnegie Mellon University, were such machines.

## Systolic Algorithms

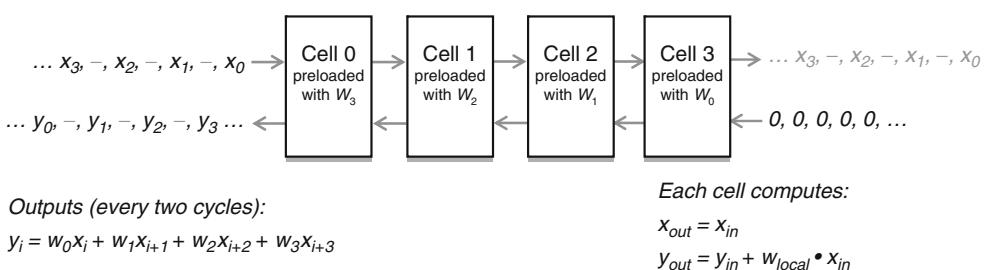
Systolic Arrays have garnered substantial attention in the development of Systolic Algorithms, which have proven to be efficient at creating solutions suited to the

demands of real-time systems in terms of reliability and the ability to meet stringent time constraints. Development of error detections and fault tolerance in Systolic Algorithms has also received substantial attention.

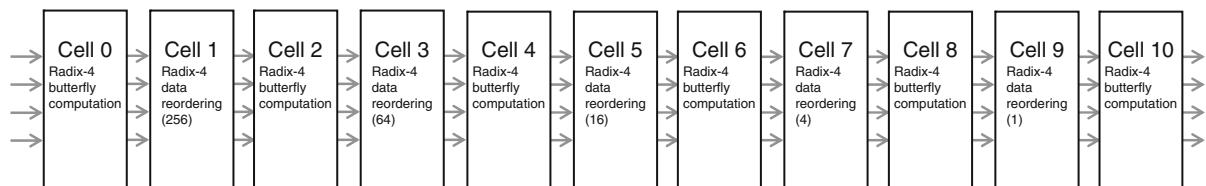
Systolic Algorithms are among the most efficient class of algorithms for massively parallel implementation. A systolic algorithm can be thought of as having two major parts: a cell program and a data flow specification. The cell program defines the local operations of each processing element, while the data flow describes the communication network and its use (Fig. 2).

One nice property of a Systolic Algorithm is that each processor communicates only with a few other processors. It is thus suitable for implementation on a cluster of computers in which we seek to avoid costly global communication operations. Systolic Algorithms require preservation of data ordering within a stream but do not require that streams of data proceed in lockstep as they would have in the earliest hardware implementations of Systolic Arrays.

A Systolic Algorithm will make multiple uses of each input data item, make extensive use of concurrency, rely on only a few simple cell types, and utilize simple and regular data and control flows. Bottlenecks to computational speedups are often caused by limited system memory bandwidths, known as von Neumann bottlenecks, rather than limited processing capabilities (Fig. 3).



**Systolic Arrays. Fig. 2** Systolic array implementation for a convolution product



**Systolic Arrays. Fig. 3** Systolic array implementation for a 4096-point FFT

Systolic Algorithms have been shown to have wide applicability. They utilize a computational model for a wide range of parallel processing structures not limited to the initially targeted special-purpose needs, and which offer efficient operations on parallel computers in general, not just special-purpose designs. For instance, linear algebra algorithms and FFT algorithms are computationally demanding, especially when high throughput rates are expected, and they display a high degree of regularity. These algorithms are ideal candidates for parallel implementation via Systolic Algorithms.

### Systolic Array Machines

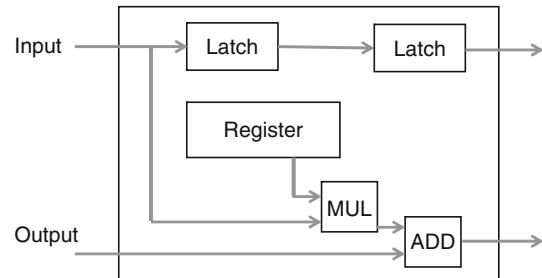
The Colossus Mark II, built in 1944, has been cited as the first digital computer known to have used a technique similar to Systolic Arrays.

In 1978, H. T. Kung and Charles E. Leiserson published their concept of Systolic Arrays. They made systematic use of ideas that were already present in older architectural paradigms in order to respond to the new challenges posed by VLSI technology. This sparked much interest and many designs for Systolic Arrays. While there have been many designs, only a handful have been actually built and put into use. Here is a brief review of key systems in chronological order.

The earliest Systolic Arrays were introduced as a straightforward way to create scalable, balanced, systems for special-purpose computations, by directly embodying a systolic algorithm in the hardware design. These early Systolic Arrays were very limited in applications because the algorithm was expressed in efficient and special-purpose designs. Changing an algorithm implemented as a Systolic Array may be difficult or impossible without redesigning the hardware, particularly the communication interconnects. More general processors used in parallel have been shown to be able to perform a broad range of applications, but are often too large, too slow, or too expensive due to inefficiencies imposed by communication overhead.

### Systolic Convolution Chip

The systolic convolution chip was created at CMU in 1979 to solve finite-sized two-dimensional convolutions. All nodes in a system performed the same operation while data flowed through the systems in a completely regular synchronous manner (Fig. 4).



**Systolic Arrays.** Fig. 4 2-D convolution systolic system cell architecture

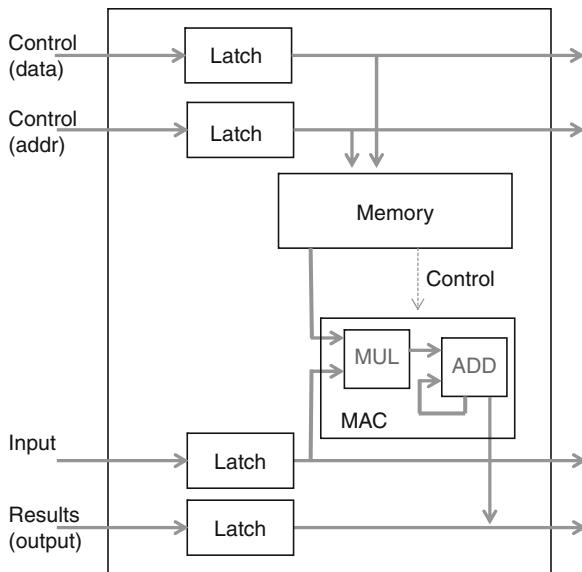
### ESL Systolic Processor

A Systolic Array to compute convolutions and other signal processing computations was designed and implemented at ESL and operational by 1982. Seven nodes fit on a board measuring 37 cm by 40 cm using discrete high-performance components with each node capable of ten million operations per second. Up to five boards could be linked together in a system, which was attached to a VAX 11/780 host machine. Accessing the capabilities of the Systolic Array from a high-level language was a key innovation of this Systolic Array. A Fortran program viewed capabilities as a function to call with information on the kernel to execute the input data and where to store the output data. The ESL systolic processor was able to greatly expand the application domain for which it was suited by including local memory to hold more kernel elements and a control unit to create addresses to access data from memory. This systolic system could perform 1-D and 2-D convolutions, matrix multiplication, and Fourier and cosine transforms (Fig. 5).

S

### NOSC Systolic Array Test Bed

A programmable Systolic Array, formed using standard off-the-shelf Intel 8051 microprocessors, was built by the Naval Ocean Systems Center (NOSC) from 1981 to 1983. The microprocessor operated as the control unit, while a separate arithmetic unit on the board consumed and output data. The arithmetic processor was directly connected to the communication ports so as to tightly link computations and communications. The NOSC test bed contained 64 nodes arranged in an 8 × 8 grid. Each node fits on a 6 × 24-cm board. Each node in the system is connected to five other nodes. The



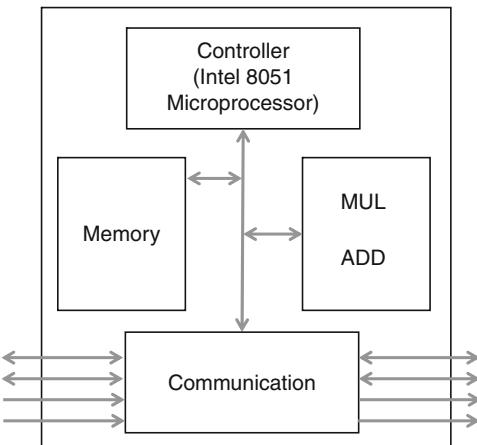
Systolic Arrays. Fig. 5 ESL systolic cell architecture

topology allows for direct implementation of several key Systolic Algorithms such as matrix multiplication by a hexagonal array.

All programming was done in assembly language. Node performance was about 24 K FLOPs for a total performance of 1.5 MFlop for the system. The machine was a test bed for software development, and the design could not extend beyond 64 nodes. NOSC research into Systolic Arrays spanned a number of machines from 1979 to 1991 including the Systolic Array Processor (SAP), the Systolic Linear Algebra Parallel Processor (SLAPP), the Video Analysis Transputer Array (VATA), and the High-Speed Systolic Array Processor (HISSAP) test bed. Subsequent algorithm development moved to iWarp and on to general-purpose machines (Fig. 6).

### Programmable Systolic Chip (PSC)

A programmable Systolic Array chip designed at CMU led to a functional nine-node system in 1984. The architecture of the PSC was similar to the NOSC Systolic Array but was organized around three buses instead of one, cells had three input and output ports to attach to other cells. All programming was in assembly language. The machine was used for low-level image processing computations (Fig. 7).



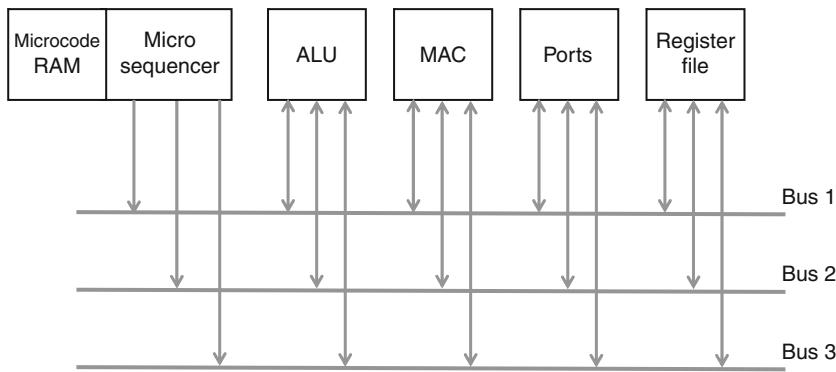
Systolic Arrays. Fig. 6 NOSC test bed cell architecture

### Geometric Arithmetic Processor (GAPP)

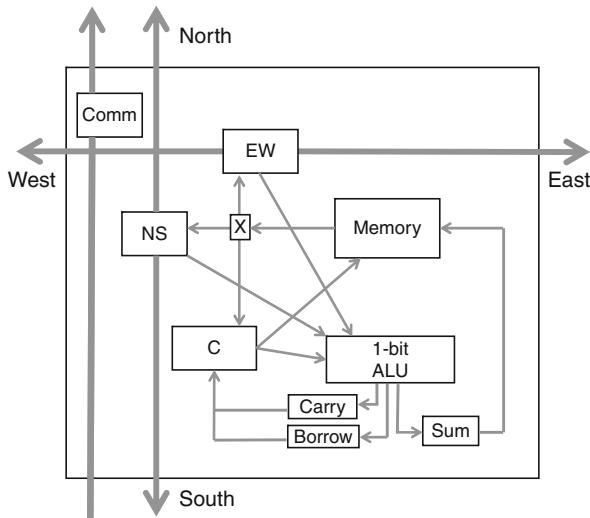
The NCR GAPP programmable Systolic Array is a mesh-connected single-bit cell that communicates directly with neighbors to the North, East, South, and West. It was first implemented as a medium-scale integration (MSI) breadboard in 1982 for a  $6 \times 12$  cell array. GAPP I was a PLA-based  $3 \times 6$  cell chip, GAPP II was a  $6 \times 12$  cell in  $3-\mu m$  CMOS, and finally a version in two-micron CMOS clocked at 10 MHz with 30 MB/s input and 30 MB/s output bandwidth was sold as NCR45CG72 and could perform 28 million eight-bit additions per second. The use of one-bit data paths and one-bit registers minimize the size of a single cell. Local memory on a node was only 128 bits. The machine broadcast long instruction words to control the machine in pure lockstep SIMD. Programming was originally done in STOIC, a variant of Forth, but later development of an “Ada-like” language compiler eased programming to create important libraries of commonly used functions. VAX, IBM PC-AT, and Sun 3 workstation host support existed. Military and space systems leveraged GAPP for use in image processing, and the construction of the largest array of processing elements of their generation (82,944 processing cells in one system reported in 1988) (Fig. 8).

### Warp and iWarp

The Warp project (1984–1993) was a series of increasingly general-purpose programmable Systolic Array



Systolic Arrays. Fig. 7 PSC cell architecture



Systolic Arrays. Fig. 8 Geometric arithmetic cell architecture

systems and related software, created by Carnegie Mellon University (CMU) and developed in conjunction with industrial partners G.E., Honeywell, and Intel with funding from the US Defense Advanced Research Projects Agency (DARPA) (see Warp and iWarp). Warp was a highly programmable Systolic Array computer with a linear array of ten or more cells, capable of performing ten million single-precision floating-point operations per second (10 MFLOPS). A ten-cell machine had a peak performance of 100 MFLOPS. The iWarp machines doubled this performance, delivering 20 MFLOPS single-precision and supporting double-precision floating point at half the performance.

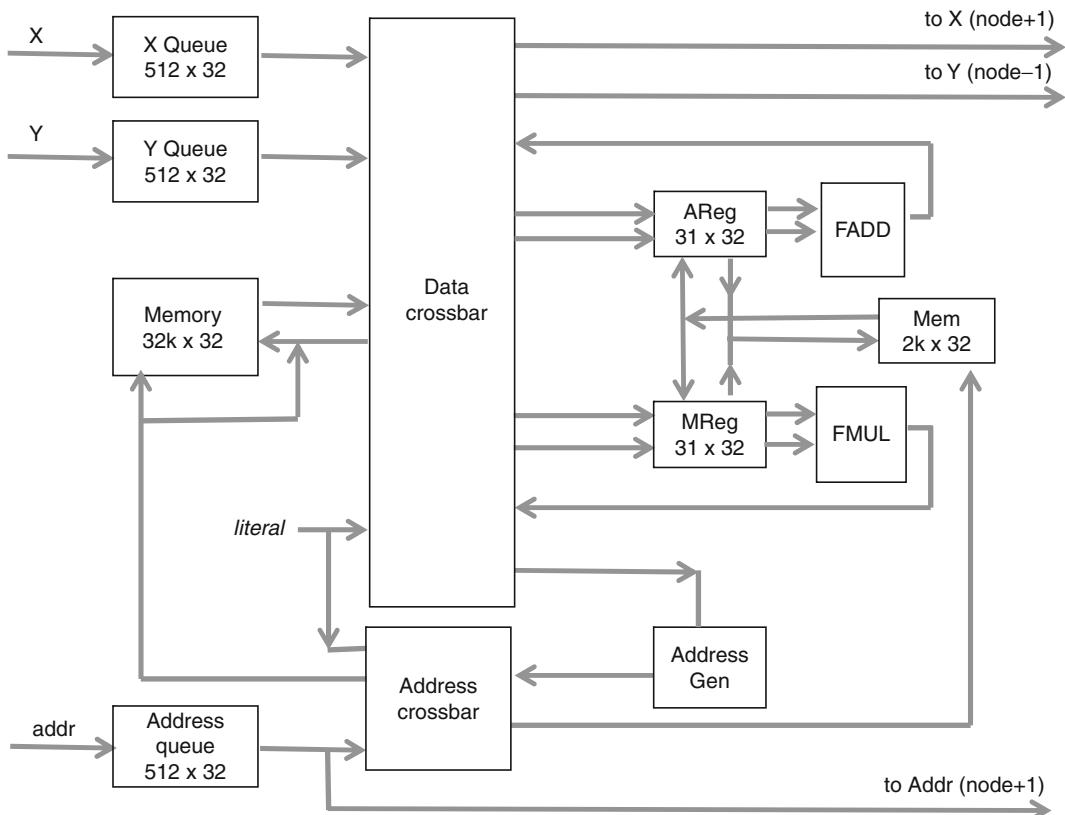
iWarp was based on a full custom VLSI component integrating a 700,000 transistor LIW microprocessor, a network interface, and a switching node into one single chip of  $1.2 \times 1.2 \text{ cm}$  silicon. The processor dissipated up to 15 watts and was packaged in a ceramic pin grid array with 280 pins. Intel marketed the iWarp with the tag line “Building Blocks for GigaFLOPs.” The standard iWarp machines configuration arranged iWarp nodes in a  $2m \times 2n$  torus. All iWarp machines included the “back edges” and, therefore, were tori.

Warp and iWarp were programmed using high-level languages and domain-specific program generators. About 20 Warp machines were built, and more than 1,500 iWarp processors were manufactured. Warp and iWarp handled a large number of image and signal processing algorithms (Figs. 9 and 10).

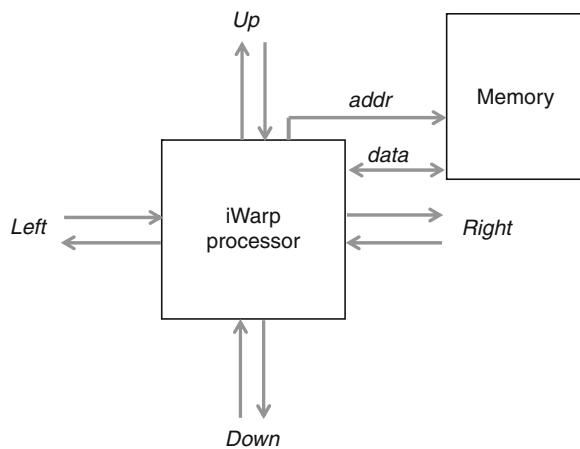
## Future Directions

Faced with relatively little growth in transistor speeds coupled with ever-expanding transistor densities in the late 1970s and early 1980s, the use of transistors for many forms for parallelism, from multiple cells to LIW, were urgently explored. Investigations in Systolic Arrays gave rise to fruitful exploration of Systolic Algorithms. Transistor speeds did finally explode, which helped divert interest from parallelism including Systolic Arrays and Systolic Algorithms.

By 2005, once again clock speed gains dramatically slowed, while transistor densities continued growing in accord with Moore’s Law as much as they had at the dawn of VLSI. Hardware parallelism, this time as multicore processors, and parallel programming have



**Systolic Arrays.** Fig. 9 PC-warp cell architecture



**Systolic Arrays.** Fig. 10 iWarp cell architecture

once again found widespread interest. Single processor designs have become the cells in multicore and many core processor designs, and interconnect debates are being revisited. Interconnection of many processor

cores faces the same challenges Systolic Arrays worked to solve the first time the industry found transistor density on the rise without transistor performance moving nearly as quickly, and Systolic Algorithms proved their usefulness via programmability.

### Related Entries

- ▶ VLIW Processors
- ▶ Warp and iWarp

### Bibliographic Notes and Further Reading

Application-specific solutions, like the early Systolic Arrays, continue today in the annual *IEEE International Conference on Application-specific Systems, Architectures and Processors*. This conference traces its origins back to the International Workshop on Systolic Arrays, first organized in 1986. It later developed into the International Conference on Application-Specific

Array Processors. With its current title, it was organized for the first time in Chicago, USA, in 1996.

Systolic Arrays were first described in 1978 by H. T. Kung and Charles E. Leiserson [3, 4]. VLSI designs took serious note of the concept and the design principles [4, 6]. Digital systems operating in a synchronous fashion utilizing a central clock predated the description of Systolic Arrays and VLSI by many years, the earliest such machine has been cited as the Colossus [3] built in 1944. Additional reading about actual realizations of Systolic Arrays are available[1, 7–13].

## Bibliography

1. Cloud EL (1988) Frontiers of massively parallel computation. In: Proceedings of the 2nd symposium on the frontiers of massively parallel computation, Fairfax, VA, pp 373–381
2. Cragon HG (2003) From fish to colossus: how the German Lorenz cipher was broken at Bletchley park. Cragon Books, Dallas, ISBN: 0-9743045-0-6
3. Frank GA, Greenawalt EM, Kulkarni AV (1982) A systolic processor for signal processing. In: Proceedings of AFIPS '82, June 7–10. ACM, New York, pp 225–231
4. Fisher AL, Kung HT, Monier LM, Dohi Y (1983) Architecture of the PSC: a programmable systolic chip. In: Proceedings of the 10th annual international symposium on computer architecture (ISCA '83), Stockholm, Sweden. ACM, New York, Vol II, Issue 3, pp 48–53
5. Gross T, O'Hallaron DR (1998) iWarp: anatomy of a parallel computing system. MIT Press, Cambridge, MA, 488 p
6. Kung HT, Leiserson CE (1978) Systolic arrays (for VLSI). In: Duff IS, Stewart GW (eds) Proceedings of sparse matrix proceedings 1978. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, pp 256–282
7. Kung HT, Song SW (1981) A systolic 2-D convolution chip. In: Proceedings of 1981 IEEE computer society workshop on computer architecture for pattern analysis and image database management, 11–13 Nov 1981, Hot Springs, Virginia, pp 159–160
8. Kung HT (1982) Why systolic architectures? IEEE Comput 15(1):37–46
9. Kung SY (1988) VLSI array processors. Prentice Hall, Upper Saddle River
10. Mead C, Conway L (1980) Chap. 8, Highly concurrent systems. In: Introduction to VLSI systems. Addison-Wesley series in computer science, Addison-Wesley, Menlo Park, CA, pp 263–332
11. Tirpak FM Jr (1991) Software development on the high-speed systolic array processor (HISSAP): lessons learned. Technical report 1429. Naval Ocean Systems Center, San Diego, CA

# T

## Task Graph Scheduling

YVES ROBERT  
Ecole Normale Supérieure de Lyon, France

### Synonyms

DAG scheduling; Workflow scheduling

### Definition

Task Graph Scheduling is the activity that consists in mapping a task graph onto a target platform. The task graph represents the application: Nodes denote computational tasks, and edges model precedence constraints between tasks. For each task, an assignment (choose the processor that will execute the task) and a schedule (decide when to start the execution) are determined. The goal is to obtain an efficient execution of the application, which translates into optimizing some objective function, most usually the total execution time.

### Discussion

#### Introduction

Task Graph Scheduling is the activity that consists in mapping a task graph onto a target platform. The task graph is given as input to the scheduler. Hence, scheduling algorithms are completely independent of models and methods used to derive task graphs. However, it is insightful to start with a discussion on how these task graphs are constructed.

Consider an application that is decomposed into a set of computational entities, called *tasks*. These tasks are linked by *precedence constraints*. For instance, if some task  $T$  produces some data that is used (read) by another tasks  $T'$ , then the execution of  $T'$  cannot start before the completion of  $T$ . It is therefore natural to represent the application as a *task graph*: The task graph is a DAG (Directed Acyclic Graph), whose nodes are the

tasks and whose edges are the precedence constraints between tasks.

The decomposition of the application into tasks is given to the scheduler as input. Note that the task graph may be directly provided by the user, but it can also be determined by some parallelizing compiler from the application program. Consider the following algorithm to solve the linear system  $Ax = b$ , where  $A$  is an  $n \times n$  nonsingular lower triangular matrix and  $b$  is a vector with  $n$  components:

```
for i = 1 to n do
 Task $T_{i,i}$: $x_i \leftarrow b_i / a_{i,i}$
 for j = i + 1 to N do
 Task $T_{i,j}$: $b_j \leftarrow b_j - a_{j,i} \times x_i$
 end
end
```

For a given value of  $i$ ,  $1 \leq i \leq n$ , each task  $T_{i,*}$  represents some computations executed during the  $i$ -th iteration of the external loop. The computation of  $x_i$  is performed first (task  $T_{i,i}$ ). Then, each component  $b_j$  of vector  $b$  such that  $j > i$  is updated (task  $T_{i,j}$ ). In the original sequential program, there is a total precedence order between tasks. Write  $T <_{seq} T'$  if task  $T$  is executed before task  $T'$  in the sequential code. Then:

$$\begin{aligned} T_{1,1} &<_{seq} T_{1,2} <_{seq} T_{1,3} <_{seq} \cdots <_{seq} T_{1,n} <_{seq} \\ T_{2,2} &<_{seq} T_{2,3} <_{seq} \cdots <_{seq} T_{n,n}. \end{aligned}$$

However, there are independent tasks that can be executed in parallel. Intuitively, independent tasks are tasks whose execution orders can be interchanged without modifying the result of the program execution. A necessary condition for tasks to be independent is that they do not update the same variable. They can read the same value, but they cannot write into the same memory location (otherwise there would be a race condition and the result would be nondeterministic). For instance,

tasks  $T_{1,2}$  and  $T_{1,3}$  both read  $x_1$  but modify distinct components of  $b$ , hence they are independent.

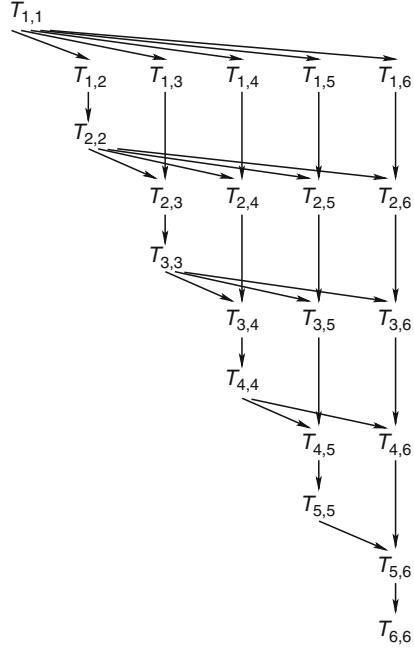
This notion of independence can be expressed more formally. Each task  $T$  has an input set  $\text{In}(T)$  (read values) and an output set  $\text{Out}(T)$  (written values). In the example,  $\text{In}(T_{i,i}) = \{b_i, a_{i,i}\}$  and  $\text{Out}(T_{i,i}) = \{x_i\}$ . For  $j > i$ ,  $\text{In}(T_{i,j}) = \{b_j, a_{j,i}, x_i\}$  and  $\text{Out}(T_{i,j}) = \{b_j\}$ . Two tasks  $T$  and  $T'$  are not independent (write  $T \perp T'$ ) if they share some written variable:

$$T \perp T' \Leftrightarrow \begin{cases} \text{In}(T) \cap \text{Out}(T') \neq \emptyset \\ \text{or } \text{Out}(T) \cap \text{In}(T') \neq \emptyset \\ \text{or } \text{Out}(T) \cap \text{Out}(T') \neq \emptyset \end{cases}$$

For instance, tasks  $T_{1,1}$  and  $T_{1,2}$  are not independent because  $\text{Out}(T_{1,1}) \cap \text{In}(T_{1,2}) = \{x_1\}$ ; therefore  $T_{1,1} \perp T_{1,2}$ . Similarly,  $\text{Out}(T_{1,3}) \cap \text{Out}(T_{2,3}) = \{b_3\}$ , and hence  $T_{1,3}$  and  $T_{2,3}$  are not independent; hence  $T_{1,3} \perp T_{2,3}$ .

Given the dependence relation  $\perp$ , a partial order  $\prec$  can be extracted from the total order  $\prec_{seq}$  induced by the sequential execution of the program. If two tasks  $T$  and  $T'$  are dependent, that is,  $T \perp T'$ , they are ordered according to the sequential execution:  $T \prec T'$  if both  $T \perp T'$  and  $T \prec_{seq} T'$ . The precedence relation  $\prec$  represents the dependences that must be satisfied to preserve the semantics of the original program; if  $T \prec T'$ , then  $T$  was executed before  $T'$  in the sequential code, and it has to be executed before  $T'$  even if there are infinitely many resources, because  $T$  and  $T'$  share a written variable. In terms of order relations,  $\prec$  is defined more accurately, as the transitive closure of the intersection of  $\perp$  and  $\prec_{seq}$ , and captures the intrinsic sequentiality of the original program. Note that transitive closure is needed to track dependence chains. In the example,  $T_{2,4} \perp T_{4,4}$  and  $T_{4,4} \perp T_{4,5}$ , hence a path of dependences from  $T_{2,4}$  to  $T_{4,5}$ , while  $T_{2,4} \perp T_{4,5}$  does not hold.

A directed graph is drawn to represent the dependence constraints that need to be enforced. The vertices of the graph denote the tasks, while the edges express the dependence constraints. An edge  $e : T \rightarrow T'$  in the graph means that the execution of  $T'$  must begin only after the end of the execution of  $T$ , whatever the number of available processors. Transitivity edges are not drawn, as they represent redundant information; only predecessor edges are shown.  $T$  is a predecessor of  $T'$  if



**Task Graph Scheduling.** Fig. 1 Task graph for the triangular system ( $n = 6$ )

$T \prec T'$  and if there is no task  $T''$  in between, that is, such that  $T \prec T''$  and  $T'' \prec T'$ . In the example, predecessor relationships are as follows (see Fig. 1):

- $T_{i,i} \prec T_{i,j}$  for  $1 \leq i < j \leq n$   
(the computation of  $x_i$  must be done before updating  $b_j$  at step  $i$  of the outer loop).
- $T_{i,j} \prec T_{i+1,j}$  for  $1 \leq i < j \leq n$   
(updating  $b_j$  at step  $i$  of the outer loop is done before reading it at step  $i + 1$ ).

In summary, this example shows how an application program can be decomposed into a task graph, either manually by the user, or with the help of a parallelizing compiler.

## Fundamental Results

Traditional scheduling assumes that the target platform is a set of  $p$  identical processors, and that no communication cost is paid. Fundamental results are presented in this section, but only two proofs are provided, that of Theorem 1, an easy result on the efficiency of a schedule, and that of Theorem 4, Graham's bound on list scheduling.

## Definitions

**Definition 1** A task graph is a directed acyclic vertex-weighted graph  $G = (V, E, w)$ , where:

- The set  $V$  of vertices represents the tasks (note that  $V$  is finite).
- The set  $E$  of edges represents precedence constraints between tasks:  
 $e = (u, v) \in E$  if and only if  $u < v$ .
- The weight function  $w : V \rightarrow \mathbb{N}^*$  gives the weight (or duration) of each task. Task weights are assumed to be positive integers.

For the triangular system (Fig. 1), it can be assumed that all tasks have equal weight:  $w(T_{i,j}) = 1$  for  $1 \leq i \leq j \leq n$ . On a contrary, a division could be considered as more costly than a multiply-add, leading to a larger weight for diagonal tasks  $T_{i,i}$ .

A schedule  $\sigma$  of a task graph is a function that assigns a start time to each task.

**Definition 2** A schedule of a task graph  $G = (V, E, w)$  is a function  $\sigma : V \rightarrow \mathbb{N}^*$  such that  $\sigma(u) + w(u) \leq \sigma(v)$  whenever  $e = (u, v) \in E$ .

In other words, a schedule must preserve the *dependence constraints* induced by the precedence relation  $<$  and embodied by the edges of the dependence graph; if  $u < v$ , then the execution of  $u$  begins at time  $\sigma(u)$  and requires  $w(u)$  units of time, and the execution of  $v$  at time  $\sigma(v)$  must start after the end of the execution of  $u$ . Obviously, if there was a cycle in the task graph, no schedule could exist, hence the restriction to acyclic graphs (DAGs).

There are other constraints that must be met by schedules, namely, *resource constraints*. When there is an infinite number of processors (in fact, when there are as many processors as tasks), the problem is *with unlimited processors*, and denoted  $Pb(\infty)$ . When there is only a fixed number  $p$  of available processors, the problem is *with limited processors*, and denoted  $Pb(p)$ . In the latter case, an allocation function  $\text{alloc} : V \rightarrow \mathcal{P}$  is required, where  $\mathcal{P} = \{1, \dots, p\}$  denotes the set of available processors. This function assigns a target processor to each task. The resource constraints simply specify that no

processor can be allocated more than one task at the same time. This translates into the following conditions:

$$\text{alloc}(T) = \text{alloc}(T') \Rightarrow \begin{cases} \sigma(T) + w(T) \leq \sigma(T') \\ \text{or } \sigma(T') + w(T') \leq \sigma(T). \end{cases}$$

This condition expresses the fact that if two tasks  $T$  and  $T'$  are allocated to the same processor, then their executions cannot overlap in time.

**Definition 3** Let  $G = (V, E, w)$  be a task graph.

1. Let  $\sigma$  be a schedule for  $G$ . Assume  $\sigma$  uses at most  $p$  processors (let  $p = \infty$  if the processors are unlimited). The makespan  $MS(\sigma, p)$  of  $\sigma$  is its total execution time:

$$MS(\sigma, p) = \max_{v \in V} \{\sigma(v) + w(v)\} - \min_{v \in V} \{\sigma(v)\}.$$

2.  $Pb(p)$  is the problem of determining a schedule  $\sigma$  of minimal makespan  $MS(\sigma, p)$  assuming  $p$  processors (let  $p = \infty$  if the processors are unlimited). Let  $MS_{opt}(p)$  be the value of the makespan of an optimal schedule with  $p$  processors:

$$MS_{opt}(p) = \min_{\sigma} MS(\sigma, p).$$

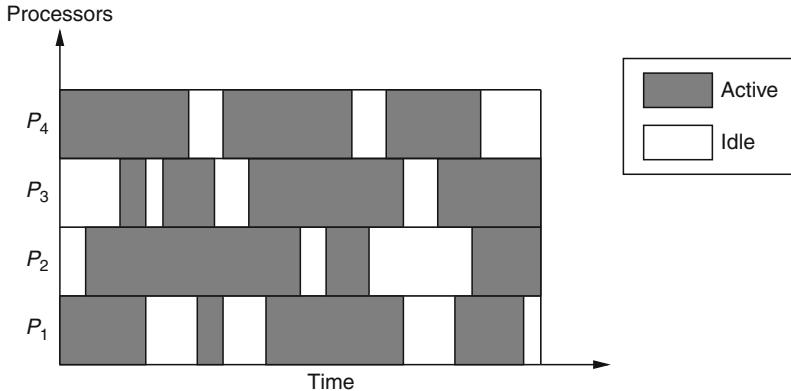
If the first task is scheduled at time 0, which is a common assumption, the expression of the makespan can be reduced to  $MS(\sigma, p) = \max_{v \in V} \{\sigma(v) + w(v)\}$ . Weights extend to paths in  $G$  as usual; if  $\Phi = (T_1, T_2, \dots, T_n)$  denotes a path in  $G$ , then  $w(\Phi) = \sum_{i=1}^n w(T_i)$ . Because schedules respect dependences, the following easy bound on the makespan is readily obtained:

**Proposition 1** Let  $G = (V, E, w)$  be a task graph and  $\sigma$  a schedule for  $G$  with  $p$  processors. Then,  $MS(\sigma, p) \geq w(\Phi)$  for all paths  $\Phi$  in  $G$ .

The last definition introduces the notions of speedup and efficiency for schedules.

**Definition 4** Let  $G = (V, E, w)$  be a task graph and  $\sigma$  a schedule for  $G$  with  $p$  processors:

1. The speedup is the ratio  $s(\sigma, p) = \frac{\text{Seq}}{MS(\sigma, p)}$ , where  $\text{Seq} = \sum_{v \in V} w(v)$  is the sum of all task weights (Seq is the optimal execution time  $MS_{opt}(1)$  of a schedule with a single processor).
2. The efficiency is the ratio  $e(\sigma, p) = \frac{s(\sigma, p)}{p} = \frac{\text{Seq}}{p \times MS(\sigma, p)}$ .



**Task Graph Scheduling. Fig. 2** Active and idle processors during execution

**Theorem 1** Let  $G = (V, E, w)$  be a task graph. For any schedule  $\sigma$  with  $p$  processors,

$$0 \leq e(\sigma, p) \leq 1.$$

*Proof* Consider the execution of  $\sigma$  as illustrated in Fig. 2 (this is a fictitious example, not related to the triangular system example). At any time during execution, some processors are active, and some are idle. At the end, all tasks have been processed. Let  $\text{Seq}$  denote the sum of all task weights, the quantity  $\text{Seq} + \text{Idle}$  is equal to the area of the rectangle in Fig. 2, that is, the product of the number of processors by the makespan of the schedule:  $\text{Seq} + \text{Idle} = p \times \text{MS}(\sigma, p)$ . Hence,  $e(\sigma, p) = \frac{\text{Seq}}{p \times \text{MS}(\sigma, p)} \leq 1$ .  $\square$

### Solving $\text{Pb}(\infty)$

Let  $G = (V, E, w)$  be a given task graph and assume unlimited processors. Remember that a schedule  $\sigma$  for  $G$  is said to be *optimal* if its makespan  $\text{MS}(\sigma, \infty)$  is minimal, that is, if  $\text{MS}(\sigma, \infty) = \text{MS}_{\text{opt}}(\infty)$ .

**Definition 5** Let  $G = (V, E, w)$  be a task graph.

1. For  $v \in V$ ,  $\text{PRED}(v)$  denotes the set of all immediate predecessors of  $v$ , and  $\text{SUCC}(v)$  the set of all its immediate successors.
2.  $v \in V$  is an entry (top) vertex if and only if  $\text{PRED}(v) = \emptyset$ .
3.  $v \in V$  is an exit (bottom) vertex if and only if  $\text{SUCC}(v) = \emptyset$ .

4. For  $v \in V$ , the top level  $tl(v)$  is the largest weight of a path from an entry vertex to  $v$ , excluding the weight of  $v$ .
5. For  $v \in V$ , the bottom level  $bl(v)$  is the largest weight of a path from  $v$  to an output vertex, including the weight of  $v$ .

In the example of the triangular system, there is a single entry vertex,  $T_{1,1}$ , and a single exit vertex,  $T_{n,n}$ . The top level of  $T_{1,1}$  is 0, and  $tl(T_{1,2}) = tl(T_{1,1}) + w(T_{1,1}) = 1$ . The value of  $T_{2,3}$  is

$$tl(T_{2,3}) = \max\{w(T_{1,1}) + w(T_{1,2}), w(T_{2,2}) + w(T_{1,1}) + w(T_{1,3})\} = 3$$

because there are two paths from the entry vertex to  $T_{2,3}$ .

The top level of a vertex can be computed by a traversal of the DAG; the top level of an entry vertex is 0, while the top level of a non-entry vertex  $v$  is

$$tl(v) = \max\{tl(u) + w(u); u \in \text{PRED}(v)\}.$$

Similarly,  $bl(v) = \max\{bl(u); u \in \text{SUCC}(v)\} + w(v)$  (and  $bl(v) = w(v)$  for an exit vertex  $v$ ). The top level of a vertex is the earliest possible time at which it can be executed, while its bottom level represents a lower bound of the remaining execution time once starting its execution. This can be stated more formally as follows.

**Theorem 2** Let  $G = (V, E, w)$  be a task graph and define  $\sigma_{\text{free}}$  as follows:

$$\forall v \in V, \sigma_{\text{free}}(v) = tl(v).$$

Then,  $\sigma_{\text{free}}$  is an optimal schedule for  $G$ .

From Theorem 2:

$$\text{MS}_{\text{opt}}(\infty) = \text{MS}(\sigma_{\text{free}}, \infty) = \max_{v \in V} \{ \text{tl}(v) + w(v) \}.$$

Hence,  $\text{MS}_{\text{opt}}(\infty)$  is simply the maximal weight of a path in the graph. Note that  $\sigma_{\text{free}}$  is not the only optimal schedule.

**Corollary 1** Let  $G = (V, E, w)$  be a directed acyclic graph.  $\text{Pb}(\infty)$  can be solved in time  $O(|V| + |E|)$ .

Going back to the triangular system (Fig. 1), because all tasks have weight 1, the weight of a path is equal to its length plus 1. The longest path is

$$T_{1,1} \rightarrow T_{1,2} \rightarrow T_{2,2} \rightarrow \dots \rightarrow T_{n-1,n-1} \rightarrow T_{n,-1,n} \rightarrow T_{n,n},$$

whose weight is  $2n - 1$ . Not as many processors as tasks are needed to achieve execution within  $2n - 1$  time units. For example, only  $n - 1$  processors can be used. Let  $1 \leq i \leq n$ ; at time  $2i - 2$ , processor  $P_1$  starts the execution of task  $T_{i,i}$ , while at time  $2i - 1$ , the first  $n - i$  processors  $P_1, P_2, \dots, P_{n-i}$  execute tasks  $T_{i,j}$ ,  $i + 1 \leq j \leq n$ .

### NP-completeness of $\text{Pb}(p)$

**Definition 6** The decision problem  $\text{Dec}(p)$  associated with  $\text{Pb}(p)$  is as follows. Given a task graph  $G = (V, E, w)$ , a number of processors  $p \geq 1$ , and an execution bound  $K \in \mathbb{N}^*$ , does there exist a schedule  $\sigma$  for  $G$  using at most  $p$  processors, such that  $\text{MS}(\sigma, p) \leq K$ ? The restriction of  $\text{Dec}(p)$  to independent tasks (no dependence, that is, when  $E = \emptyset$ ) is denoted  $\text{Indep-tasks}(p)$ . In both problems,  $p$  is arbitrary (it is part of the problem instance). When  $p$  is fixed a priori, say  $p = 2$ , problems are denoted as  $\text{Dec}(2)$  and  $\text{Indep-tasks}(2)$ .

Well-known complexity results are summarized in the following theorem.

### Theorem 3

- $\text{Indep-tasks}(2)$  is NP-complete but can be solved by a pseudo-polynomial algorithm. Moreover,  $\forall \epsilon > 0$ ,  $\text{Indep-tasks}(2)$  admits a  $(1 + \epsilon)$ -approximation whose complexity is polynomial in  $\frac{1}{\epsilon}$ .
- $\text{Indep-tasks}(p)$  is NP-complete in the strong sense.
- $\text{Dec}(2)$  (and hence  $\text{Dec}(p)$ ) is NP-complete in the strong sense.

### List Scheduling Heuristics

Because  $\text{Pb}(p)$  is NP-complete, heuristics are used to schedule task graphs with limited processors. The most

natural idea is to use greedy strategies: At each instant, try to schedule as many tasks as possible onto available processors. Such strategies deciding *not to deliberately keep a processor idle* are called *list scheduling* algorithms. Of course, there are different possible strategies to decide which tasks are given priority in the (frequent) case where there are more free tasks than available processors. But a key result due to Graham [10] is that any list algorithm can be shown to achieve at most twice the optimal makespan.

**Definition 7** Let  $G = (V, E, w)$  be a task graph and let  $\sigma$  be a schedule for  $G$ . A task  $v \in V$  is free at time  $t$  (note  $v \in \text{FREE}(\sigma, t)$ ) if and only if its execution has not yet started ( $\sigma(v) \geq t$ ) but all its predecessors have been executed ( $\forall u \in \text{PRED}(v), \sigma(u) + w(u) \leq t$ ).

A list schedule is a schedule such that no processor is deliberately left idle; at each time  $t$ , if  $|\text{FREE}(\sigma, t)| = r \geq 1$ , and if  $q$  processors are available, then  $\min(r, q)$  free tasks start executing.

**Theorem 4** Let  $G = (V, E, w)$  be a task graph and assume there are  $p$  available processors. Let  $\sigma$  be any list schedule of  $G$ . Let  $\text{MS}_{\text{opt}}(p)$  be the makespan of an optimal schedule. Then,

$$\text{MS}(\sigma, p) \leq \left(2 - \frac{1}{p}\right) \text{MS}_{\text{opt}}(p).$$

It is important to point out that Theorem 4 holds for any list schedule, regardless of the strategy to choose among free tasks when there are more free tasks than available processors.

**Lemma 1** There exists a dependence path  $\Phi$  in  $G$  whose weight  $w(\Phi)$  satisfies

$$\text{Idle} \leq (p - 1) \times w(\Phi),$$

where  $\text{Idle}$  is the cumulated idle time of the  $p$  processors during the whole execution of the list schedule.

*Proof* Define the ancestors of a task are its predecessors, the predecessors of its predecessors, and so on. Let  $T_{i_1}$  be a task whose execution terminates at the end of the schedule:

$$\sigma(T_{i_1}) + w(T_{i_1}) = \text{MS}(\sigma, p).$$

Let  $t_1$  be the largest time smaller than  $\sigma(T_{i_1})$  and such that there exists an idle processor during the time interval  $[t_1, t_1 + 1]$  (let  $t_1 = 0$  if such a time does not exist).

Why is this processor idle? Because  $\sigma$  is a list schedule, no task is free at  $t_1$ , otherwise the idle processor would start executing a free task. Therefore, there must be a task  $T_{i_2}$  that is an ancestor of  $T_{i_1}$  and that is being executed at time  $t_1$ ; otherwise  $T_{i_1}$  would have been started at time  $t_1$  by the idle processor. Because of the definition of  $t_1$ , it is known that all processors are active between the end of the execution of  $T_{i_2}$  and the beginning of the execution of  $T_{i_1}$ .

Then, start the construction again from  $T_{i_2}$  so as to obtain a task  $T_{i_3}$  such that all processors are active between the end of  $T_{i_3}$  and the beginning of  $T_{i_2}$ . Iterating the process, one ends up with  $r$  tasks  $T_{i_r}, T_{i_{r-1}}, \dots, T_{i_1}$  that belong to a dependence path  $\Phi$  of  $G$  and such that all processors are active except perhaps during their execution. In other words, the idleness of some processors can only occur during the execution of these  $r$  tasks, during which at least one processor is active (the one that executes the task). Hence,  $\text{Idle} \leq (p-1) \times \sum_{j=1}^r w(T_{i_j}) \leq (p-1) \times w(\Phi)$ .  $\square$

*Proof* Going back to the proof of [Theorem 4](#), recall that  $p \times \text{MS}(\sigma, p) = \text{Idle} + \text{Seq}$ , where  $\text{Seq} = \sum_{v \in V} w(v)$  is the sequential time, that is, the sum of all task weights (see [Fig. 2](#)). Now take the dependence path  $\Phi$  constructed in [Lemma 1](#):  $w(\Phi) \leq \text{MS}_{\text{opt}}(p)$ , because the makespan of any schedule is greater than the weight of all dependence paths in  $G$  (simply because dependence constraints are met). Furthermore,  $\text{Seq} \leq p \times \text{MS}_{\text{opt}}(p)$  (with equality only if all  $p$  processors are active all the time). Putting this together:

$$\begin{aligned} p \times \text{MS}(\sigma, p) &= \text{Idle} + \text{Seq} \leq (p-1)w(\Phi) + \text{Seq} \\ &\leq (p-1)\text{MS}_{\text{opt}}(p) + p\text{MS}_{\text{opt}}(p) \\ &= (2p-1)\text{MS}_{\text{opt}}(p), \end{aligned}$$

which proves the theorem.  $\square$

Fundamentally, [Theorem 4](#) says that any list schedule is within 50% of the optimum. Therefore, list scheduling is guaranteed to achieve half the best possible performance, regardless of the strategy to choose among free tasks.

**Proposition 2** Let  $\text{MS}_{\text{list}}(p)$  be the shortest possible makespan produced by a list scheduling algorithm.

The bound

$$\text{MS}_{\text{list}}(p) \leq \frac{2p-1}{p} \text{MS}_{\text{opt}}(p)$$

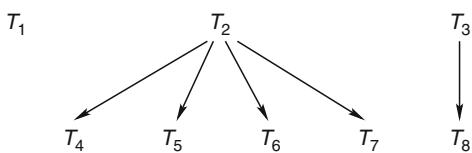
is tight.

Note that implementing a list scheduling algorithm is not difficult, but it is somewhat lengthy to describe in full detail; see Casanova et al. [3].

### Critical Path Scheduling

A widely used list scheduling technique is *critical path scheduling*. The selection criterion for free tasks is based on the value of their bottom level. Intuitively, the larger the bottom level, the more “urgent” the task. The *critical path* of a task is defined as its bottom level and is used to assign priority levels to tasks. Critical path scheduling is list scheduling where the priority level of a task is given by the value of its critical path. Ties are broken arbitrarily.

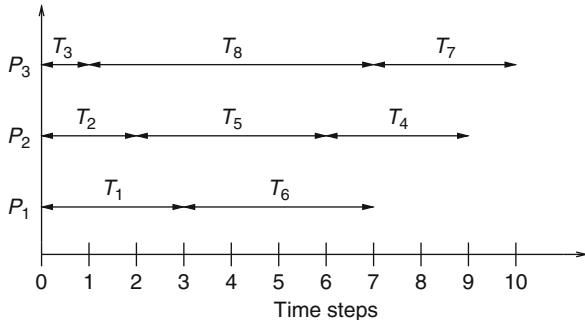
Consider the task graph shown in [Fig. 3](#). There are eight tasks, whose weights and critical paths are listed in [Table 1](#). Assume there are  $p = 3$  available processors and let  $\mathcal{Q}$  be the priority queue of free tasks. At  $t = 0$ ,  $\mathcal{Q}$  is initialized as  $\mathcal{Q} = (T_3, T_2, T_1)$ . These three tasks are executed. At  $t = 1$ ,  $T_8$  is added to the queue:  $\mathcal{Q} = (T_8)$ . There is one processor available, which starts the execution of  $T_8$ . At  $t = 2$ , the four successors of  $T_2$  are added to the queue:  $\mathcal{Q} = (T_5, T_6, T_4, T_7)$ . Note that ties have been broken arbitrarily (using task indices in this case). The available processor picks the first task  $T_5$  in  $\mathcal{Q}$ . Following this scheme, the execution goes on up to  $t = 10$ , as summarized in [Fig. 4](#).



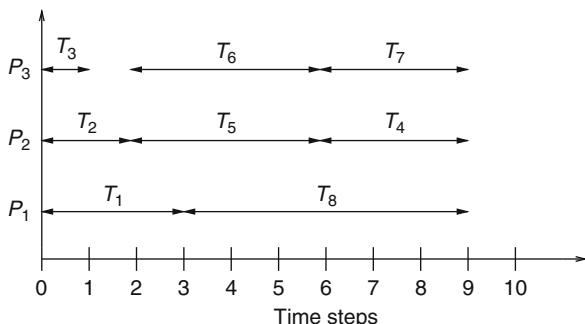
**Task Graph Scheduling. Fig. 3** A small example

**Task Graph Scheduling. Table 1** Weights and critical paths for the task graph in [Fig. 3](#)

| Tasks          | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $T_6$ | $T_7$ | $T_8$ |
|----------------|-------|-------|-------|-------|-------|-------|-------|-------|
| Weights        | 3     | 2     | 1     | 3     | 4     | 4     | 3     | 6     |
| Critical paths | 3     | 6     | 7     | 3     | 4     | 4     | 3     | 6     |



**Task Graph Scheduling.** Fig. 4 Critical path schedule for the example in Fig. 3



**Task Graph Scheduling.** Fig. 5 Optimal schedule for the example in Fig. 3

Note that it is possible to schedule the graph in only 9 time units, as shown in Fig. 5. The trick is to leave a processor idle at time  $t = 1$  deliberately; although it has the highest critical path,  $T_8$  can be delayed by two time units.  $T_5$  and  $T_6$  are given preference to achieve a better load balance between processors. The schedule shown in Fig. 5 is optimal, because  $\text{Seq} = 26$ , so that three processors require at least  $\lceil \frac{26}{3} \rceil = 9$  time units. This small example illustrates the difficulty of scheduling with a limited number of processors.

## Taking Communication Costs into Account

### The Macro-Dataflow Model

Thirty years ago, communication costs have been introduced in the scheduling literature. Because the performance of network communication is difficult to model in a way that is both precise and conducive to understanding the performance of algorithms, the vast majority of results hold for a very simple model, which is as follows.

The target platform consists of  $p$  identical processors that are part of a fully connected clique. All interconnection links have same bandwidth. If a task  $T$  communicates data to a successor task  $T'$ , the cost is modeled as

$$\text{cost}(T, T') = \begin{cases} 0 & \text{if } \text{alloc}(T) = \text{alloc}(T') \\ c(T, T') & \text{otherwise,} \end{cases}$$

where  $\text{alloc}(T)$  denotes the processor that executes task  $T$ , and  $c(T, T')$  is defined by the application specification. The time for communication between two tasks running on the same processor is negligible. This so-called *macro-dataflow* model makes two main assumptions: (i) communication can occur as soon as data are available and (ii) there is no contention for network links. Assumption (i) is reasonable as communication can overlap with (independent) computations in most modern computers. Assumption (ii) is much more questionable. Indeed, there is no physical device capable of sending, say, 1,000 messages to 1,000 distinct processors, at the same speed as if there were a single message. In the worst case, it would take 1,000 times longer (serializing all messages). In the best case, the output bandwidth of the network card of the sender would be a limiting factor. In other words, assumption (ii) amounts to assuming infinite network resources. Nevertheless, this assumption is omnipresent in the traditional scheduling literature.

**Definition 8** A communication task graph (or *commTG*) is a direct acyclic graph  $G = (V, E, w, c)$ , where vertices represent tasks and edges represent precedence constraints. The computation weight function is  $w : V \rightarrow \mathbb{N}^*$  and the communication cost function is  $c : E \rightarrow \mathbb{N}^*$ . A schedule  $\sigma$  must preserve dependences, which is written as

$$\forall e = (T, T') \in E, \left\{ \begin{array}{l} \sigma(T) + w(T) \leq \sigma(T') \\ \quad \quad \quad \text{if } \text{alloc}(T) = \text{alloc}(T') \\ \sigma(T) + w(T) + c(T, T') \leq \sigma(T') \\ \quad \quad \quad \text{otherwise.} \end{array} \right.$$

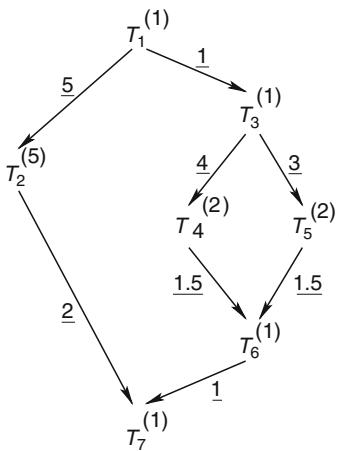
The expression of resource constraints is the same as in the no-communication case.

## Complexity and List Heuristics with Communications

Including communication costs in the model makes everything difficult, including solving  $Pb(\infty)$ . The intuitive reason is that a trade-off must be found between allocating tasks to either many processors (hence balancing the load but communicating intensively) or few processors (leading to less communication but less parallelism as well). Here is a small example, borrowed from [9].

Consider the commTG in Fig. 6. Task weights are indicated close to the tasks within parentheses, and communication costs are shown along the edges, underlined. For the sake of this example, two non-integer communication costs are used:  $c(T_4, T_6) = c(T_5, T_6) = 1.5$ . Of course, every weight  $w$  and cost  $c$  could be scaled to have only integer values. Observe the following:

- On the one hand, if all tasks are assigned to the same processor, the makespan will be equal to the sum of all task weights, that is, 13.
- On the other hand, with unlimited processors (no more than seven processors are needed because there are seven tasks), each task can be assigned to a different processor. Then, the makespan of the ASAP schedule is equal to 14. To see this, it is important to point out that once the allocation of tasks to processors is given, the makespan is computed easily: For each edge  $e : T \rightarrow T'$ , add a virtual node of weight  $c(T, T')$  if the edge links two different processors ( $\text{alloc}(T) \neq \text{alloc}(T')$ ), and do nothing



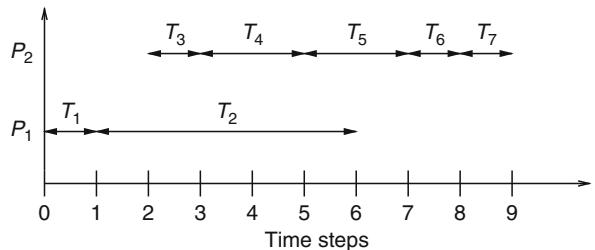
Task Graph Scheduling. Fig. 6 An example commTG

otherwise. Then, consider the new graph as a DAG (without communications) and traverse it to compute the length of the longest path. Here, because all tasks are allocated to different processors, a virtual node is added on each edge. The longest path is  $T_1 \rightarrow T_2 \rightarrow T_7$ , whose length is  $w(T_1) + c(T_1, T_2) + w(T_2) + c(T_2, T_7) + w(T_7) = 14$ .

There is a difficult trade-off between executing tasks in parallel (hence with several distinct processors) and minimizing communication costs. In the example, it turns out that the best solution is to use two processors, according to the schedule in Fig. 7, whose makespan is equal to 9. Using more processors does not always lead to a shorter execution time. Note that dependence constraints are satisfied in Fig. 7. For example,  $T_2$  can start at time 1 on processor  $P_1$  because this processor executes  $T_1$ , hence there is no need to pay the communication cost  $c(T_1, T_2)$ . By contrast,  $T_3$  is executed on processor  $P_2$ , hence it cannot be started before time 2 even though  $P_2$  is idle:  $\sigma(T_1) + w(T_1) + c(T_1, T_3) = 0 + 1 + 1 = 2$ .

With unlimited processors, the optimization problem becomes difficult:  $Pb(\infty)$  is NP-complete in the strong sense. Even the problem in which all task weights and communication costs have the same (unit) value, the so-called UET-UCT problem (unit execution time-unit communication time), is NP-hard [13].

With limited processors, list heuristics can be extended to take communication costs into account, but Graham's bound does not hold any longer. For instance, the *Modified Critical Path* (MCP) algorithm proceeds as follows. First, bottom levels are computed using a pessimistic evaluation of the longest path, accounting for each potential communication (this corresponds to the allocation where there is a different processor per task).



Task Graph Scheduling. Fig. 7 An optimal schedule for the example

These bottom levels are used to determine the priority of free tasks. Then each free task is assigned to the processor that allows its earliest execution, given previous task allocation decisions. It is important to explain further what “previous task allocation decisions” means. Free tasks from the queue are processed one after the other. At any moment, it is known which processors are available and which ones are busy. Moreover, for the busy processors, it is known when they will finish computing their currently allocated tasks. Hence, it is always possible to select the processor that can begin executing the task soonest. It may well be the case that a currently busy processor is selected.

### Extension to Heterogeneous Platforms

This section explains how to extend list scheduling techniques to heterogeneous platforms, that is, to platforms that consist of processors with different speeds and interconnection links with different bandwidths. Key differences with the homogeneous case are outlined.

Given a commTG with  $n$  tasks  $T_1, \dots, T_n$ , the goal is to schedule it on a platform with  $p$  heterogeneous processors  $P_1, \dots, P_p$ . There are now many parameters to instantiate:

**Computation costs :** The execution cost of  $T_i$  on  $P_q$  is modeled as  $w_{iq}$ . Therefore, an  $n \times p$  matrix of values is needed to specify all computation costs. This matrix comes directly for the specific scheduling problem at hand. However, when attempting to evaluate competing scheduling heuristics over a large number of synthetic scenarios, one must generate this matrix. One can distinguish two approaches. In the first approach one generates a *consistent* (or *uniform*) matrix with  $w_{iq} = w_i \times \gamma_q$ , where  $w_i$  represents the number of operations required by  $T_i$  and  $\gamma_q$  is the inverse of the speed of  $P_q$  (in operations per second). With this definition the relative speed of the processors does not depend on the particular task they execute. If instead some processors are faster for some tasks than some other processors, but slower for other tasks, one speaks of an *inconsistent* (or *nonuniform*) matrix. This corresponds to the case in which some processors are specialized for some tasks (e.g., specialized hardware or software).

**Communication costs :** Just as processors have different speeds, communication links may have different

bandwidths. However, while the speed of a processor may depend upon the nature of the computation it performs, the bandwidth of a link does not depend on the nature of the bytes it transmits. It is therefore natural to assume *consistent* (or *uniform*) links. If there is a dependence  $e_{ij} : T_i \rightarrow T_j$ , if  $T_i$  is executed on  $P_q$  and  $T_j$  executed on  $P_r$ , then the communication time is modeled as

$$\text{comm}(i, j, q, r) = \text{data}(i, j) \times v_{qr},$$

where  $\text{data}(i, j)$  is the data volume associated to  $e_{ij}$  and  $v_{qr}$  is the communication time for a unit-size message from  $P_q$  to  $P_r$  (i.e., the inverse of the bandwidth). Like in the homogeneous case, let  $v_{qr} = 0$  if  $q = r$ , that is, if both tasks are assigned the same processor. If one wishes to generate synthetic scenarios to evaluate competing scheduling heuristics, one then must generate two matrices: one of size  $n \times n$  for data and one of size  $p \times p$  for  $v_{qr}$ .

The main list scheduling principle is unchanged. As before, the priority of each task needs to be computed, so as to decide which one to execute first when there are more free tasks than available processors. The most natural idea is to compute averages of computation and communication times, and use these to compute priority levels exactly as in the homogeneous case:

- $\overline{w}_i = \frac{\sum_{q=1}^p w_{iq}}{p}$ , the *average execution time* of  $T_i$ .
- $\overline{\text{comm}}_{ij} = \text{data}(i, j) \times \frac{\sum_{1 \leq q, r \leq p, q \neq r} v_{qr}}{p(p-1)}$ , the *average communication cost* for edge  $e_{ij} : T_i \rightarrow T_j$ .

The last (but important) modification concerns the way in which tasks are assigned to processors: Instead of assigning the current task to the processor that will *start* its execution first (given all already taken decisions), one should assign it to the processor that will *complete* its execution first (given all already taken decisions). Both choices are equivalent with homogeneous processors, but intuitively the latter is likely to be more efficient in the heterogeneous case. Altogether, this leads to the list heuristic called HEFT, for *Heterogeneous Earliest Finish Time* [19].

T

### Workflow Scheduling

This section discusses *workflow scheduling*, that is, the problem of scheduling a (large) collection of identical task graphs rather than a single one. The main idea is to pipeline the execution of successive instances. Think

of a sequence of video images that must be processed in a pipelined fashion: Each image enters the platform and follows the same processing chain, and a new image can enter the system while previous ones are still being executed. This section is intended to give a flavor of the optimization problems to be solved in such a context. It restricts to simpler problem instances.

Consider “chains,” that is, applications structured as a sequence of stages. Each stage corresponds to a different computational task. The application must process a large number of data sets, each of which must go through all stages. Each stage has its own communication and computation requirements: It reads an input from the previous stage, processes the data, and outputs a result to the next stage. Initial data are input to the first stage and final results are obtained as the output from the last stage. The pipeline operates in synchronous mode: After some initialization delay, a new task is completed every period. The period is defined as the longest “cycle-time” to operate a stage, and it is the inverse of the throughput that can be achieved.

For simplicity, it is assumed that each stage is assigned to a single processor, that is in charge of processing all instances (all data sets) for that stage. Each pipeline stage can be viewed a sequential task that may write some global data structure, to disk or to memory, for each processed data set. In this case, tasks must always be processed in a sequential order within a stage. Moreover, due to possible local updates, each stage must be mapped onto a single processor. For a given stage, one cannot process half of the tasks on one processor and the remaining half on another without maintaining global information, which might be costly and difficult to implement. In other words, a processor that is assigned a stage will execute the operations required by this stage (input, computation, and output) for all the tasks fed into the pipeline.

Of course, other assumptions are possible: some stages could be replicated, or even data-parallelized. The reader is referred the bibliographical notes at the end of the chapter for such extensions.

## Objective Functions

An important metric for parallel applications that consists of many individual computations is the *throughput*. The throughput measures the aggregate rate of data processing; it is the rate at which data sets can enter

the system. Equivalently, the inverse of the throughput, defined as the *period*, is the time interval required between the beginning of the execution of two consecutive data sets. The period minimization problem can be stated informally as follows: Which stage to assign to which processor so that the largest period of a processor is kept minimal?

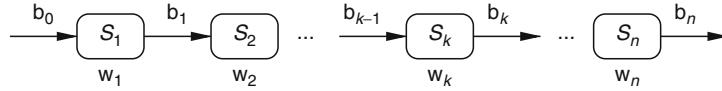
Another important metric is derived from makespan minimization, but it must be adapted. With a large number of data sets, the total execution time is less relevant, but the execution time for each data set remains important, in particular for real-time applications. One talks of *latency* rather than of makespan, in order to avoid confusion. The latency is the time elapsed between the beginning and the end of the execution of a given data set, hence it measures the response time of the system to process the data set entirely.

Minimizing the latency is antagonistic to maximizing the throughput. In fact, assigning all application stages to the fastest processor (thus working in a fully sequential way) would suppress all communications and accelerate computations, thereby minimizing the latency, but achieving a very bad throughput. Conversely, mapping each stage to a different processor is likely to decrease the period, hence increase the throughput (work in a fully pipelined manner), but the resulting latency will be high, because all interstage communications must be accounted for in this latter mapping. Trade-offs will have to be found between these criteria.

How to deal with several objective functions? In traditional approaches, one would form a linear combination of the different objectives and treat the result as the new objective to optimize for. But it is not natural for the user to maximize a quantity like  $0.7T + 0.3L$ , where  $T$  is the throughput and  $L$  the latency. Instead, one is more likely to fix a throughput  $T$ , and to search for the best latency that can be achieved while enforcing  $T$ ? One single criterion is optimized, under the condition that a threshold is enforced for the other one.

## Period and Latency

Consider a pipeline with  $n$  stages  $S_k$ ,  $1 \leq k \leq n$ , as illustrated in Fig. 8. Tasks are fed into the pipeline and processed from stage to stage, until they exit the pipeline after the last stage. The  $k$ -th stage  $S_k$  receives an input from the previous stage, of size  $b_{k-1}$ , performs a number of  $w_k$  operations, and outputs data of size  $b_k$  to the



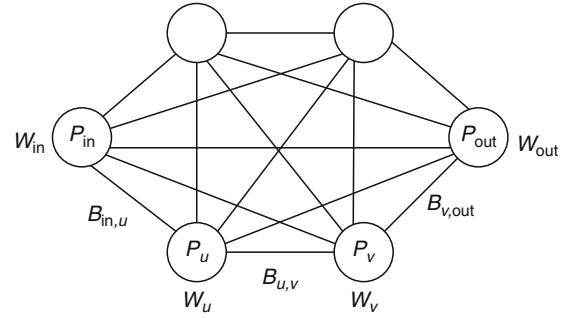
**Task Graph Scheduling.** Fig. 8 The application pipeline

next stage. The first stage  $S_1$  receives an initial input of size  $b_0$ , while the last stage  $S_n$  returns a final result of size  $b_n$ .

The target platform is a clique with  $p$  processors  $P_u$ ,  $1 \leq u \leq p$ , that are fully interconnected (see Fig. 9). There is a bidirectional link  $\text{link}_{u,v} : P_u \leftrightarrow P_v$  with bandwidth  $B_{u,v}$  between each processor  $P_u$  and  $P_v$ . The literature often enforces more realistic communication models for workflow scheduling than for Task Graph Scheduling. For the sake of simplicity, a very strict model is enforced here: A given processor can be involved in a single communication at any time unit, either a send or a receive. Note that independent communications between distinct processor pairs can take place simultaneously. Finally, there is no overlap between communications and computations, so that all the operations of a given processor are fully sequentialized. The speed of processor  $P_u$  is denoted as  $W_u$ , and it takes  $X/W_u$  time units for  $P_u$  to execute  $X$  operations. It takes  $X/B_{u,v}$  time units to send (respectively, receive) a message of size  $X$  to (respectively, from)  $P_v$ .

The mapping problem consists in assigning application stages to processors. For *one-to-one mappings*, it is required that each stage  $S_k$  of the application pipeline be mapped onto a distinct processor  $P_{\text{alloc}(k)}$  (which is possible only if  $n \leq p$ ). The function  $\text{alloc}$  associates a processor index to each stage index. For convenience, two fictitious stages  $S_0$  and  $S_{n+1}$  are created, assigning  $S_0$  to  $P_{\text{in}}$  and  $S_{n+1}$  to  $P_{\text{out}}$ .

What is the period of  $P_{\text{alloc}(k)}$ , that is, the minimum delay between the processing of two consecutive tasks? To answer this question, one needs to know to which processors the previous and next stages are assigned. Let  $t = \text{alloc}(k-1)$ ,  $u = \text{alloc}(k)$ , and  $v = \text{alloc}(k+1)$ .  $P_u$  needs  $b_{k-1}/B_{t,u}$  time units to receive the input data from  $P_t$ ,  $w_k/W_u$  time units to process it, and  $b_k/B_{u,v}$  time units to send the result to  $P_v$ , hence a cycle-time of  $b_{k-1}/B_{t,u} + w_k/W_u + b_k/B_{u,v}$  time units for  $P_u$ . These three steps are serialized (see Fig. 10 for an illustration). The *period* achieved with the mapping is the maximum of the cycle-times of the processors, which corresponds to the rate at which the pipeline can be activated.



**Task Graph Scheduling.** Fig. 9 The target platform

In this simple instance, the optimization problem can be stated as follows: Determine a one-to-one allocation function  $\text{alloc} : [1, n] \rightarrow [1, p]$  (augmented with  $\text{alloc}(0) = \text{in}$  and  $\text{alloc}(n+1) = \text{out}$ ) such that

$$T_{\text{period}} = \max_{1 \leq k \leq n} \left\{ \frac{b_{k-1}}{B_{\text{alloc}(k-1), \text{alloc}(k)}} + \frac{w_k}{W_{\text{alloc}(k)}} + \frac{b_k}{B_{\text{alloc}(k), \text{alloc}(k+1)}} \right\}$$

is minimized.

Natural extensions are *interval mappings*, in which each participating processor is assigned an interval of consecutive stages. Note that when  $p < n$  interval mappings are mandatory. Intuitively, assigning several consecutive tasks to the same processor will increase its computational load, but will also decrease communication. The best interval mapping may turn out to be a one-to-one mapping, or instead may utilize only a very small number of fast computing processors interconnected by high-speed links. The optimization problem associated to interval mappings is formally expressed as follows. The intervals achieve a partition of the original set of stages  $S_1$  to  $S_n$ . One searches for a partition of  $[1, \dots, n]$  into  $m$  intervals  $I_j = [d_j, e_j]$  such that  $d_j \leq e_j$  for  $1 \leq j \leq m$ ,  $d_1 = 1$ ,  $d_{j+1} = e_j + 1$  for  $1 \leq j \leq m-1$ , and  $e_m = n$ . Recall that the function  $\text{alloc} : [1, n] \rightarrow [1, p]$  associates a processor index to each stage index. In a one-to-one mapping, this function was a one-to-one

| Time unit | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18  | ... |
|-----------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|-----|-----|
| $P_1$     | ★ | ■ | ■ | △ | △ | ★ | ■ | ■ | △ | △  | ★  | ■  | ■  | △  | △  | ★  | ■  | ■   | ... |
| $P_2$     |   |   | ★ | ★ | ■ | △ | △ | ★ | ★ | ■  | △  | △  | ★  | ★  | ■  | △  | △  | ... |     |
| $P_3$     |   |   |   | ★ | ★ | ■ | △ | ★ | ★ | ■  | △  | ★  | ★  | ■  | △  | ★  | ★  | ... |     |

**Task Graph Scheduling.** Fig. 10 An example of one-to-one mapping with three stages and processors. Each processor periodically receives input data from its predecessor (★), performs some computation (■), and outputs data to its successor (△). Note that these operations are shifted in time from one processor to another. The cycle-time of  $P_1$  and  $P_2$  is 5 while that of  $P_3$  is 4, hence  $T_{\text{period}} = 5$

assignment. In an interval mapping, for  $1 \leq j \leq m$ , the whole interval  $I_j$  is mapped onto the same processor  $P_{\text{alloc}(d_j)}$ , that is, for  $d_j \leq i \leq e_j$ ,  $\text{alloc}(i) = \text{alloc}(d_j)$ . Also, two intervals cannot be mapped to the same processor, that is, for  $1 \leq j, j' \leq m$ ,  $j \neq j'$ ,  $\text{alloc}(d_j) \neq \text{alloc}(d_{j'})$ . The period is expressed as

$$T_{\text{period}} = \max_{1 \leq j \leq m} \left\{ \frac{b_{d_j-1}}{B_{\text{alloc}(d_j-1), \text{alloc}(d_j)}} + \frac{\sum_{i=d_j}^{e_j} w_i}{W_{\text{alloc}(d_j)}} \right. \\ \left. + \frac{b_{e_j}}{B_{\text{alloc}(d_j), \text{alloc}(e_j+1)}} \right\}$$

Note that  $\text{alloc}(d_j - 1) = \text{alloc}(e_{j-1}) = \text{alloc}(d_{j-1})$  for  $j > 1$  and  $d_1 - 1 = 0$ . Also,  $e_j + 1 = d_{j+1}$  for  $j < m$ , and  $e_m + 1 = n + 1$ . It is still assumed that  $\text{alloc}(0) = \text{in}$  and  $\text{alloc}(n + 1) = \text{out}$ . The optimization problem is then to determine the mapping that minimizes  $T_{\text{period}}$ , over all possible partitions into intervals, and over all mappings of these intervals to the processors.

The latency of an interval mapping is computed as follows. Each data set traverses all stages, but only communications between two stages mapped on the same processors take zero time units. Overall, the latency is expressed as

$$T_{\text{latency}} = \sum_{1 \leq j \leq m} \left\{ \frac{b_{d_j-1}}{B_{\text{alloc}(d_j-1), \text{alloc}(d_j)}} + \frac{\sum_{i=d_j}^{e_j} w_i}{W_{\text{alloc}(d_j)}} \right\} \\ + \frac{b_n}{B_{\text{alloc}(n), \text{alloc}(n+1)}}$$

The latency for a one-to-one mapping obeys the same formula (with the restriction that each interval has length 1). Just as for the period, there are two minimization problems for the latency, with one-to-one and interval mappings.

It goes beyond the scope of this entry to assess the complexity of these period/latency optimization problems, and of their bi-criteria counterparts. The aim was to provide the reader with a quick primer on workflow scheduling, an activity that borrows several concepts from Task Graph Scheduling, while using more realistic platform models, and different objective functions.

## Related Entries

- Loop Pipelining
- Modulo Scheduling and Loop Pipelining
- Scheduling Algorithms

## Recommended Reading

Without communication costs, pioneering work includes the book by Coffman [5]. The book by El-Rewini et al. [7] and the IEEE compilation of papers [15] provide additional material. On the theoretical side, Appendix A5 of Garey and Johnson [8] provides a list of NP-complete scheduling problems. Also, the book by Brucker [2] offers a comprehensive overview of many complexity results.

The literature with communication costs is more recent. See the survey paper by Chrétienne and Picouleau [4]. See also the book by Darte et al. [6], where many heuristics are surveyed. The book by Sinnem [16] provides a thorough discussion on communication models. In particular, it describes several extensions for modeling and accounting for communication contention.

Workflow scheduling is quite a hot topic with the advent of large-scale computing platforms. A few representative papers are [1, 11, 17, 18].

Modern scheduling encompasses a wide spectrum of techniques: divisible load scheduling, cyclic scheduling, steady-state scheduling, online scheduling, job scheduling, and so on. A comprehensive survey is available in the book [14]. See also the handbook [12].

Most of the material presented in this entry is excerpted from the book by Casanova et al. [3].

## Bibliography

1. Benoit A, Robert Y (2008) Mapping pipeline skeletons onto heterogeneous platforms. *J Parallel Distr Comput* 68(6): 790–808
2. Brucker P (2004) Scheduling algorithms. Springer, New York
3. Casanova H, Legrand A, Robert Y (2008) Parallel algorithms. Chapman & Hall/CRC Press, Beaumont, TX
4. Chrétienne P, Picouleau C (1995) Scheduling with communication delays: a survey. In: Chrétienne P, Coffman EG Jr, Lenstra JK, Liu Z (eds) *Scheduling theory and its applications*. Wiley, Hoboken, NJ, pp 65–89
5. Coffman EG (1976) Computer and job-shop scheduling theory. Wiley, Hoboken, NJ
6. Darte A, Robert Y, Vivien F (2000) Scheduling and automatic parallelization. Birkhäuser, Boston
7. El-Rewini H, Lewis TG, Ali HH (1994) Task scheduling in parallel and distributed systems. Prentice Hall, Englewood Cliffs
8. Garey MR, Johnson DS (1991) Computers and intractability, a guide to the theory of NP-completeness. WH Freeman and Company, New York
9. Gerasoulis A, Yang T (1992) A comparison of clustering heuristics for scheduling DAGs on multiprocessors. *J Parallel Distr Comput* 16(4):276–291
10. Graham RL (1996) Bounds for certain multiprocessor anomalies. *Bell Syst Tech J* 45:1563–1581
11. Harry SL, Ozguner F (1999) Precedence-constrained task allocation onto point-to-point networks for pipelined execution. *IEEE Trans Parallel Distr Syst* 10(8):838–851
12. Leung JY-T (ed) (2004) *Handbook of scheduling: algorithms, models, and performance analysis*. Chapman and Hall/CRC Press, Boca Raton
13. Picouleau C (1995) Task scheduling with interprocessor communication delays. *Discrete App Math* 60(1–3):331–342
14. Robert Y, Vivien F (eds) (2009) *Introduction to scheduling*. Chapman and Hall/CRC Press, Boca Raton
15. Shirazi BA, Hurson AR, Kavi KM (1995) Scheduling and load balancing in parallel and distributed systems. *IEEE Computer Science Press*, San Diego
16. Sinnen O (2007) Task scheduling for parallel systems. Wiley, Hoboken
17. Spencer M, Ferreira R, Beynon M, Kurc T, Catalyürek U, Sussman A, Saltz J (2002) Executing multiple pipelined data analysis operations in the grid. *Proceedings of the ACM/IEEE supercomputing conference*. ACM Press, Los Alamitos
18. Subhlok J, Vondran G (1995) Optimal mapping of sequences of data parallel tasks. *Proceedings of the 5th ACM SIGPLAN symposium on principles and practice of parallel programming*. ACM Press, San Diego, pp 134–143
19. Topcuoglu H, Hariri S, Wu MY (2002) Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans Parallel Distr Syst* 13(3):260–274

## Task Mapping, Topology Aware

### ► Topology Aware Task Mapping

## Tasks

### ► Processes, Tasks, and Threads

## TAU

SAMEER SHENDE, ALLEN D. MALONY, ALAN MORRIS,  
WYATT SPEAR, SCOTT BIERSDORFF  
University of Oregon, Eugene, OR, USA

## Synonyms

[TAU performance system®](#); [Tuning and analysis utilities](#)

## Definition

The TAU Performance System® is an integrated suite of tools for instrumentation, measurement, and analysis of parallel programs with particular focus on large-scale, high-performance computing (HPC) platforms. TAU's objectives are to provide a flexible and interoperable framework for performance tool research and development, and robust, portable, and scalable set of technologies for performance evaluation on high-end computer systems.

## Discussion

### Introduction

Scalable parallel systems have always evolved together with the tools used to observe, understand, and optimize their performance. Next-generation parallel computing environments are guided to a significant degree

by what is known about application performance on current machines and how performance factors might be influenced by technological innovations. State-of-the-art performance tools play an important role in helping to understand application performance, diagnose performance problems, and guide tuning decisions on modern parallel platforms. However, performance tool technology must also respond to the growing complexity of next-generation parallel systems in order to help deliver the promises of high-end computing (HEC).

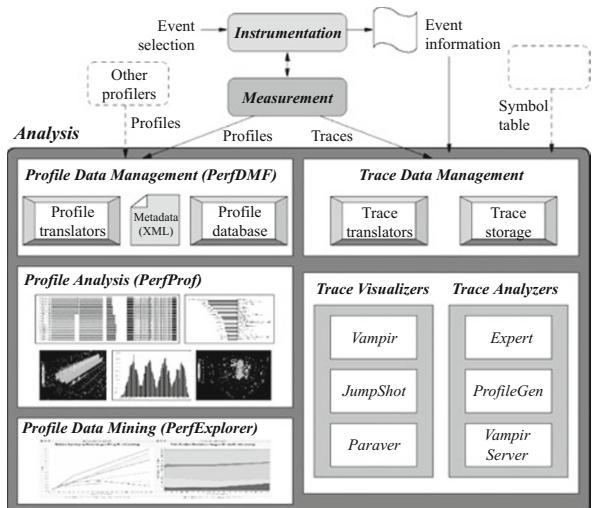
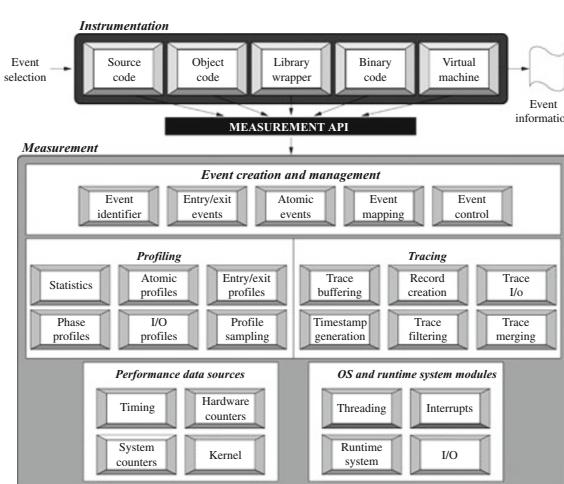
The TAU project began in the early 1990s with the goal of creating a performance instrumentation, measurement, and analysis framework that could produce robust, portable, and scalable performance tools for use in all parallel programs and systems over several technology generations. Today, the TAU Performance System® is a ubiquitous performance tools suite for shared-memory and message-passing parallel applications written in multiple programming languages (e.g., C, C++, Fortran, OpenMP, Java, Python, UPC, Chapel) that can scale to the largest parallel machines available.

## TAU Design

TAU is of a class of performance systems based on the approach of *direct performance observation*, wherein execution *actions* of performance interest are exposed as *events* to the performance system through direct insertion of instrumentation in the application, library, or

system code, at locations where the actions arise. In general, the actions reflect an occurrence of some execution state, most commonly as a result of a code location being reached (e.g., entry in a subroutine). However, it could also include a change in data. The key point is that the observation mechanism is direct. Generated events are made visible to the performance system in this way and contain implicit meta information as to their associated action. Thus, for any *performance experiment* using direct observation, the performance events of interest must be decided and necessary instrumentation done for their generation. Performance measurements are made of the events during execution and saved for analysis. Knowledge of the events is used to process the performance data and interpret the analysis results.

The TAU framework architecture, shown in Fig. 1, separates the functional concerns of a direct performance observation approach into three primary layers – *instrumentation*, *measurement*, and *analysis*. Each layer uses multiple modules which can be configured in a flexible manner under user control. This design makes it possible for TAU to target alternative models of parallel computation, from shared-memory multi-threading to distributed memory message passing to mixed-mode parallelism [17]. TAU defines an abstract computation model for parallel systems that captures general architecture and software execution features and can be mapped to existing complex system types [16].



TAU. Fig. 1 TAU architecture: instrumentation and measurement (left), analysis (right)

TAU's design has proven to be robust, sound, and highly adaptable to generations of parallel systems. The framework architecture has allowed new components to be added that have extended the capabilities of the TAU toolkit. This is especially true in areas concerning kernel-level performance integration [11, 14], performance monitoring [10, 12, 13], performance data mining [6], and GPU performance measurement [8].

## TAU Instrumentation

The role of the instrumentation layer in direct performance observation is to insert code (a.k.a. *probes*) to make performance events visible to the measurement layer. Performance events can be defined and instrumentation inserted in a program at several levels of the program transformation process. In fact, it is important to realize that a complete performance view may require contribution of event information across code levels [15]. For these reasons, TAU supports several instrumentation mechanisms based on the code type and transformation level: source (manual, preprocessor, library interposition), binary/dynamic, interpreter, and virtual machine. There are multiple factors that affect the choice of what level to instrument, including accessibility, flexibility, portability, concern for intrusion, and functionality. It is not a question of what level is "correct" because there are trade-offs for each and different events are visible at different levels. TAU is distinguished by its broad support for different instrumentation methods and their use together.

TAU supports two general classes of events for instrumentation using any method: *atomic* events and *interval* events. An atomic event denotes a single action. Instrumentation is inserted at a point in the program code to expose an atomic action, and the measurement system obtains performance data associated with the action where and when it occurs. An interval event is a pair of events: *begin* and *end*. Instrumentation is inserted at two points in the code, and the measurement system uses data obtained from each event to determine performance for the interval between them (e.g., the time spent in a subroutine from entry (beginning of the interval) to exit (end of the interval)). In addition to the two general events classes, TAU allows events to be selectively enabled / disabled for any instrumentation method.

## TAU Measurement

The TAU performance measurement system is a highly robust, scalable infrastructure portable to all HPC platforms. As shown in Fig. 1, TAU supports the two dominant methods of measurement for direct performance observation – parallel *profiling* and *tracing* – with rich access to performance data through portable timing facilities, integration with hardware performance counters, and user-level information. The choice of measurement method and performance data is made independently of the instrumentation decisions. This allows multiple performance experiments to be conducted to gain different performance views for the same set of events. TAU also provides unique support for novel performance mapping [15], runtime monitoring [10, 12, 13], and kernel-level measurements [11, 14].

TAU's measurement system has two core capabilities. First, the *event management* handles the registration and encoding of events as they are created. New events are represented in an *event table* by instantiating a new *event record*, recording the *event name*, and linking in storage allocated for the event performance data. The event table is used for all atomic and interval events regardless of their complexity. Event type and context information are encoded in the event names. The TAU event-management system hashes and maps these names to determine if an event has already occurred or needs to be created. Events are managed for every thread of execution in the application. Second, a runtime representation, called the *event callstack*, captures the nesting relationship of interval performance events. It is a powerful runtime measurement abstraction for managing the TAU performance state for use in both profiling and tracing. In particular, the event callstack is key for managing execution context, allowing TAU to associate this context to the events being measured.

Parallel profiling in TAU characterizes the behavior of every application thread in terms of its aggregate performance metrics. For interval events, TAU computes *exclusive* and *inclusive* metrics for each event. The TAU profiling system supports several profiling variants. The standard type of profiling is called *flat profiling*, which shows the exclusive and inclusive performance of each event but provides no other performance information about events occurring when an interval is active (i.e., nested events). In contrast, TAU's *event path profiling* can capture performance data with respect to event nesting

relationships. It is also interesting to observe performance data relative to an execution *state*. The structural, logical, and numerical aspects of a computation can be thought of as representing different execution *phases*. TAU supports an interface to create (*phase events*) and to mark their entry and exit. Internally in the TAU measurement system, when a phase,  $P$ , is entered, all subsequent performance will be measured with respect to  $P$  until it exits. When phase profiles are recorded, a separate parallel profile is generated for each phase.

TAU implements robust, portable, and scalable parallel tracing support to log events in time-ordered tuples containing a time stamp, a location (e.g., node, thread), an identifier that specifies the type of event, event-specific information, and other performance-related data (e.g., hardware counters). All performance events are available for tracing. TAU will produce a trace for every thread of execution in its modern trace format as well as in OTF [7] and EPILOG [9] formats. TAU also provides mechanisms for online and hierarchical trace merging [3].

## TAU Analysis

As the complexity of measuring parallel performance increases, the burden falls on analysis and visualization tools to interpret the performance information. As shown in Fig. 1, TAU includes sophisticated tools for parallel profile analysis and performance data mining. In addition, TAU leverages advanced trace analysis technologies from the performance tool community, primarily the Vampir [2] and Scalasca [18] tools. The following focuses on the features of the TAU profiling tools.

TAU's parallel profile analysis environment consists of a framework for managing parallel profile data, *PerfDMF* [5], and TAU's parallel profile analysis tool, *ParaProf* [1]. The complete environment is implemented entirely in Java. The performance data management framework (PerfDMF) in TAU provides a common foundation for parsing, storing, and querying parallel profiles from multiple performance experiments. It builds on robust SQL relational database engines and must be able to handle both large-scale performance profiles, consisting of many events and threads of execution, as well as many profiles from

multiple performance experiments. To facilitate performance analysis development, the PerfDMF architecture includes a well-documented data-management API to abstract query and analysis operation into a more programmatic, non-SQL form.

TAU's parallel profile analysis tool, *ParaProf*[1], is capable of processing the richness of parallel profile information produced by the measurement system, both in terms of the profile types (flat, callpath, phase, snapshots) as well as scale. ParaProf provides the users with a highly graphical tool for viewing parallel profile data with respect to different viewing scopes and presentation methods. Profile data can be input directly from a PerfDMF database and multiple profiles can be analyzed simultaneously. ParaProf can show parallel profile information in the form of bargraphs, callgraphs, scalable histograms, and cumulative plots. ParaProf is also capable of integrating multiple performance profiles for the same performance experiment but using different performance metrics for each. ParaProf uses scalable histogram and three-dimensional displays for larger datasets.

To provide more sophisticated performance analysis capabilities, we developed support for parallel performance data mining in TAU. *PerfExplorer* [4, 6] is a framework for performance data mining motivated by our interest in automatic parallel performance analysis and by our concern for extensible and reusable performance tool technology. PerfExplorer is built on PerfDMF and targets large-scale performance analysis for single experiments on thousands of processors and for multiple experiments from parametric studies. PerfExplorer uses techniques such as clustering and dimensionality reduction to manage large-scale data complexity.

## Summary

The TAU Performance System® has undergone several incarnations in pursuit of its primary objectives of flexibility, portability, integration, interoperability, and scalability. The outcome is a robust technology suite that has significant coverage of the performance problem solving landscape for high-end computing. TAU follows a direct performance observation methodology since it is based on the observation of effects directly associated with the program's execution, allowing performance

data to be interpreted in the context of the computation. Hard issues of instrumentation scope and measurement intrusion have to be addressed, but these have been aggressively pursued and the technology enhanced in several ways during TAU's lifetime. TAU is still evolving, and new capabilities are being added to the tools suite. Support for whole-system performance analysis, model-based optimization using performance expectations and knowledge-based data mining, and heterogeneous performance measurement are being pursued.

## Related Entries

- ▶ [Metrics](#)
- ▶ [Performance Analysis Tools](#)

## Bibliography

1. Bell R, Malony A, Shende S (2003) A portable, extensible, and scalable tool for parallel performance profile analysis. In: European conference on parallel computing (EuroPar 2003), Klagenfurt
2. Brunsch H, Kranzmüller D, Nagel WE (2004) Tools for scalable parallel program analysis – Vampir NG and DeWiz. In: Distributed and parallel systems, cluster and grid computing, vol 777. Springer, New York
3. Brunsch H, Nagel W, Malony A (2003) A distributed performance analysis architecture for clusters. In: IEEE international conference on cluster computing (Cluster 2003), pp 73–83. IEEE Computer Society, Los Alamitos
4. Huck KA, Malony AD (2005) Perfexplorer: a performance data mining framework for large-scale parallel computing. In: High performance networking and computing conference (SC'05). IEEE Computer Society, Los Alamitos
5. Huck K, Malony A, Bell R, Morris A (2005) Design and implementation of a parallel performance data management framework. In: International conference on parallel processing (ICPP 2005). IEEE Computer Society, Los Alamitos
6. Huck K, Malony A, Shende S, Morris A (2008) Knowledge support and automation for performance analysis with PerfExplorer 2.0. *J Sci Program* 16(2–3):123–134 (Special issue on large-scale programming tools and environments)
7. Knüpfer A, Brendel R, Brunsch H, Mix H, Nagel WE (2006) Introducing the Open Trace Format (OTF). In: International conference on computational science (ICCS 2006). Lecture notes in computer science, vol 3992. Springer, Berlin, pp 526–533
8. Mayanglambam S, Malony A, Sottile M (2009) Performance measurement of applications with GPU acceleration using CUDA. In: Parallel computing (ParCo), Lyon
9. Mohr B, Wolf F (2003) KOJAK – a tool set for automatic performance analysis of parallel applications. In: European conference on parallel computing (EuroPar 2003). Lecture notes in computer science, vol 2790. Springer, Berlin, pp 1301–1304
10. Nataraj A, Sottile M, Morris A, Malony AD, Shende S (2007) TAUoverSupermon: low-overhead online parallel performance monitoring. In: European conference on parallel computing (EuroPar 2007), Rennes
11. Nataraj A, Morris A, Malony AD, Sottile M, Beckman P (2007) The ghost in the machine: observing the effects of kernel operation on parallel application performance. In: High performance networking and computing conference (SC'07), Reno
12. Nataraj A, Malony A, Morris A, Arnold D, Miller B (2008) In search of sweet-spots in parallel performance monitoring. In: IEEE international conference on cluster computing (Cluster 2008), Tsukuba
13. Nataraj A, Malony A, Morris A, Arnold D, Miller B (2008) TAUoverMRNet (ToM): a framework for scalable parallel performance monitoring. In: International workshop on scalable tools for high-end computing (STHEC '08), Kos
14. Nataraj A, Malony AD, Shende S, Morris A (2008) Integrated parallel performance views. *Clust Comput* 11(1):57–73
15. Shende S (2001) The role of instrumentation and mapping in performance measurement. Ph.D. thesis, University of Oregon
16. Shende S, Malony A (2006) The TAU parallel performance system. *Int J Supercomput Appl High Speed Comput* 20(2, Summer):287–311 (ACTS collection special issue)
17. Shende S, Malony AD, Cuny J, Lindland K, Beckman P, Karmesin S (1998) Portable profiling and tracing for parallel scientific applications using C++. In: SIGMETRICS symposium on parallel and distributed tools, SPDT'98, Welches, pp 134–145
18. Wolf F et al (2008) Usage of the SCALASCA toolset for scalable performance analysis of large-scale parallel applications. In: Proceedings of the second HLRS parallel tools workshop, Stuttgart. Lecture notes in computer science. Springer, Berlin

## TAU Performance System®

- ▶ [TAU](#)

## TBB (Intel Threading Building Blocks)

- ▶ [Intel® Threading Building Blocks \(TBB\)](#)

## Tensilica

- ▶ [Green Flash: Climate Machine \(LBNL\)](#)

## Tera MTA

BURTON SMITH

Microsoft Corporation, Redmond, WA, USA

### Synonyms

Cray MTA; Cray XMT; Horizon

### Definition

The MTA (for Multi-Threaded Architecture) is a highly multithreaded scalar shared-memory multiprocessor architecture developed by Tera Computer Company (renamed Cray Inc. in 2000) in Seattle, Washington. Work began in 1985 at The Institute for Defense Analyses Center for Computing Sciences on a closely related predecessor (Horizon), and development of both hardware and software was continuing at Cray Inc. as of 2010.

### Discussion

#### Introduction

The Tera MTA [1] is in many respects a direct descendant of the Denelcor HEP computer [2]. Like the HEP, the MTA is a scalar shared-memory system equipped with full/empty bits at every 64-bit memory location and multiple protection domains to permit multiprogramming within a processor. However, the MTA introduced a few innovations including VLIW instructions without any register set partitioning, additional ILP via dependence data encoded in each instruction, two-phase blocking synchronization, unlimited data breakpoints, speculative loads, division and square root to full accuracy using iterative methods, operating system entry via procedure calls, traps that never change privilege, and no interrupts at all.

Software developed for the MTA introduced its share of novel ideas as well, including a user-mode runtime responsible for synchronization and work scheduling, negotiated resource management between the user-mode runtime and the operating system, an operating system that returns control to the user-mode runtime when the call blocks, a compiler for Fortran and C++ that parallelizes and restructures a wide variety of loops including those whose inter-iteration dependences require a parallel prefix computation, and dynamic scheduling of loop nests having mixed rectangular, triangular, and skyline loop bounds.

### Beginnings

While spending the summer of 1984 as an intern at Denelcor, UC Berkeley graduate student Stephen W. Melvin invented a scheme for organizing a register file in a fine-grain multi-threaded processor to let VLIW instructions enjoy multiple register accesses per instruction while preserving a flat register address space within each hardware thread. The idea was simple: organize the register file into multiple banks with each bank containing all of the registers for an associated subset of the hardware threads; let each issued instruction use multiple cycles to read and write its associated bank as many times as necessary to implement the instruction; and have the instruction issue logic refrain from issuing from threads associated with currently busy banks. From this beginning, an architectural proposal emerged that was whimsically referred to as “Vulture” and was later known as the “HEP array processor”. Denelcor envisioned heterogeneous systems with second-generation HEP processors sharing memory with processors based on this new VLIW idea.

Denelcor filed for Chapter 7 bankruptcy in mid-1985 whereupon its CTO, Burton J. Smith, joined the Supercomputing Research Center (now called the Center for Computing Sciences) of the Institute for Defense Analyses (IDA) in Maryland. His plan was to further evaluate the merits of the multithreaded VLIW ideas. It had already become clear that code generation and optimization for such a processor was only slightly more difficult than for the HEP but the resulting performance was potentially much higher. The design that resulted from collaborations on this topic at IDA over the years 1985–1987 was known as *Horizon* and was described in a series of papers presented at Supercomputing’88 [3, 4].

*Horizon* instructions were 64 bits wide and typically contained three operations: a memory reference (M) operation, an arithmetic or logic (A) operation, and a control (C) operation which did branches but could also do some arithmetic and logic. As many as ten five-bit register references might appear in any single instruction. The memory reference semantics were derived from those of the HEP but added data trap bits to implement data break points (watchpoints) and possibly other things. The A operations included fused multiply-adds for both integer and floating point arithmetic and a rich variety of operations on vectors and matrices of bits. Branches were encoded compactly as an opcode, an eight-bit condition mask, a two-bit condition code

number specifying one of the four most recently generated three-bit condition codes, and two bits naming a branch target register that had been preloaded with the branch address. Most A- and C-operations emitted a condition code as an optional side-effect.

Horizon increased ILP beyond three with an idea known as *explicit-dependence lookahead*, replacing the usual register reservation scheme. Intel would later employ a kindred concept for encoding dependence information within instructions in the Itanium architecture, calling it *explicitly parallel instruction computing* (EPIC). The Horizon version included in every instruction an explicit 3-bit unsigned integer, the *lookahead* that bounded from below the number of instructions separating this instruction from later ones that might depend on it. Subsequent instructions within the bound could overlap with the current instruction. To implement this scheme, every hardware thread was equipped with a three-bit *flag* counter, incremented when the thread is selected to issue its next instruction, and an array of eight three-bit *lock* counters. On instruction issue, the lock counter subscripted by the lookahead plus the flag (mod 8) is incremented; when the instruction fully retires, that same lock is decremented. A thread is permitted to issue its next instruction when the lock subscripted by its flag is zero. Instruction instances were thus treated very much like operations in a static data flow machine. As a further refinement, branches were available to terminate lookahead along the unlikely control flow direction, potentially increasing ILP along the likelier one.

## Tera

Early in 1988, Smith and a colleague, James E. Rottolk, decided to start a company to build general-purpose parallel computer systems based on the Horizon concepts. They named the new company *Tera*, acquired initial funding from private investors and from the Defense Advanced Research Projects Agency (DARPA), and began to search for a suitable home. In August 1988, Seattle, Washington was chosen and Tera began recruiting engineers. The University of Washington helped the company in its early days.

## Languages

The Tera language strategy [5] was to add directives and pragmas to Fortran and C++ to guide compiler

loop parallelization and provide performance and compatibility with existing vector processors. A consistency model strongly resembling release consistency was adopted based on acquire and release synchronization points to let the compiler cache values in registers and restructure code aggressively. Basically, memory references could not move backward over acquires or forward over releases. Only the built-in synchronization based on full/empty bits was permitted at first; later, volatile variable references were made legal (but deprecated) for synchronization. A *future* statement borrowed from Multilisp [6] was introduced in both Fortran and C++ to support task parallelism as well as divide-and-conquer data parallelism. It uses the full/empty bits to synchronize completion of the future with its invoker. The body of the future appears in-line and can reference variables in its enclosing environment.

## Compiler Optimization

The MTA compilers can automatically parallelize a variety of loops [7]. Consider the example below:

```
void sort(int *src, int *dst, int nitems, int nvals) {
 int i, j, t1[nvals], t2[nvals];
 for (j = 0; j < nvals; j++) {
 t1[j] = 0;
 }
 for (i = 0; i < nitems; i++) {
 t1[src[i]]++; //atomic update
 t2[0] = 0;
 for (j = 1; j < nvals; j++) {
 t2[j] = t2[j-1] + t1[j-1]; //parallel prefix
 $t2 = (sync int *) t2;
 }
 }
 #pragma tera assert parallel
 for (i = 0; i < nitems; i++) {
 dst[$t2[src[i]]++] = src[i];
 }
}
```

All four loops are parallelized by the MTA compiler. Updates like the one in the second loop are automatically made atomic using full/empty bits if and as necessary. The third loop is an example of a *parallel prefix computation* [8] (also called a *parallel scan*) which the compiler can automatically parallelize as long as the internal state of the accumulating prefix is bounded [9]. The fourth loop will not be parallelized automatically by the compiler and requires a pragma and explicit use of full/empty bits via the *sync* type qualifier.

To achieve as high a level of ILP as the instruction set can afford, software pipelining [10] is exploited by distributing loops based on estimates of register pressure and then unrolling and packing to obtain a good schedule. Experiments at both IDA and Tera led to a modification of the lookahead scheme to overlap only memory references. Software pipelining was further enabled by implementing speculative loads. A *poison* bit is associated with every register in the thread, and memory protection violations can optionally be made to poison the destination register instead of raising an exception. Any instruction that attempts to use a poisoned value will trap instead. The speculative load feature allows prefetching in a software pipeline without having to “unpeel” final iterations to prevent accesses beyond mapped memory. In any case the instruction can be reattempted if the cause of the protection violation is remediable.

Nests of parallelizable “for” loops may vary in iterations per loop, sometimes dynamically, making them hard to execute efficiently. The MTA compiler schedules these nests as a whole, even when inner loops have bounds that depend on outer iteration variables. First, code is generated to compute the total number of (inner loop) iterations of the whole nest. Functions are then generated to compute the iteration number of each loop from the total iteration count. Finally, code is generated to dynamically schedule the loop nest by having each worker thread acquire a “chunk” of total iterations using a variant of guided self-scheduling [11], reconstruct the iteration variable bindings, and then jump into the loop nest to iterate until the chunk is consumed.

### User-Level Runtime

A user-level runtime environment was developed to help implement the language features and schedule fine-grain parallel tasks. The Horizon architecture was modified to make full/empty synchronization operations lock the location and generate a user-level trap after a programmable limit on the number of synchronization retry attempts is exceeded. The runtime trap handler saves the state of the blocked task, initializes a queue of blocked tasks containing this one as its first element, and places a pointer to this new queue in the still-locked full/empty location. It then sets a trap bit and unlocks the location so that subsequent references of any kind will trap immediately. In this way the tasks that arrive later

either block, joining the queue of waiting tasks right away, or dequeue a waiting task immediately. The retry limit is set to match the time needed to save and later restore the task state, making this scheme within a factor of two of optimal. When memory references are frequent, the retry rate is throttled to be much less than that for new memory references, and if the processor is not starved for hardware threads the polling cost becomes almost negligible and the retry limit can be increased substantially.

When formerly blocked tasks are unblocked, they are enqueued in a pool of runnable tasks. Since these tasks are equipped with a stack and may have additional memory associated with them, they are run in preference to tasks associated with future statements that have not yet run. Still, the number of blocked tasks can be substantial. To reduce memory waste, stacks are organized as linked lists of fixed-size blocks. Automatic arrays and other things that must be contiguously allocated are stored in the heap. Interprocedural analysis is used to avoid most stack bounds checks.

Another modification to the Horizon architecture comprised instructions to allocate multiple hardware threads. Each protection domain has an operating system-imposed limit on the number of hardware threads in the domain and a *reservation* which can be increased (or decreased) by one of the new instructions [12]. One of them reserves a variable number, from zero up to a maximum specified as an argument, depending on availability. The other instruction either reserves the requested amount or none at all. In either case, the number of additional hardware threads actually allocated is returned in a register so a loop can be used to initiate them. The primary motivation for this reservation capability was to accommodate rapidly varying quantities of parallelism found in short parallel loops. Hardware threads can be materialized and put to work quickly when such opportunities are encountered.

### Operating System

Since the MTA’s operating system (OS) plays no role in user-level thread synchronization and allocates but does not micromanage the dynamic quantity of hardware threads, the usual OS invocation machinery (trapping) was rejected in favor of procedure calls. A protection ring-crossing instruction guarantees only valid OS

entry points are called. The operating system can allocate its own stack space when and if necessary. If an OS call ultimately blocks, the hardware thread is returned to user level via a runtime entry point that associates the continuation of the original user computation with a “cookie” supplied by the OS. When the original OS call completes, the appropriate cookie is passed to the user runtime so it can unblock the user-level continuation. As a result, the operating system executes in parallel with the user-level program. This scheme was invented independently [13] at the University of Washington and is referred to as *scheduler activations*.

When an illegal operation is attempted, a trap to the user-level trap handler occurs. If OS intervention is required, the trap handler calls the OS to service the trap. To evict a process from a protection domain, all of its hardware threads are made to trap in response to an OS-generated signal. The OS also has the ability to kill all hardware threads in a protection domain.

## Memory Mapping

The MTA has a large virtual data address space, making high-performance memory address translation challenging. To address this issue, segmentation is used instead of paging, with a 48-bit virtual address comprising a 20-bit segment number and a 28-bit offset. Contiguous allocations of memory larger than 256 MB can use multiple segments. Segment size granularity is 8 KB. Each protection domain has base and limit registers that define its own range of segment numbers. A segment map entry specifies minimum privilege levels for loads and stores, and whether physical addresses are to be *distributed* across the multiple memory banks of the system by “scrambling” the address bits [14]. When memory is distributed in this fashion there are virtually no bank conflicts due to strided accesses. Program memory, as addressed by the program counters, is handled differently using a paging scheme with 4 KB pages. The program space is mapped to data space via a non-distributed (i.e., local) segment.

## Arithmetic

The instruction set supports the usual variety of two’s complement and unsigned integers and both 32- and 64-bit IEEE 754 floating point. Division for signed and unsigned integers [15] along with floating point division

and square root all use Newton’s method but nevertheless implement full accuracy and correct semantics. Excepting these few operations, denormalized arithmetic is fully supported in hardware. High-precision integer arithmetic is abetted by a 128-bit unsigned integer multiply instruction and the ability to propagate carry bits easily. There is also support for a 128-bit floating point format using pairs of 64-bit floats; the smaller value is insignificant with respect to the larger, thereby yielding 106 bits of significand precision or more. The existence of a fused multiply-add makes this format relatively inexpensive to implement and use, but instruction modifications were needed to mitigate a variety of pathologies. A true 128-bit IEEE format would doubtless be preferable.

## MTA-1 and MTA-2

The MTA-1 was a water-cooled system built from Gallium Arsenide logic. To provide adequate memory bandwidth, the processors sparsely populated a 3D toroidal mesh interconnection network. As an example, 512 routing nodes were required for 64 processors. To make network wiring implementable, one-third of the mesh links were elided. The first and only MTA-1 system was delivered to the San Diego Supercomputer Center in June 1999.

The MTA-2 was a major improvement in manufacturability over the MTA-1. It used CMOS logic and had an interconnection network based on notions from group theory [16]. It was first delivered in 2002 to the US Naval Research Laboratory in Washington, DC. A few other MTA-2 systems were built before deliveries of the Cray XMT (*q.v.*) began in 2007.

## Related Entries

- ▶ [Cray MTA](#)
- ▶ [Cray XMT](#)
- ▶ [Data Flow Computer Architecture](#)
- ▶ [Denelcor HEP](#)
- ▶ [EPIC Processors](#)
- ▶ [Futures](#)
- ▶ [Interconnection Networks](#)
- ▶ [Latency Hiding](#)
- ▶ [Little’s Law](#)
- ▶ [Memory Wall](#)
- ▶ [MIMD \(Multiple Instruction, Multiple Data\) Machines](#)

- ▶ [Modulo Scheduling and Loop Pipelining](#)
- ▶ [Multilisp](#)
- ▶ [Multi-Threaded Processors](#)
- ▶ [Networks, Direct](#)
- ▶ [Networks, Multistage](#)
- ▶ [Processes, Tasks, and Threads](#)
- ▶ [Processors-in-Memory](#)
- ▶ [Shared-Memory Multiprocessors](#)
- ▶ [SPMD Computational Model](#)
- ▶ [Synchronization](#)
- ▶ [Ultracomputer, NYU](#)
- ▶ [VLIW Processors](#)

## Bibliography

1. Alverson R, Callahan D, Cummings D, Koblenz B, Porterfield A, Smith B (1990) The Tera computer system. In: Proceedings of the 1990 international conference on supercomputing, Amsterdam
2. Smith BJ (1981) Architecture and applications of the HEP multiprocessor computer system. Proc SPIE Real-Time Signal Process IV 298:241–248
3. Kuehn JT, Smith BJ (1988) The Horizon supercomputing system: architecture and software. In: Proceedings of the 1988 ACM/IEEE conference on supercomputing, Orlando
4. Thistle MR, Smith BJ (1988) A processor architecture for Horizon. In: Proceedings of the 1988 ACM/IEEE conference on supercomputing, Orlando
5. Callahan D, Smith B (1990) A future-based parallel language for a general-purpose highly-parallel computer. In: Selected papers of the second workshop on languages and compilers for parallel computing, Irvine
6. Halstead RH (1985) MultiLisp: a language for concurrent symbolic computation. ACM T Program Lang Syst 7(4):501–538
7. Alverson G, Briggs P, Coatney S, Kahan S, Korry R (1997) Tera hardware-software cooperation. In: Proceedings of supercomputing, San Jose
8. Ladner RE, Fischer MJ (1980) Parallel prefix computation. J ACM 27(4):831–838
9. Callahan D (1991) Recognizing and parallelizing bounded recurrences. In: Proceedings of the fourth workshop on languages and compilers for parallel computing, Santa Clara
10. Lam M (1988) Software pipelining: an effective scheduling technique for VLIW machines. In: Proceedings of the ACM SIGPLAN 88 conference on programming language design and implementation, Atlanta
11. Polychronopoulos C, Kuck D (1987) Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. IEEE T Comput C-36(12):1425–1439
12. Alverson G, Alverson R, Callahan D, Koblenz B, Porterfield A, Smith B (1992) Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In: Proceedings of the 1992 international conference on supercomputing, Washington, DC
13. Anderson T, Bershad B, Lazowska E, Levy H (1992) Scheduler activations: effective kernel support for the user-level management of parallelism. ACM T Comput Syst 10(1):53–79
14. Norton A, Melton E (1987) A class of Boolean linear transformations for conflict-free power-of-two stride access. In: Proceedings of the international conference on parallel processing, St. Charles, IL
15. Alverson R (1991) Integer division using reciprocals. In: Proceedings of the 10th IEEE symposium on computer arithmetic, Grenoble
16. Akers S, Krishnamurthy B (1989) A group-theoretic model for symmetric interconnection networks. IEEE T Comput C-38(4):555–566
17. Alverson G, Kahan S, Korry R, McCann C, Smith B (1995) Scheduling on the Tera MTA. In: Proceedings of the first workshop on job scheduling strategies for parallel processing, Santa Barbara. Lecture Notes in Computer Science 949:19–44

## Terrestrial Ecosystem Carbon Modeling

DALI WANG<sup>1</sup>, DANIEL RICCIUTO<sup>1</sup>, WILFRED POST<sup>1</sup>,

MICHAEL W. BERRY<sup>2</sup>

<sup>1</sup>Oak Ridge National Laboratory, Oak Ridge, TN, USA

<sup>2</sup>The University of Tennessee, Knoxville, TN, USA

## Synonyms

Carbon cycle research; System integration; Terrestrial ecosystem modeling; Uncertainty quantification

## Definition

A Terrestrial Ecosystem Carbon Model (TECM) is a category of process-based ecosystem models that describe carbon dynamics of plants and soils within global terrestrial ecosystems. A TECM generally uses spatially explicit information on climate/weather, elevation, soils, vegetation, and water availability as well as soil- and vegetation-specific parameters to make estimates of important carbon fluxes and carbon pool sizes in terrestrial ecosystems.

## Discussion

### Introduction

Terrestrial ecosystems are a primary component of research on global environmental change. Observational and modeling research on terrestrial ecosystems at the global scale, however, has lagged behind

their counterparts for oceanic and atmospheric systems, largely because of the unique challenges associated with the tremendous diversity and complexity of terrestrial ecosystems. There are eight major types of terrestrial ecosystem: tropical rain forest, savannas, deserts, temperate grassland, deciduous forest, coniferous forest, tundra, and chaparral. The carbon cycle is an important mechanism in the coupling of terrestrial ecosystems with climate through biological fluxes of CO<sub>2</sub>. The influence of terrestrial ecosystems on atmospheric CO<sub>2</sub> can be modeled via several means at different timescales to incorporate several important processes, such as plant dynamics, change in land use, as well as ecosystem biogeography. Over the past several decades, many terrestrial ecosystem models (see the “►Model Developments” section) have been developed to understand the interactions between terrestrial carbon storage and CO<sub>2</sub> concentration in the atmosphere, as well as the consequences of these interactions. Early TECMs generally adapted simple box-flow exchange models, in which photosynthetic CO<sub>2</sub> uptake and respiratory CO<sub>2</sub> release are simulated in an empirical manner with a small number of vegetation and soil carbon pools. Demands on kinds and amount of information required from global TECMs have grown. Recently, along with the rapid development of parallel computing, spatially explicit TECMs with detailed process-based representations of carbon dynamics become attractive, because those models can readily incorporate a variety of additional ecosystem processes (such as dispersal, establishment, growth, mortality, etc.) and environmental factors (such as landscape position, pest populations, disturbances, resource manipulations, etc.), and provide information to frame policy options for climate change impact analysis.

### Key Components of TECM

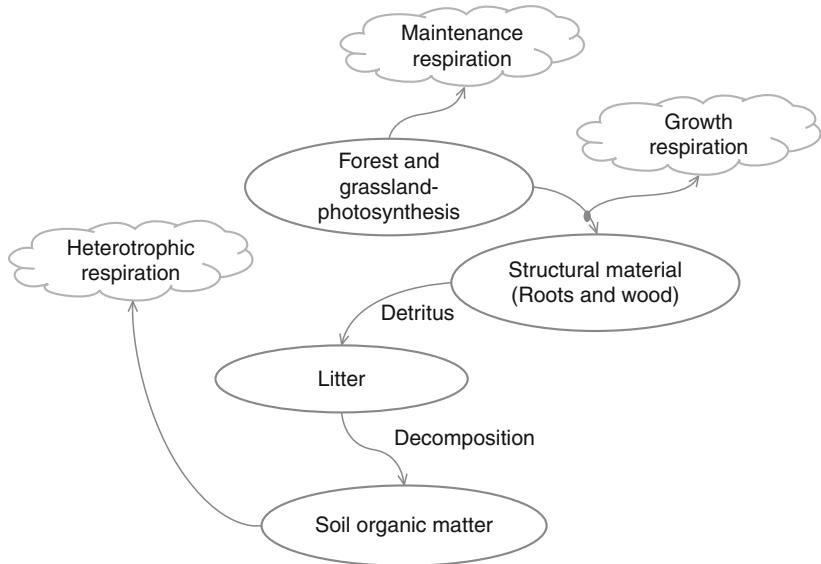
#### 1. Fundamental terrestrial ecosystem carbon dynamics

Terrestrial carbon processes can be described by an exchange between four major compartments: (1) foliage where photosynthesis occurs; (2) structural material, including roots and wood; (3) surface detritus or litter; and (4) soil organic matter (including peat). Nearly all life on Earth depends (directly or indirectly) on photosynthesis, in which carbon dioxide and water are used,

and oxygen is released. The majority of the carbon in the living vegetation of terrestrial ecosystems is found in woody material, which constitutes a major carbon reservoir in the carbon cycle. Detritus refers to leaf litter and other organic matter on or below the soil surface. Dead woody material, often called coarse woody debris, is a large component of the surface detritus in forest ecosystems. Detritus is typically colonized by communities of microorganisms which act to decompose (or remineralize) the material. Transformation and translocation of detritus is the source of soil organic matter, another major component in the global carbon cycle. Globally three times as much carbon is stored in soils as in the atmosphere with peatlands contributing a third of this. Thus even relatively small changes in soil C stocks might contribute significantly to atmospheric CO<sub>2</sub> concentrations and thus global climate change. The soil carbon pool is vulnerable to impacts of human activity especially agriculture. A simplified scheme of carbon cycle dynamics is shown in Fig. 1.

#### 2. Terrestrial carbon observations and experiments

Early research generally focused on determining characteristics of individual plants and small soil samples, often in a laboratory setting. This type of research continues today and provides a wealth of information that is used to develop and to parameterize TECMs. However, successful modeling of the carbon cycle also requires understanding the structure and response of entire ecosystems. Observation networks involving ecosystems have expanded greatly in the past two decades. One important development for in situ monitoring of ecosystem-level carbon exchange has been the establishment of flux towers that use the eddy covariance method. Atmospheric CO<sub>2</sub> concentration measurements using satellites, tall towers, and aircraft provide information about carbon dioxide fluxes over a larger scale. Finally, remote sensing products provide important information about changes in land use and vegetation characteristics (e.g., total leaf area) that can be used to either drive or validate TECMs. While these observations are important for characterizing the carbon cycle at present, they do not provide information about how the carbon cycle may change in the future as a result of climate change. In order to address those challenging questions, several large-scale ecosystem-level experiments have been conducted to mimic possible



**Terrestrial Ecosystem Carbon Modeling. Fig. 1** Simplified schematic of the carbon dynamics (photosynthesis, autotrophic respiration, allocation, and heterotrophic respiration) within a typical TECM model

future conditions (such as rising CO<sub>2</sub> concentrations, potential future precipitation patterns and the future temperature scenarios) and associated impacts and mitigation options for terrestrial ecosystems. Figure 2 illustrates some of these carbon observation systems and experiments.

### 3. Terrestrial ecosystem carbon model developments

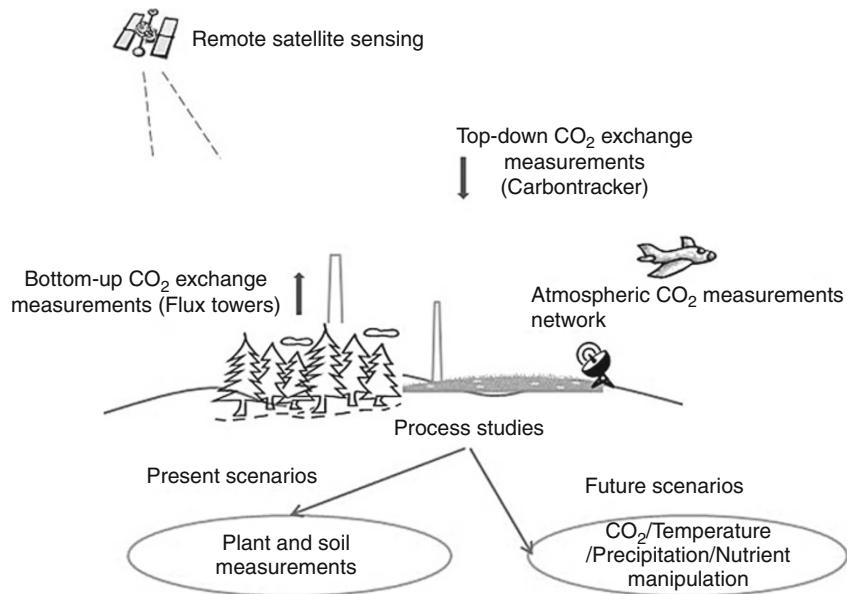
In the early 1970s, several process-based conceptual models were developed to study the primary productivity of the biosphere and the uptake of anthropogenic CO<sub>2</sub> emissions. Early box models were improved in the 1980s to include more spatially explicit ecological representations of terrestrial ecosystems along with a significant push to understand the relationships between climatic measurements and properties of ecosystem processes. The concept of biomes was used to categorize terrestrial ecosystems using several climatically and geographically related factors (i.e., plant structure, leaf types, and climate), instead of the traditional classification by taxonomic similarity. In the 1990s, rapid developments of general circulation models and scientific computing, along with the increasing availability of remote sensing data (from satellites), led to the development of land-surface models. These

models used satellite images to obtain information about the spatial distribution of surface properties (such as vegetation type, phenology, and density) along with spatially explicit forcing from numerical weather prediction reanalysis or coupled general circulation models (e.g., temperature and precipitation) to improve prediction and enhance the model representation of land-atmosphere water and energy interactions within global climate models. Recently, emphasis has been focused on improving the predictive capacity of climate models at the decadal to century scale through a better characterization of carbon cycle feedbacks with climate. For example, several TECMs are incorporating nutrient cycles and shifts in vegetation distribution, in response and a potential feedback to climate change.

### The Contributions of Parallel Computing to TECM Developments

The contributions of parallel computing to the TECM can be classified into three separate categories: (1) model construction, (2) model integration, and (3) model behavior controls.

As more processes are incorporated into TECMs to replace simple empirical relationships, computational demands have increased. Since these processes are very



**Terrestrial Ecosystem Carbon Modeling.** Fig. 2 Illustrated carbon observations and experiments

sensitive to environmental heterogeneity inherent in spatial patterns of temperature, radiation, precipitation, soil characteristics, etc., increased spatial resolution improved simulation accuracy. A new class of models is becoming more widespread for global scale TECMs. This class, instead of using a traditional system of differential equations, is agent-based and requires a fine grid spatial representation to represent the competition for resources among the coexisting agents. This approach dramatically increases the computational demand, but is more compatible with experimental and observational data and population scale vegetation change processes. Parallel computing has enabled models to be constructed with these additional complexities.

Over several decades of research, TECMs have dramatically changed in structure and in the amount and kind of information required and produced making model integration a challenge. In addition, these models are now being incorporated into climate models to form Earth System Models (ESMs). From a parallel computing perspective, there is huge demand from the modeling community to develop a parallel model coupling framework (Earth System Modeling Frame (ESMF) is one of these kinds of efforts) to enable further parallel model developments and validations.

Instead of rewriting a package *wrapper* for each component, memory-based IO staging systems may provide an alternative method for fast and seamless coupling. There are two basic methods to provide climate forcing information for a TECM, from observation data or coupling to a global climate model simulation. Currently, model simulation can provide global data, but only at low spatial resolutions. Observation datasets are available at those fine resolutions, and can be used for validation over observed time frames at those observation stations. However, further research and parallel computing will be needed for gap-filling and downscaling those observation datasets for global terrestrial ecosystem carbon modeling.

One consequence of TECM complexity is the increasing demand for better methodologies that can exploit ever-increasing rich datastreams and thereby improve model behavior (e.g., the ecosystem model's sensitivity to the model parameters, and software structure). Quantitative methods need to be established to determine model uncertainty and reduce uncertainty through model-data analysis. As computers become larger and larger in the number of CPU cores, not necessarily faster and faster at the single CPU core level, we envision that ultrascale software designs for systematic

uncertainty quantification for TECM will become one research area which will require the full advantage of parallel computing and statistics.

As our understanding of the global carbon cycle improves, high fidelity, process-based models will continue to be developed, and the increasing complexity of these ecosystem model systems will require that parallel computing play an increasingly important role. We have explained several key components of terrestrial ecosystem carbon modeling, and have classified three categories that parallel computing can play significant contributions to the TECM developments. It is our view that parallel computing will increasingly be an integral part of modern terrestrial ecosystem modeling efforts, which require solid, strong partnerships between the high-performance computing community and the carbon cycle science community. Through such partnerships these two communities can share a common mission to advance our understanding of global change using computational sciences.

## Related Entries

- [Analytics, Massive-Scale](#)
- [Computational Sciences](#)
- [Exascale Computing](#)

## Bibliographic Notes and Further Reading

As mentioned in the model development section, terrestrial carbon modeling started in the early 1970s [1, 2], when beta-factor concept was developed to account for CO<sub>2</sub> fertilization using a nonspatial representation of terrestrial carbon dynamics. In 1975, Lieth described a model (MIAMI, the first gridded model) [3] to estimate the primary productivity of the biosphere. A carbon accounting model was developed at Marine Biological Laboratory (MBL) at Woods Hole to track carbon fluxes associated with land-use change. Along with the success of International Biological Program, spatially distributed compartment models representing different ecosystem types responding to local environmental conditions were developed. Widely used examples include the Terrestrial Ecosystem Model ([www.mbl.edu/eco42/](http://www.mbl.edu/eco42/)) at MBL, and CENTURY ([www.nrel.colostate.edu/projects/century/](http://www.nrel.colostate.edu/projects/century/)) at Colorado State University. As satellite measurements of basic terrestrial

properties became available, several models were developed that utilized this information directly, including the Ames-Stanford Approach (CASA) ([geo.arc.nasa.gov/sge/casa/bearth.html](http://geo.arc.nasa.gov/sge/casa/bearth.html)) and Biome-BGC ([www.ntsg.umt.edu/models/bgc/](http://www.ntsg.umt.edu/models/bgc/)). In the 1990s, land surface components of climate models incorporated an aspect of terrestrial carbon cycling, namely photosynthesis, for the purpose of providing a mechanistic model of latent heat exchange with the atmosphere. The Simple Biosphere (SiB) biophysical model [4] and Biosphere-Atmosphere Transfer Scheme (BATS) [5] at National Center for Atmospheric Research (NCAR), and STOMATE [6] at Laboratoire des Sciences du Climat et de l'Environnement (LSCE) are examples. Later at NCAR, additional components of terrestrial carbon cycle were included in the Land Surface Model (LSM) ([www.cgd.ucar.edu/tss/lsm/](http://www.cgd.ucar.edu/tss/lsm/)). The Community Land Model (CLM-CN) ([www.cgd.ucar.edu/tss/clm/](http://www.cgd.ucar.edu/tss/clm/)) is the successor of LSM and is being further developed as a community-based model. Agent-based models at the global scale, a class of what are called Dynamic Global Vegetation Models (DGVM), have been developed independently because of their data and computation demands. Developments in parallel computer systems are making incorporation of such dynamics plausible for earth system models. HYBRID [7] was an early experimental model, and now prototypes exist for the NCAR CCSM land surface CLM-CN which is based on the Lund-Postdam-Jena (LPJ) model [8] and called CLM-CN-DV. Evolved from the Ecosystem Demography (ED) model [9, 10], ENT [11] is another Dynamic Global Terrestrial Ecosystem Model (DGTEM), being coupled with NASA's GEOS-5 General Circulation Models.

Observation networks involving ecosystems have expanded greatly in the past two decades. AmeriFlux ([public.ornl.gov/ameriflux/](http://public.ornl.gov/ameriflux/)) is an effort to use flux towers to monitor ecosystem-level carbon exchange with atmosphere. Since 1990, more than 400 such flux towers have been established on six continents representing every major biome. First established in Mauna Loa in 1958, the CO<sub>2</sub> measurements have become a global operation. Inversion techniques are used to infer the pattern of CO<sub>2</sub> fluxes required to produce the observed CO<sub>2</sub> concentrations; one such product using this technique is CarbonTracker ([www.esrl.noaa.gov/gmd/ccgg/carbontracker/](http://www.esrl.noaa.gov/gmd/ccgg/carbontracker/)), which provides weekly flux



estimates that can be compared against output from process-based TECMs. Currently, a variety of remote sensing products (such as Moderate Resolution Imaging Spectroradiometer (MODIS)) are available to either drive or validate TECMs.

Several experiments have been conducted or initiated to understand potential climate change impacts. The Free-Air CO<sub>2</sub> Enrichment (FACE) ([public.ornl.gov/face/global\\_face.shtml](http://public.ornl.gov/face/global_face.shtml)) experiment has been running for over a decade at several sites in different biomes to study the potential effects of higher CO<sub>2</sub> concentration. The Throughfall Displacement Experiment (TDE) ([tde.ornl.gov/](http://tde.ornl.gov/)) used elaborate systems to alter the amount of precipitation that is available to an ecosystem. A new experiment has been initiated at Oak Ridge National Laboratory to assess the responses of northern peatland ecosystems to increased temperature and exposures to elevated atmospheric CO<sub>2</sub> concentrations ([mnspruce.ornl.gov](http://mnspruce.ornl.gov)).

More information on terrestrial ecosystem carbon modeling can be found in books devoted to this subject [12, 13].

## Bibliography

1. Bacastow RB, Keeling CD (1973) Atmospheric carbon dioxide and radiocarbon in the natural carbon cycle: II. Changes from AD 1700 to 2070 as deduced from a geochemical model. In: Woodwell GM, Pecan EV (eds) Carbon and the biosphere. CONF-720510. National Technical Information Service, Springfield, Virginia, pp 86–135
2. Emanuel WR, Killough GG, Post WM, Shugart HH (1984) Modeling terrestrial ecosystems in the global carbon cycle with shifts in carbon storage capacity by land-use change. *Ecology* 65(3): 970–983
3. Lieth H (1975) Modeling the primary productivity of the world. In: Lieth H, Wittaker RH (eds) Primary productivity of the biosphere, ecological studies, vol 14. Springer-Verlag, New York, pp 237–283
4. Sellers JP, Randell DA, Collatz GJ, Berry JA, Field CB, Dazlich DA, Zhang C, Collelo GD, Bounua L (1996) A revised land surface parametrization (SiB 2) for atmospheric GCMs. Part I: model formulation. *J Climate* 9:676–705
5. Dickinson R, Henderson-sellers A, Kennedy P (1993) Biosphere-atmosphere transfer scheme (BATS) version as coupled to the NCAR community climate model. Technical report, National Center for Atmospheric Research
6. Ducoudré N, Laval K, Perrier A (1993) SECHIBA, a new set of parametrizations of the hydrologic exchanges at the land/atmosphere interface within the LMD atmospheric general circulation model. *J Climate* 6(2):248–273
7. Friend AD, Stevens AK, Knox RG, Cannell MGR (1997) A process-based, terrestrial biosphere model of ecosystem dynamics (Hybrid v3.0). *Ecol Model* 95:249–287
8. Prentice IC, Heimann M, Sitch S (2000) The carbon balance of the terrestrial biosphere: ecosystem models and atmospheric observations. *Ecol Appl* 10:1553–1573
9. Moorcroft P, Hurtt GC, Pacala SW (2001) A method for scaling vegetation dynamics: the ecosystem demography model (ED). *Ecol Monogr* 71(4):557–586
10. Govindarajan S, Dietze MC, Agarwal PK, Clark JS (2004) A scalable simulator for forest dynamics. In: Proceedings of the twentieth annual symposium on computational geometry SCG 04, Brooklyn, NY, pp 106–115, doi:10.1145/997817.997836
11. Yang W, Ni-Meister W, Kiang NY, Moorcroft P, Strahler AH, Oliphant A (2010) A clumped-foliage canopy radiative transfer model for a Global Dynamic Terrestrial Ecosystem Model II: Comparison to measurements. *Agricultural and Forest Meteorology*, 150(7):895–907, doi:10.1016/j.agrformet.2010.02.008
12. Trabalka JR, Reichle DE (ed) (1986) The changing carbon cycle: a global analysis. Springer-Verlag, Berlin
13. Field CB, Raupach MR (ed) (2004) The global carbon cycle: integrating human, climate, and the natural world. Island, Washington, DC

## Terrestrial Ecosystem Modeling

### ► Terrestrial Ecosystem Carbon Modeling

## The High Performance Substrate

### ► HPS Microarchitecture

## Theory of Mazurkiewicz-Traces

### ► Trace Theory

## Thick Ethernet

### ► Ethernet

## Thin Ethernet

### ► Ethernet

## Thread Level Speculation (TLS) Parallelization

- ▶ Speculation, Thread-Level

## Thread-Level Data Speculation (TLDS)

- ▶ Speculative Parallelization of Loops
- ▶ Speculation, Thread-Level

## Thread-Level Speculation

- ▶ Speculative Parallelization of Loops
- ▶ Speculation, Thread-Level

## Threads

- ▶ Processes, Tasks, and Threads

## Tiling

FRANÇOIS IRIGOIN  
MINES ParisTech/CRI, Fontainebleau, France

### Synonyms

Blocking; Hyperplane partitioning; Loop blocking;  
Loop tiling; Supernode partitioning

### Definition

Tiling is a program transformation used to improve the spatial and/or temporal memory locality of a loop nest by changing its iteration order, and/or to reduce its synchronization or communication overhead by controlling the granularity of its parallel execution. Tiling adds some control overhead because the number of loops is doubled, and reduces the amount of parallelism available in the outermost loops. The  $n$  initial loops

are replaced by  $n$  outer loops used to enumerate the tiles and  $n$  inner loops used to execute all the iterations within a tile.

## Discussion

### Introduction

Tiling is useful for most recent parallel computer architectures, with shared or distributed memory, since they all rely on locality to exploit their memory hierarchies and on parallelism to exploit several cores. It is also useful for heterogeneous architectures with hardware accelerators, and for monoprocessors with caches. Unlike many loop transformations, tiling is not a unimodular transformation. Iterations that are geometrically close in the loop nest iteration set are grouped in so-called *tiles* to be executed together atomically. Tiles are also called blocks when their edges are parallel to the axes or more generally when their facets are orthogonal to the base vectors. For instance, the parallel stencil written in C:

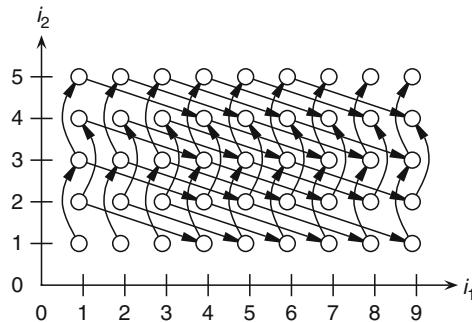
```
for(i1=1;i1<n;i1++)
 for(i2=1;i2<m;i2++)
 a[i1][i2] = 0.2*(b[i1-1][i2]
 +b[i1][i2]
 +b[i1+1][i2]
 +b[i1][i2-1]
 +b[i1][i2+1])
```

can be transformed into:

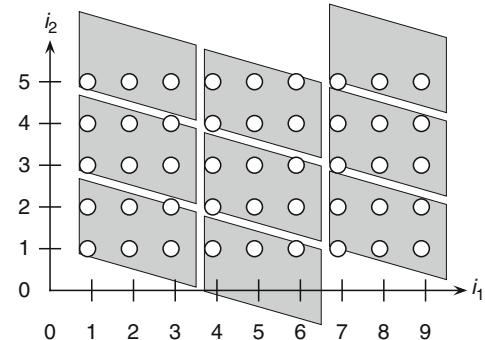
```
#pragma omp parallel for
for(t1=1;t1<n;t1+=b1)
#pragma omp parallel for
for(t2=1;t2<m;t2+=b2)
 // tile code
 for(i1=t1;i1<min(t1+b1, n);i1++)
 for(i2=t2;i2<min(t2+b2, m);i2++)
 a[i1][i2] = 0.2*(b[i1-1][i2]
 +b[i1][i2]
 +b[i1+1][i2]
 +b[i1][i2-1]
 +b[i1][i2+1])
```

where  $t_1$  and  $t_2$  are tile coordinates and  $b_1$  and  $b_2$  are the tile or block sizes.

Initially, this tiling transformation was called loop blocking by Allen & Kennedy and tiling by Wolfe [32, 33] before it was extended to slanted tiles under the name of supernode partitioning by Irigoin and Triolet [21]. Wolfe suggested to use systematically the



Tiling. Fig. 1 Iteration space with dependence vectors



Tiling. Fig. 2 Tiled iteration set

name tiling as it is short and easy to understand. He uses it in his textbook about program transformations [34]. But loop blocking is still used because it is a proper subset of tiling: see, for instance, Allen and Kennedy [4].

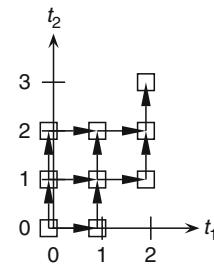
Slanted tiles can be used with these sequential Fortran loops taken from Xue [35]

```
do i1 = 1, 9
 do i2 = 1, 5
 a(i1,i2) = a(i1,i2-2)+a(i1-3,i2+1)
 enddo
enddo
```

whose 2-D iteration set and dependence vectors are shown in Fig. 1 (All figures are taken or derived from Figure 4.1, page 103, of Xue [35] by courtesy of the publisher. Some were adapted or derived to fit the notations used in this entry.). These two loops can be transformed into

```
do t1 = 0, 2
 do t2 = max(it-1,0), (it+4)/2
 do i1 = 3*t1+1, 3*t1+3
 do i2 = max(-t1+2*t2+1,1),
 min((-t1+6*t2+9)/3,5)
 a(i1,i2) = a(i1,i2-2)
 +a(i1-3,i2+1)
 enddo
 enddo
 enddo
enddo
```

using slanted tiles with vectors  $(3, -1)$  and  $(0, 2)$  shown in Fig. 2. The sets of integer points in each tile are not slanted in this case, but they are not horizontally aligned as shown by the grey areas. The tile iteration set is shown in Fig. 3. Note that Fortran allows negative array indices,



Tiling. Fig. 3 Iteration set for tiles, with tile dependence vectors

which makes references to  $a(1, -1)$  and  $a(-2, 2)$  possibly legal.

Mathematically speaking, this grouping/blocking is a partition of the loop iteration space that induces a renumbering and a reordering of the iterations. This reordering should not modify the program semantics. Hence, several issues are linked to tiling as to any other program transformations: Why should tiling be considered? What are the legal tilings for a given loop nest? How is an optimal tiling chosen? How is the transformed code generated?

T

## Motivations for Tiling

Tiling has several positive impacts. Depending on the target architecture, it reduces the synchronization overhead, the communication overhead, the cache coherency traffic, the number of external memory accesses, and the amount of memory required to execute a loop nest in an accelerator or a scratch pad memory, or the number of cache misses. Thus, the execution time and/or the energy used to execute the loop

nest are reduced, or the execution with a small memory is made possible.

Tiling also has two possibly negative impacts. The control overhead is increased, if only because the number of loops is doubled, and the amount of parallelism degree is smaller at the tile level because the initial parallelism is partly transferred within each tile and traded for locality and communication and synchronization overheads. The control overhead depends on the code generation phase, especially when partial tiles are needed to cover the boundaries of the iteration set.

Tile selection depends on the target architecture. For shared memory multiprocessors, including multicores, the primary bottleneck is the memory bandwidth and tiling is used to improve the cache hit ratio by reducing the memory footprint, that is, the set of live variables that should be kept in cache, and to reduce the cache coherency traffic. Some array references in the initial code must exhibit some spatial and/or temporal locality for tiling to be beneficial.

Tiling can be applied again, recursively or hierarchically, to increase locality at the different cache levels ( $L_0, L_1, L_2, L_3, \dots$ ) and even at the register level by using very small tiles compatible with the number of registers. These register tiles are fully unrolled to exploit the registers. The tiling of tiles is also known as multilevel tiling. Tiling can also be used to increase locality at the virtual memory page level as shown in 1969 by McKeller and Coffman in [24].

For vector multiprocessors, the size of the tiles must be large enough to use the vector units efficiently, but small enough for their memory footprint to fit in the local cache, which is one of many trade-offs encountered in tile selection.

For distributed memory multiprocessors, tiling is used to generate automatically distributed code. The tiles are mapped on the processors and the processors communicate data on or close to the tile boundaries. Let  $p$  be the edge length of a  $n$ -dimensional tile. The idea is to compute  $O(p^n)$  values and exchange only  $O(p^{n-1})$  values so as to overlap the computation with communication although computations are faster than communications. The amount of memory on each processor is supposed large enough not to be a constraint, but the value of  $p$  is adjusted to trade parallelism against communication and synchronization overheads. As mentioned above, these large tiles can be tiled again if

locality or parallelism is an issue at the elementary processor level.

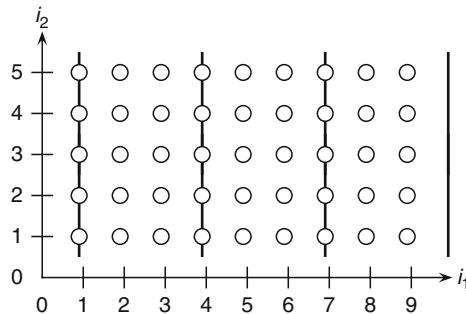
Heterogeneous processors using hardware accelerators, FPGA- or GPGPU- based, require the same kind of trade-offs. Either large tiles must be executed on the accelerator to benefit from its parallel architecture, or small tiles only are possible because of the limited local memory, but in both cases communications between the host and the accelerator must take place asynchronously during the computation. Tiling is used to meet the local memory or vector register size constraints, and to generate opportunities for asynchronous transfers overlapped with the computation.

Multiprocessors System-on-Chip (MPSoC) designed for embedded processing may combine some local internal memories, a.k.a. scratchpad memories, and a global external memory, which make them distributed systems with a global memory. Tiling is used to meet the local memory constraints, but communications between the processors or between the processors and the external global memory must be generated. Other transformations, such as *loop fusion* and *array contraction* (see ▶Parallelization, Automatic), are used in combination with tiling to reduce the communication and the execution time. A combination of loop fusion and loop distribution may give a better result than the tiling of fused loops, at least when the scratchpad memory is very small.

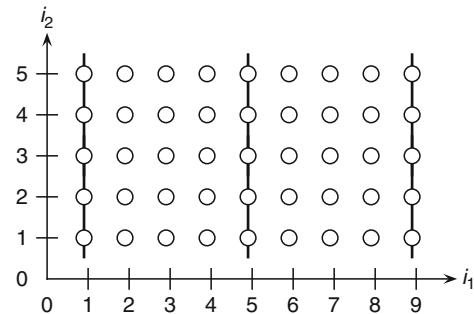
Tiling can also be applied to multidimensional arrays instead of loop nests. This approach is used by *High-Performance Fortran* (see ▶High Performance Fortran (HPF)). The code generation is derived from the initial code and from mapping constraints, based, for instance, on the *owner-computes* rule: the computation must be located where the result is stored. This idea may also be applied to speed up array IOs and out-of-core computations.

Finally, tiling can be applied to more general spaces and sets. For instance, Griebl [15] use the ▶Polyhedron Model to map larger pieces of code to a unique space of large dimension. Sequences of loops can be mapped onto such a space and be tiled globally.

Because of the many machine architectures that can benefit from tiling, numerous papers have been published on the subject. It is important to check what kind of architecture is targeted before reading or comparing them.



**Tiling.** Fig. 4 First hyperplane partitioning with  $h_1 = (\frac{1}{3}, 0)$  and  $o = (1, 1)$



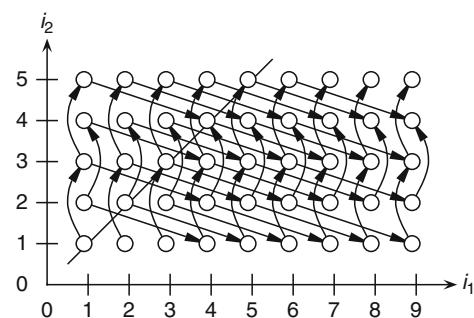
**Tiling.** Fig. 5 Same hyperplane partitioning with a smaller  $h_1 = (\frac{1}{4}, 0)$

### Legality of Tiling

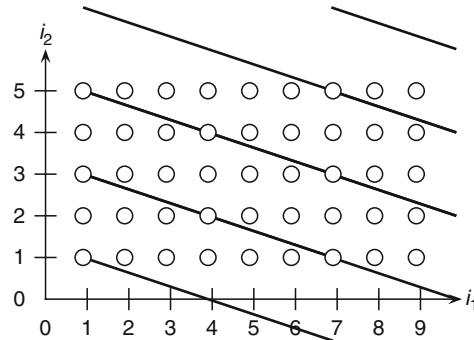
The general definition and legality of tiling were introduced by Irigoin and Triolet who gave sufficient legality conditions in [21]. Necessary conditions were added later by Xue [35].

The basic idea is to use parallel hyperplanes defined by a normal vector  $h$  to slice the iteration space  $Z^n$ , where  $n$  is the number of nested loops, to obtain a partition. The partition of a set  $E$  is a set  $P$  of nonempty subsets of  $E$ , whose two-by-two intersection is always empty and whose union is equal to  $E$ . Each slice is mathematically speaking a part (there is no agreement about the naming of elements of  $P$ : part, block or cell are used. We chose to use *part*) and two iteration vectors  $j_1$  and  $j_2$  belong to the same part if  $[h.(j_1 - o)] = [h.(j_2 - o)]$ , where  $.$  denotes the scalar product,  $[ ]$  the floor function, and  $o$  is an offset vector. For instance, using the normal vector  $h = (\frac{1}{3}, 0)$  and the offset  $o = (1, 1)$ , the iteration set of Fig. 1 is partitioned in three parts shown in Fig. 4. Because of this definition, the slices, or parts become larger when the norm of  $h$  decreases (see Fig. 5). To make sure that the parts can be executed one after the other, dependence cycles between two parts must be avoided. For instance, the diagonal partition in Fig. 6 creates a cycle between the two subsets. Iteration (2, 1) must be executed before iteration (2, 3), so the left subset must be executed before the right subset, but iteration (1, 2) must be executed before iteration (4, 1), which is incompatible.

Cycles between subsets are avoided if each dependence vector  $d$  in the loop nest meets the condition  $h.d \geq 0$ . This condition does depend neither on  $\|h\|$ , the norm of  $h$ , nor on the iteration set, which makes



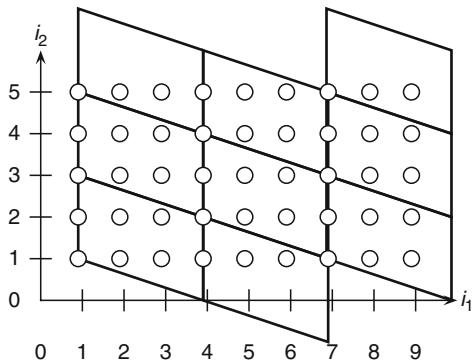
**Tiling.** Fig. 6 Dependence cycle between two parts



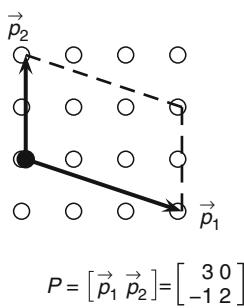
**Tiling.** Fig. 7 Second hyperplane partitioning with  $h_2 = (\frac{1}{6}, \frac{1}{2})$  and  $o = (1, 1)$

all such legal tilings scalable to meet the different needs enumerated above.

Several hyperplane partitionings  $h_1, h_2, \dots$  can be combined to increase the number of parts and reduce their sizes (see Figs. 7 and 8). The vectors  $h_1, h_2, \dots$  are usually grouped (Xue [35] uses  $H$  to denote the transposed matrix of  $H$ , that is, the  $h$  vectors are transformed into affine forms) together in a matrix,  $H$ . When the



Tiling. Fig. 8 Combined 2-D hyperplane partitioning

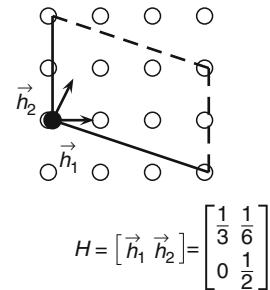


Tiling. Fig. 9 Partitioning or clustering matrix

number of different hyperplane families  $h_i$  used is equal to the number of nested loops  $n$  and when the  $h_i$  are linearly independent vectors, the part sizes are bounded, regardless of the iteration set, and all parts of the iteration space are equal up to a translation if  $H^{-1}$  is an integer matrix. However, the parts of the iteration set  $L$  may differ because of the loop boundaries. See, for instance, tile (2, 3), the upper right tile in Fig. 2, which contains only three iterations.

The transpose of  $H^{-1}$ , the partitioning matrix  $P$ , contains the edges of the tile and its determinant is the number of iterations within each tile. So  $P$  (Fig. 9) is easier to visualize than  $H$  (Fig. 10). Note that  $\det(P) = 6$ , which is the maximum number of iterations within one tile, as can be seen on Fig. 2.

A tiling  $H$  is defined by the number of hyperplane sets used, by the directions of the normal vectors  $h_i$  and by their relative norms, that is, the tile shape, and by the number of iterations within a tile, that is, the tile size. The tile shape is defined when  $\det(H) = 1$ . The tile size is controlled by a scaling coefficient. The tiling origin is another parameter impacting mostly the code generation, but also the execution time. Often, a

Tiling. Fig. 10 Hyperplane matrix  $H$ 

tiling selection is decomposed into the selection of a shape and the selection of a size and finally the choice of an offset.

The legality conditions are summed up by  $H^T R \geq 0$ , where  $R$  is a matrix made of the rays of a convex cone containing all possible dependence vectors  $d$ , because the condition  $h.d \geq 0$  is convex. The computation of  $R$  by a dependence test is explained by Irigoin and Triolet [21] and the dependence cone is one of many approximations of the dependence vector set. When dependences are uniform (see ▶Dependences),  $R$  can be built directly with the dependence vectors. The valid hyperplanes  $h$  belong to another cone, dual of  $R$ .

A necessary and sufficient condition,  $[H^T d] \geq 0$ , was introduced by Xue [35] in 1997, where  $\geq$  is the lexicographic order, but it does not bring any practical improvement over the previous sufficient condition. Xue also provided an exact legality test based on integer programming and using information about the iteration set  $L(j)$ , that is, the loop bounds of the initial loop nest.

Note also that the subset of tilings such that  $\det(H) = 1$  is the set of unimodular transformations and that  $H^T d \geq 0$  is their legality condition (see ▶Loop Nest Parallelization).

## Tile Selection and Optimal Tiling

Tile selection requires some choice criterion, and optimal tile selection some cost function. The cost function is obviously dependent on the target machine, which makes many optimal tilings possible since many target architectures can benefit from tiling. Also, if the cost function is the execution time, the tiling per se is only part of the compilation scheme. The execution time of one tile depends on the scheduling of the local iterations, for instance because each processor has some

vector capability. It also depends on the tiles previously executed on the same processor, whether a cache or a local memory is used, that is, it depends on the *mapping* of tiles on processor. And finally, the total execution time of the tiled nest depends on the schedule and on the mapping of the tiles on the logical or physical resources, threads, or cores.

In other words, any optimal tiling is optimal with respect to a cost function modeling the execution time or the energy for a given target. Models used to derive analytical optimal solutions often assume that the execution and the communication times are respectively proportional to the computation and communication volumes, which is not realistic, especially with multicore, superword parallelism, and several levels of cache memories. Models may also assume that the number of processors available (because of multicore and multi-threaded architectures, the definition of *processor*, virtual or physical, is not well defined. Here, the processor is not a chip, but rather the total number of physical threads in a multicore or the total number of user processes running simultaneously in the machine) is greater than the number of tiles that can be executed simultaneously, which simplifies the mapping of tiles on processors.

When the target architecture has some kind of implicit vector capability, for instance when the cache lines are loaded, a partial tiling with  $\text{rank}(H) < n$ , that is, a set of hyperplane partitionings, may be more effective than a full tiling. Tiles are not longer bounded by the hyperplanes, but they remain bounded by the initial loop nest iteration set.

Because the execution time of real machine becomes more and more complex with the number of transistors used, iterative compilation is used when performance is key. The code is compiled and executed with different tile sizes and the best tile size is retained. Symbolic tiling is useful to speed up the process.

Some decisions can be made at run time. For instance, Rastello et al. [25] use run-time scheduling to speed-up the execution. Note that they overlap the computations and the communications related to *one* tile, which somehow breaks the tile atomicity constraint.

## Tiled Code Generation

As for tiling optimality, tile code generation depends on the target machine. A parallel machine requires the

mapping and the parallel execution of the tiles. A distributed memory machine also requires communication generation. A processor with a memory hierarchy and/or a vector capability requires loop optimization at the tile level. The minimal requirement is that all iterations of the initial loop nest are performed by the tiled nest.

Let vector  $j$  be an iteration of loop nest  $L$ ,  $t$  a tile coordinate, and  $l$  the local coordinate of an iteration within a tile  $t$ . Since no redundant computations are added by hyperplane partitioning, the relationship between  $j$  and  $(t, l)$  is a one-to-one mapping from the initial set of iterations  $L(j)$  to the new iteration set  $T(t, l)$ . Ancourt and Irigoin [5] show that an affine relationship can be built between  $j$  and  $(t, l)$  and that the new loop bounds for  $t$  and  $l$  can be derived from this relationship and from  $L$  when the matrix  $H$  is numerically known and when  $L$  is a parametric polyhedron, that is, when the loop bounds are affine functions of loop indices and parameters. To simplify array subscript expressions, the code may be generated using  $t$  and  $j$  instead of  $t$  and  $l$ . This optimization is used in the code examples given above.

This loop nest generation is sufficient for shared memory machines, although it is better to generate several versions of the tile code, one for the full tiles, and several ones for the partial tiles on the iteration set boundaries, in order to reduce the average control overhead. Multilevel tiling is also used to reduce the overhead due to partial tiles on the boundaries (see the top left and top right tiles in Fig. 2).

This does not specify the mapping of tiles onto threads or cores when the tile parallelism is greater than the number of processors. But locality-aware scheduling lets tiles inherit data from other tiles previously executed on the same thread as suggested by Xue and Huang in [37] who minimize the number of partitioning hyperplanes, that is, the rank of  $H$ .

Parametric tiling does not require  $H$  to be numerically known at compile time. The tile size, if not the tile shape, can be adjusted at run time or optimized dynamically. A technique is proposed by Renganarayanna et al. [28] for multilevel tiling, and another one by Hartono et al. [18, 23].

Note that parallelism within tiles or across tiles is obtained by *wavefronting*, a unimodular loop transformation (see ▶Loop Nest Parallelization), unless the initial loop nest is fully parallel, in which case cone  $R$  is

empty or reduced to  $\{0\}$ . For instance, the tiles  $(1, 0)$  and  $(1, 0)$  on Fig. 3 can be computed in parallel.

## Applicability

Tiling and hyperplane partitioning are defined for perfectly nested loops only, but many algorithms, including matrix multiply, are made of non-perfectly nested loops. It is possible to move all non-perfectly nested statements into the loop nest body by adding guards (a.k.a. statement sinking), but these guards must then be carefully moved or removed when the tile code is generated. The issue is tackled directly by Ahmed et al. [3] and Griebl [15, 16], who avoid statement sinking by mapping all statement in another space (see ► [Polyhedron Model](#)) and by applying transformations, including tiling, on this space before code generation, and by Hartono et al. [18] who use a polyhedral representation of the code to generate multilevel tilings of imperfectly nested loops. See the Polyhedron Model entry for more information about the mapping of a piece of code onto a polyhedral space.

Tiling can also be applied to loop nests containing commutative and associative reductions, but this does not fit the general legality condition  $HR \geq 0$ .

Tiling can be applied by the programmer. For instance, 3-D tiling improved with array padding has been used to optimize 3-D PDE solvers and 3-D stencil codes. Tiling has been used to optimize some instances of dynamic programming, the resolution of the heat equation, and even some sparse computations. Because tiling is difficult to apply by hand, source-to-source tilers have been developed.

Finally, Guo et al. suggest in [17] to support tiling at the programming language level, using hierarchically tiled arrays (HTA) to keep the code readable, while letting the programmer be in control.

## Related Loop Transformations

The partitioning matrix  $P = (H^T)^{-1}$  can be built step-by-step by a combination of loop skewing, or more generally any unimodular loop transformations, strip-minings (1-D tiling), and loop interchanges. Loop skewing is a unimodular transformation used to change the iteration coordinates and to make loop blocking legal because the new loop nest obtained is *fully permutable*.

In other words, the  $P$  matrix is replaced by the product of a diagonal matrix  $\Lambda$ , which defines a rectangular tiling, a.k.a. loop blocking, and of  $\frac{1}{\det(P)}P$ , and the tiling by a sequence of easier transformations. This is advocated by Allen & Kennedy in their textbook [4]. Reducing tiling to blocking via basis changes, for example, using a Smith normal form of  $H$ , is also often used to optimize the tile shape and size, but some of the tile shape problem remains and the constraints of the iteration set  $L$  usually become more complex.

Strip-mining is a 1-D tiling, a degenerated case of hyperplane partitioning. It is used to adapt the parallelism available to the hardware resources, for instance vector registers.

Loop interchange is a unimodular transformation. Like all unimodular transformations, it is an extreme case of tiling with no tiling effect because  $\det(H) = 1$ , that is, each tile contains only one element. It is often used to increase locality.

Loop unroll-and-jam first unrolls an outer loop by some factor  $k$ , that is, it is a strip-mining followed by a full unroll of the new loop. Then, the replicated innermost loops are fused (jammed). This is equivalent to a rectangular hyperplane partitioning with blocking factors  $(k, 1, 1, \dots)$ , followed by an unrolling of the tile loop. Unroll-and-jam can be applied to several outer loops with several factors, which again is equivalent to a rectangular tiling with the same factors followed by an unrolling of the tile loops. Unroll-and-jam is used to increase locality and is effective like tiling if some references exhibit temporal locality along outer loops.

Tiling is designed to forbid redundant computations. However, overlap between tiles can reduce communications at the expense of additional computation. Data overlaps are also used to compile ► [HPF \(High-Performance Fortran\)](#) using the owner compute rule.

Finally, tiling is also related to the partitioning of ► [systolic arrays](#), used to fit a large parametric size iteration set on a fixed-size chip.

## Future Directions

Although tiling is a powerful transformation by itself, and quite complex to use, it does not include some other key transformations such as loop fusion or loop peeling. Furthermore, the loop body is handled as a unique statement although it may contain sequences, tests, and loops.

So more complex code transformations were advocated in 1991 by Wolf and Lam [31] to optimize parallelism and locality. More recently, Griebl [15] and Bondhugula et al. [8] use the polyhedral framework (see ▶ [Polyhedron Model](#)) to handle each elementary statement individually, at least within static control pieces of code. This is also attempted within gcc with the Graphite plug-in.

Otherwise, it is possible to move away from the complexity of tiling by replacing it with sequences of simpler transformations, including hyperplane partitioning. The difficulty is then to decide which sequence leads to an optimal or at least to a well-performing code.

In case the execution time of each iteration is different or even very different, the tile equality constraint could be lifted up to obtain a nonuniform partitioning. This has already been done in the 1-D case. In such cases, strip-mining is replaced by more complex partitions to map the parallel iterations onto the processors. The larger partitions are executed first to reduce the imbalance between processors at the end without increasing the control overhead at the beginning (see ▶ [Nested loops scheduling](#)).

## Related Entries

- ▶ [Code Generation](#)
- ▶ [Dependences](#)
- ▶ [Dependence Abstractions](#)
- ▶ [Dependence Analysis](#)
- ▶ [Distributed-Memory Multiprocessor](#)
- ▶ [HPF \(High Performance Fortran\)](#)
- ▶ [Locality of Reference and Parallel Processing](#)
- ▶ [Loop Nest Parallelization](#)
- ▶ [Parallelization, Automatic](#)
- ▶ [Polyhedron Model](#)
- ▶ [Shared-Memory Multiprocessors](#)
- ▶ [Systolic Arrays](#)
- ▶ [Unimodular Transformations](#)

## Bibliographic Notes and Further Reading

The best reference about tiling is the book written by Xue [35]. It provides the necessary background on linear algebra and program transformations. It starts with rectangular tilings before moving to slanted, that is, parallelepiped, tilings. Code generation for distributed

memory machines and tiling optimizations are finally addressed.

Tile size optimization is addressed explicitly by Coleman and McKinley [12] to eliminate cache capacity, self-interference, and cross-interference misses, that is, for locality improvement.

For shared memory machines, Högstedt et al. [20] introduce the concepts of *idle time* and *rise* for a tiling, and optimal tile shape for a multiprocessor with a memory hierarchy. They propose an algorithm to select an non-rectangular optimal tile shape for a shared memory multiprocessor, with enough processors to use all parallelism available. They assume that the tile execution time is proportional to its volume. Rastello and Robert [26] provide a closed form for the tile shape that minimizes the number of cache misses during the execution of a rectangular tile for a given cache size and for parallel loops, that is, without tiling legality constraints. They also provide a heuristic to optimize the same function for any shape of tiles.

For distributed memory machines, the contributions to the quest for analytic solutions are numerous. Boulet et al. [10] carefully include a question mark in their paper title, (*Pen*)-ultimate tiling?. Hodzic and Shang give several closed forms for the size and the relative side lengths [19]. Xue [36] uses communication/computation overlap and provides a closed form for the optimal tile size.

Tile shapes are restrained to orthogonal shapes by Andonov et al. [7] in order to find an optimal solution. For 2-D iteration spaces, using the BSP model and an unbounded number of processors, Andonov et al. [6] provide closed forms for the optimal tiling parameters and the optimal number of processors.

Agarwal et al. [1, 2] introduce a method for deriving an optimal hyperparallelepiped tiling of iteration spaces for minimal communication in multiprocessors with caches and for distributed shared-memory multiprocessors. More recently, Bondhugula et al. [9] tile sequences of imperfectly nested loops for locality and parallelism. They use an analytical model and integer linear optimization.

Carter et al. introduce hierarchical tiling for superscalar machines [11], but the tuning is handmade. Renganarayanna and Rajopadhye [27] determine optimal tile sizes with a BSP-like model. Strzodka et al. [29] use multilevel tiling to speed up stencil computations

by optimizing simultaneously locality, parallelism and vectorization.

For distributed memory machines, communication code must be generated too. See Ancourt [5], Tang [30], Xue [35], Chapter 6 and 7, and finally Goumas et al. [14], who generate MPI code automatically.

Goumas et al. propose [13] a tile code generation algorithm for parallelepiped tiles. This can be used for general tiles thanks to changes of basis.

## Bibliography

1. Agarwal A, Kranz D, Natarajan V (1993) Automatic partitioning of parallel loops for cache-coherent multiprocessors. In: International conference on parallel processing (ICPP), Syracuse University, Syracuse, NY, 16–20 August 1993, vol 1, pp 2–11
2. Agarwal A, Kranz DA, Natarajan V (September 1995) Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors. *IEEE Trans Parallel Distrib Syst* 6(9):943–962
3. Ahmed N, Mateev N, Pingali K (2000) Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In: Proceedings of the 14th international conference on supercomputing, Santa Fe, 8–11 May 2000, pp 141–152
4. Allen R, Kennedy K (2002) Optimizing compilers for modern architectures: a dependence-based approach. Morgan-Kaufmann. San Francisco, pp 477–491
5. Ancourt C, Irigoin F (1991) Scanning polyhedra with DO loops. In: Third ACM symposium on principles and practice of parallel programming, Williamsburg, VA, pp 39–50
6. Andonov R, Balev S, Rajopadhye S, Yanev N (July 2001) Optimal semi-oblique tiling. In: Proceedings of the 13th annual ACM symposium on parallel algorithms and architectures, Crete Island, pp 153–162
7. Andonov R, Rajopadhye SV, Yanev N (1998) Optimal orthogonal tiling. In: Proceedings of the fourth international Euro-Par conference on parallel processing, Southampton, 1–4 Sept 1998, pp 480–490
8. Bondhugula U, Baskaran M, Krishnamoorthy S, Ramanujam J, Rountev A, Sadayappan P (2008) Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In: Proceedings of the joint European conferences on theory and practice of software 17th international conference on compiler construction, Budapest, Hungary, 29 March–6 April 2008
9. Bondhugula U, Hartono A, Ramanujam J, Sadayappan P (June 2008) A practical automatic polyhedral parallelizer and locality optimizer. In: PLDI 2008. ACM SIGPLAN Not 43(6)
10. Boulet P, Darte A, Risset T, Robert Y (1996) (Pen)-ultimate tiling? *Integr: VLSI J* 17:33–51
11. Carter L, Ferrante J, Hummel SF (1995) Hierarchical tiling for improved superscalar performance. In: Proceedings of the ninth international symposium on parallel processing, Santa Barbara, 25–28 April 1995, pp 239–245
12. Coleman S, McKinley KS (June 1995) Tile size selection using cache organization and data layout. In: PLDI'95; ACM SIGPLAN Not 30(6):279–290
13. Goumas G, Athanasaki M, Koziris N (2002) Automatic code generation for executing tiled nested loops onto parallel architectures. In: Proceedings of the 2002 ACM symposium on applied computing, Madrid, Spain, 11–14 March 2002
14. Goumas G, Drosinos N, Athanasaki M, Koziris N (November 2006) Message-passing code generation for non-rectangular tiling transformations. *Parallel Computing* 32(10): 711–732
15. Griebel M (July 2001) On tiling space-time mapped loop nests. In: Proceedings of the 13th annual ACM symposium on parallel algorithms and architectures, Crete Island, pp 322–323
16. Griebel M (June 2004) Automatic parallelization of loop programs for distributed memory architectures. Habilitation thesis, Department of Informatics and Mathematics, University of Passau. <http://www.fim.uni-passau.de/cl/publications/docs/Gri04.pdf>
17. Guo J, Bikshandi G, Fraguerau BB, Garzaran MJ, Padua D (2008) Programming with tiles. In: Proceedings of the 13th ACM SIGPLAN symposium on principles and practice of parallel programming, Salt Lake City, UT, USA, 20–23 Feb 2008
18. Hartono A, Manikandan Baskaran M, Bastoul C, Cohen A, Krishnamoorthy S, Norris B, Ramanujam J, Sadayappan P (2009) Parametric multi-level tiling of imperfectly nested loops. In: Proceedings of the 23rd international conference on supercomputing, Yorktown Heights, NY, USA, 8–12 June 2009
19. Hodzic E, Shang W (December 2002) On time optimal supernode shape. *IEEE Trans Parallel Distrib Syst* 13(12):1220–1233
20. Högstedt K, Carter L, Ferrante J (March 2003) On the parallel execution time of tiled loops. *IEEE Trans Parallel Distrib Syst* 14(3):307–321
21. Irigoin F, Triolet R (1988) Supernode partitioning. In: Fifteenth annual ACM symposium on principles of programming languages, San Diego, CA, pp 319–329
22. Jiménez M, Llaberia JM, Fernández A (July 2002) Register tiling in nonrectangular iteration spaces. *ACM Trans Program Lang Syst* 24(4):409–453
23. Manikandan Baskaran M, Hartono A, Tavarageri S, Henretty T, Ramanujam J, Sadayappan P (2010) Parameterized tiling revisited. In: CGO'10: proceedings of the eighth annual IEEE/ACM international symposium on code generation and optimization, pp 200–209
24. McKeller AC, Coffman EG (1969) The organization of matrices and matrix operations in a paged multiprogramming environment. *Commun ACM* 12(3):153–165
25. Rastello F, Rao A, Pande S (February 2003) Optimal task scheduling at run time to exploit intra-tile parallelism. *Parallel Comput* 29(2):209–239
26. Rastello F, Robert Y (May 2002) Automatic partitioning of parallel loops with parallelepiped-shaped tiles. *IEEE Trans Parallel Distrib Syst* 13(5):460–470
27. Renganarayana L, Rajopadhye S (2004) A geometric programming framework for optimal multi-level tiling. In: Proceedings of the 2004 ACM/IEEE conference on supercomputing, Pittsburgh, PA, 6–12 Nov 2004, p 18

28. Renganarayanan L, Kim D, Rajopadhye S, Strout MM (June 2007) Parameterized tiled loops for free. In: PLDI'07, ACM SIGPLAN Not 42(6)
29. Strzodka R, Shaheen M, Pajak D, Seidel H-P (2010) Cache oblivious parallelograms in iterative stencil computations. In: ICS'10: proceedings of the 24th ACM international conference on supercomputing, Tsukuba, Japan, pp 49–59
30. Tang P, Xue J (2000) Generating efficient tiled code for distributed memory machines. Parallel Comput 26(II):1369–1410
31. Wolf ME, Lam MS (October 1991) A loop transformation theory and an algorithm to maximize parallelism. IEEE Trans Parallel Distrib Syst 2(4):452–471
32. Wolfe MJ (1987) Iteration space tiling for memory hierarchies. In: Rodrigue G (ed) Parallel processing for scientific computing. SIAM, Philadelphia, pp 357–361
33. Wolfe MJ (1989) More iteration space tiling. In: Proceedings of the 1989 ACM/IEEE conference on supercomputing, Reno, NV, 12–17 Nov 1989, pp 655–664
34. Wolfe MJ (1995) High performance compilers for parallel computing. Addison-Wesley Longman, Boston
35. Xue J (2000) Loop tiling for parallelism. Kluwer, Boston
36. Xue J, Cai W (June 2002) Time-minimal tiling when rise is larger than zero. Parallel Comput 28(6):915–939
37. Xue J, Huang C-H (December 1998) Reuse-driven tiling for improving data locality. Int J Parallel Program 26(6):671–696

## Titanium

KATHERINE YELICK<sup>1</sup>, SUSAN L. GRAHAM<sup>2</sup>, PAUL HILFINGER<sup>2</sup>, DAN BONACHEA<sup>3</sup>, JIMMY SU<sup>2</sup>, AMIR KAMIL<sup>2</sup>, KAUSHIK DATTA<sup>2</sup>, PHILLIP COLELLA<sup>2</sup>, TONG WEN<sup>2</sup>

<sup>1</sup>University of California at Berkeley and Lawrence Berkeley National Laboratory, Berkeley, CA, USA

<sup>2</sup>University of California, Berkeley, CA, USA

<sup>3</sup>Lawrence Berkeley National Laboratory, Berkeley, CA, USA

### Definition

Titanium is a parallel programming language designed for high-performance scientific computing. It is based on Java™ and uses a Single Program Multiple Data (SPMD) parallelism model with a Partitioned Global Address Space (PGAS).

### Discussion

#### Introduction

Titanium is an explicitly parallel dialect of Java™ designed for high-performance scientific programming

[14, 15]. The Titanium project started in 1995, at a time when custom supercomputers were losing market share to PC clusters. The motivation was to create a language design and implementation that would enable portable programming for a wide range of parallel platforms by striking an appropriate balance between expressiveness, user-provided information about concurrency and memory locality, and compiler and runtime support for parallelism. The goal was to design a language that could be used for high performance on some of the most challenging applications, such as those with adaptivity in time and space, unpredictable dependencies, and sparse, hierarchical, or pointer-based data structures.

The strategy was to build on the experience of several Partitioned Global Address Space (PGAS) languages, but to design a higher-level language offering object orientation with strong typing and safe memory management in the context of applications requiring high performance and scalable parallelism. Titanium uses Java as the underlying base language, but is neither a strict superset nor subset of that language. Titanium adds general multidimensional arrays, support for extending the value types in the language, and an unordered loop construct. In place of Java threads, which are used for both program structuring and concurrency, Titanium uses a static thread model with a partitioned address space to allow for locality optimizations.

#### Titanium's Parallelism Model

Titanium uses a Single Program Multiple Data (SPMD) parallelism model, which is familiar to users of message-passing models. The following simple Titanium program illustrates the use of built-in methods `Ti.numProcs()` and `Ti.thisProc()`, which query the environment for the number of threads (or processes) and the index within that set of the executing thread. The example prints these indices in arbitrary order. The number of Titanium threads need not be equal to the number of physical processors, a feature that is often useful when debugging parallel code on single-processor machines. However, high-performance runs typically use a one-to-one mapping between Titanium threads and physical processors.

```
class HelloWorld {
 public static void main (String [] argv) {
```

```

 System.out.println("Hello from proc " +
 Ti.thisProc() + " out of " + Ti.numProcs());
 }
}

```

Titanium supports Java's synchronized blocks, which are useful for protecting asynchronous accesses to shared objects. Because many scientific applications use a bulk-synchronous style, Titanium also has a barrier-synchronization construct, `Ti.barrier()`, as well as a set of collective communication operations to perform broadcasts, reductions, and scans. A novel feature of Titanium's parallel execution model is that barriers must be textually aligned in the program – not only must all threads reach a barrier before any one of them may proceed, but they must all reach the same textual barrier. For example, the following program is not legal in Titanium:

```

if (Ti.thisProc() == 0) Ti.barrier();
//illegal barrier
else Ti.barrier(); //illegal barrier

```

Aiken and Gay developed the static analysis the compiler uses to enforce this alignment restriction, based on two key concepts [1]:

- A *single method* is one that must be invoked by all threads collectively. Only single methods can execute barriers.
- A *single-valued expression* is an expression that is guaranteed to take on the same sequence of values on all processes. Only single-valued expressions may be used in conditional expressions that affect which barriers or single-method calls get executed.

The compiler automatically determines which methods are single by finding barriers or (transitively) calls to other single methods. Single-valued expressions are required in statements that determine the flow of control to barriers, ensuring that the barriers are executed by all threads or by none. Titanium extends the Java type system with the single qualifier. Variables of single-qualified type may only be assigned values from single-valued expressions. Literals and values that have been broadcast are simple examples of single-valued expressions. The following example illustrates these concepts. Because the loop contains barriers, the expressions in the for-loop header must be single-valued. The compiler can check that property statically, since the variables

are declared single and are assigned from single-valued expressions.

```

int single allTimestep = 0;
int single allEndTime = broadcast
 inputTimeSteps from 0;
for (; allTimestep < allEndTime;
 allTimestep)++{
 < read values belonging to other threads >
 Ti.barrier();
 < compute new local values >
 Ti.barrier();
}

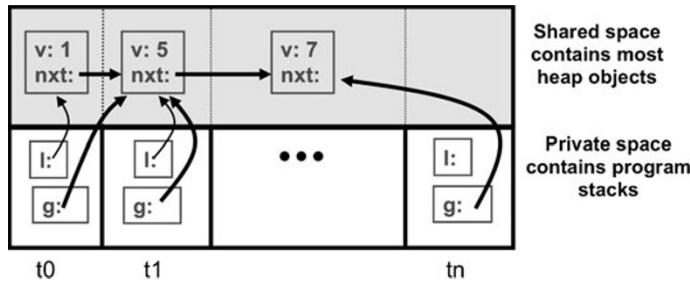
```

Barrier analysis is entirely static and provides compile-time prevention of barrier-based deadlocks. It can also be used to improve the quality of concurrency analysis used in optimizations. Single qualification on variables and methods is a useful form of program design documentation, improving readability by making replicated quantities and collective methods explicitly visible in the program source and subjecting these properties to compiler enforcement.

## Titanium's Memory Model

The two basic mechanisms for communicating between threads are accessing shared variables and sending messages. Shared memory is generally considered easier to program, because communication is one-sided: Threads can access shared data at any time without interrupting other threads, and shared data structures can be directly represented in memory. Titanium is based on a Partitioned Global Address Space (PGAS) model, which is similar to shared memory but with an explicit recognition that access time is not uniform. As shown in Fig. 1, memory is partitioned such that each partition has affinity to one thread. Memory is also partitioned orthogonally into private and shared memory, with stack variables living in private memory, and heap objects, by default, living in the shared space. A thread may access any variable that resides in shared space, but has fast access to variables in its own partition. Objects created by a given thread will reside in its own part of the memory space.

Titanium statically makes an explicit distinction between local and global references: A local reference must refer to an object within the same thread partition, while a global reference may refer to either a remote or



Titanium. Fig. 1 Titanium’s partitioned global address space memory model

local partition. In Fig. 1, instances of `l` are local references, whereas `g` and `nxt` are global references and can therefore cross partition boundaries. The motivation for this distinction is performance. Global references are more general than local ones, but they often incur a space penalty to store affinity information and a time penalty upon dereference to check whether communication is required. References in Titanium are global by default, but may be designated local using the `local` type qualifier. The compiler performs type inference to automatically label variables as local [10].

The partitioned memory model is designed to scale well on distributed memory platforms without the need for caching of remote data and the associated coherence protocols. Titanium also runs well on shared memory multiprocessors and uniprocessors, where the partitioned-memory model may not correspond to any physical locality on the machine and the global references generally incur no overhead relative to local ones. Naively written Titanium programs may ignore the partitioned-memory model and, for example, allocate all data structures in one thread’s shared memory partition or perform fine-grained accesses on remote data. Such programs would run correctly on any platform but would likely perform poorly on a distributed memory platform. In contrast, a program that carefully manages its data-structure partitioning and access behavior in order to scale well on distributed memory hardware is likely to scale well on shared memory platforms as well. The partitioned model provides the ability to start with a functional, shared memory style code and incrementally tune performance for distributed memory hardware by reorganizing the affinity of key data structures or adjusting access patterns in program bottlenecks to improve communication performance.

## Titanium Arrays

Java arrays do not support sub-array objects that are shared with larger arrays, nonzero base indices, or true multidimensional arrays. Titanium retains Java arrays for compatibility, but adds its own multidimensional array support, which provides the same kinds of sub-array operations available in Fortran 90. Titanium arrays are indexed by integer tuples known as points and built on sets of points, called domains. The design is taken from that of a language for Finite Difference Calculations, FIDIL, designed by Colella and Hilfinger [7]. Points and domains are first-class entities in Titanium – they can be stored in data structures, specified as literals, passed as values to methods, and manipulated using their own set of operations. For example, NAS multigrid (MG) benchmark requires a  $256^3$  grid. The problem has periodic boundaries, which are implemented using a one-deep layer of surrounding ghost cells, resulting in a  $258^3$  grid. Such a grid can be constructed with the following declaration:

```
double [3d] gridA
= new double [[-1,-1,-1]:[256,256,256]];
```

The 3D Titanium array `gridA` has a rectangular index set that consists of all points  $[i, j, k]$  with integer coordinates such that  $-1 \leq i, j, k \leq 256$ . Titanium calls such an index set a rectangular domain of Titanium type `RectDomain`, since all the points lie within a rectangular box. Titanium also has a type `Domain` that represents an arbitrary set of points, but Titanium arrays can only be built over `RectDomains`. Titanium arrays may start at an arbitrary base point, as the example with a  $[-1, -1, -1]$  base shows. In this example, the grid was designed to have space for ghost regions, which are

all the points that have either -1 or 256 as a coordinate. On machines with hierarchical memory systems, **gridA** resides in memory with affinity to exactly one process, namely the process that executes the above statement. Similarly, objects reside in a single logical memory space for their entire lifetime (there is no transparent migration of data), though they are accessible from any process in the parallel program.

The power of Titanium arrays stems from array operators that can be used to create alternative views of an array's data, without an implied copy of the data. While this is useful in many scientific codes, it is especially valuable in hierarchical grid algorithms like Multigrid and Adaptive Mesh Refinement (AMR). In a Multigrid computation on a regular mesh, there is a set of grids at various levels of refinement, and the primary computations involve sweeping over a given level of the mesh performing nearest neighbor computations (called stencils) on each point. To simplify programming, it is common to separate the interior computation from computation at the boundary of the mesh, whether those boundaries come from partitioning the mesh for parallelism or from special cases used at the physical edges of the computational domain. Since these algorithms typically deal with many kinds of boundary operations, the ability to name and operate on sub-arrays is useful.

## Domain Calculus

Titanium's domain calculus operators support sub-arrays both syntactically and from a performance standpoint. The tedious business of index calculations and array offsets has been migrated from the application code to the compiler and runtime system. For example, the following Titanium code creates two blocks that are logically adjacent, with a boundary of ghost cells around each to hold values from the adjacent block. The shrink operation creates a view of **gridA** by shrinking its domain on all sides, but does not copy any of its elements. Thus, **gridAInterior** will have indices from [0, 0, 0] to [255, 255, 255] and will share corresponding elements with **gridA**. The **copy** operation in the last line updates one plane of the ghost region in **gridB** by copying only those elements in the intersection of the two arrays. Operations on Titanium arrays such as **copy** are not opaque method calls to the Titanium compiler.

The compiler recognizes and treats such operations specially, and thus can apply optimizations to them, such as turning blocking operations into non-blocking ones.

```
double [3d] gridA =
 new double [[-1,-1,-1]:[256,256,256]];
double [3d] gridB =
 new double [[-1,-1,256]:[256,256,512]];
//define interior without creating a copy
double [3d] gridAInterior = gridA.shrink(1);
//update overlapping ghost cells
 from neighboring block
//by copying values from gridA to gridB
gridB.copy(gridAInterior);
```

The above example appears in a NAS MG implementation in Titanium [4], except that **gridA** and **gridB** are themselves elements of a higher-level array structure. The copy operation as it appears here performs contiguous or noncontiguous memory copies, and may perform interprocessor communication when the two grids reside in different processor memory spaces. The use of a global index space across distinct array objects (made possible by the arbitrary index bounds of Titanium arrays) makes it easy to select and copy the cells in the ghost region, and is also used in the more general case of adaptive meshes.

## Unordered Loops, Value Types, and Overloading

The **foreach** construct provides an unordered looping construct designed for iterating through a multidimensional space. In the foreach loop below, the point **p** plays the role of a loop index variable.

```
foreach (p in gridAInterior.domain()) {
 gridB[p] = applyStencil(gridAInterior, p);
}
```

The **applyStencil** method may safely refer to elements that are one point away from **p**, since the loop is over the interior of a larger array.

This one loop concisely expresses an iteration over a multidimensional domain that would correspond to a multi-level loop nest in other languages. A common class of loop bounds and indexing errors is avoided by having the compiler and runtime system automatically manage the iteration boundaries for the multidimensional traversal. The foreach loop is a purely serial iteration construct – it is not a data-parallel construct. In addition, if the order of loop execution is irrelevant to a computation, then using a foreach loop

to traverse the points in a domain explicitly allows the compiler to reorder loop iterations to maximize performance – for instance, by performing automatic cache blocking and tiling optimizations [12]. It also simplifies bounds-checking elimination and array access strength-reduction optimizations.

The Titanium immutable class feature provides language support for defining application-specific primitive types (often called “lightweight” or “value” classes), allowing the creation of user-defined unboxed objects, analogous to C structs. Immutables provide efficient support for extending the language with new types which are manipulated and passed by value, avoiding pointer-chasing overheads which would otherwise be associated with the use of tiny objects in Java.

One compelling example of the use of immutables is for defining a Complex number class, which was used in a Titanium implementation of the NAS FT benchmark.

Titanium also allows for operator overloading, a feature that was strongly desired by application developers on the team, and was used in the FT example to simplify the expressions on complex values.

## Distributed Arrays

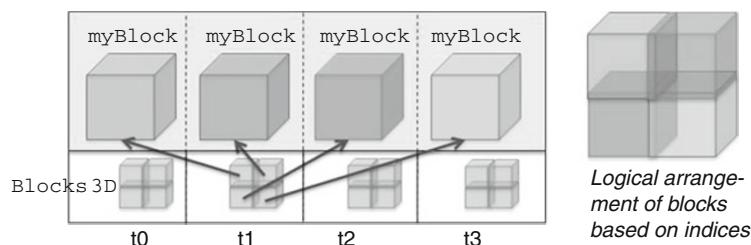
Titanium also supports the construction of distributed array data structures, which are built from local pieces rather than declared as distributed types. This reflects the design emphasis on adaptive and sparse data structures in Titanium, rather than the simpler “regular array” computations that could be supported with simpler flat arrays. The general pointer-based distribution mechanism combined with the use of arbitrary base indices for arrays provides an elegant and powerful mechanism for shared data.

The following code is a portion of the parallel Titanium code for a multigrid computation. It is run on

every processor and creates the **blocks3D distributed** array, which can access any processor’s portion of the grid.

```
Point< 3 > startCell =
myBlockPos * numCellsPerBlockSide;
Point< 3 > endCell = startCell + (numCellsPerBlock
Side - [1,1,1]);
double [3d] myBlock =
new double [startCell:endCell];
// "blocks" is used to create "blocks3D" array
double [1d] single [3d] blocks =
new double [0:(Ti.numProcs()-1)] single [3d];
blocks.exchange(myBlock);
// create local "blocks3D" array
double [1d] single [3d] blocks3D =
new double [[0,0,0]:numBlocksInGridSide -
[1,1,1]] single [3d];
// map from "blocks" to "blocks3D" array
foreach (p in blocks3D.domain())
blocks3D[p] = blocks[procForBlockPosition(p)];
```

Each processor computes its start and end indices by performing arithmetic operations on **Points**. These indices are used to create a local **myBlock** array. Every processor also allocates its own 1D array **blocks**. Then, by combining the **myBlock** arrays using the exchange operation, **blocks** becomes a distributed data structure. As shown in Fig. 2, the exchange operation performs an all-to-all broadcast and stores each processor’s contribution in the corresponding element of its local blocks array. To create a more natural mapping, a 3D processor array is used, with each element containing a reference to a particular local block. By using global indices in the local block – meaning that each block has a different set of indices that overlap only in the area of ghost regions – the copy operations described above can be used to update the ghost cells. The generality of Titanium’s distributed data structures is not fully utilized in the example of a uniform mesh, but in an adaptive block structured mesh, a union of rectangles can be used to



**Titanium. Fig. 2** Distributed 3D array in titanium’s PGAS address space. The pointers in the blocks3D array are shown only for thread t1 for simplicity

fill a spatial area, and the global indexing and global address space used to simplify much more complicated ghost region updates.

## Implementation Techniques and Research

The Titanium compiler translates Titanium code into C code, and then hands that code off to a C compiler to be compiled and linked with the Titanium runtime system and, in the case of distributed memory back ends, with the GASNet communication system [5]. The choice of C as a target was made to achieve portability, and produces reasonable performance without the overhead of a virtual machine. GASNet is a one-sided communication library that is used within a number of other PGAS language implementations, including Co-Array Fortran, Chapel, and multiple UPC implementations. GASNet is itself designed for portability, and it runs on top of Ethernet (UDP) and MPI, but there are optimized implementations for most of the high-speed networks that are used in clusters and supercomputers designs. Titanium can also run on shared memory systems using a runtime layer based on POSIX Threads, and on combinations of shared and distributed memory by combining this with GASNet. Titanium, like Java, is designed for memory safety, and the Titanium runtime system includes the Boehm-Weiser garbage collector for shared memory code. To handle distributed memory environments, the runtime system tracks references that leak to remote nodes, but also adds a scalable region-based memory management concept to the language along with compiler analysis [5].

Aggressive program analysis is crucial for effective optimization of parallel code. In addition to serial loop optimizations [12] and some standard optimizations to reduce the size and complexity of generate C code, the compiler performs a number of novel analyses on parallelism constructs. For example, information about what sections of code may operate concurrently is useful for many optimizations and program analyses. In combination with alias analysis, it allows the detection of potentially erroneous race conditions, the removal of unnecessary synchronization operations, and the ability to provide stronger memory consistency guarantees. Titanium's textually aligned barriers divide the code into independent phases, which can be exploited to improve the quality of concurrency analysis. The single-valued

expressions are also used to improve concurrency analysis on branches. These two features allow a simple graph encoding of the concurrency in a program based on its control-flow graph. We have developed quadratic-time algorithms that can be applied to the graph in order to determine all pairs of expressions that can run concurrently.

Alias analysis identifies pointer variables that may, must, or cannot reference the same object. The Titanium compiler uses alias analysis to enable other analyses (such as locality and sharing analysis), and to find places where it is valid to introduce restrict qualifiers in the generated C code, enabling the C compiler to apply more aggressive optimizations. The Titanium compiler's alias analysis is a Java derivative of Andersen's points-to analysis with extensions to handle multiple threads. The modified analysis is only a constant factor slower than the sequential analysis, and its running time is independent of the number of runtime threads.

## Application Experience

A number of benchmarks and larger applications have been written in Titanium, starting with some of the NAS Benchmarks [4]. In addition, Yau developed a distributed matrix library that supports blocked-cyclic layouts and implemented Cannon's Matrix Multiplication algorithm, Cholesky and LU factorization (without pivoting). Balls and Colella built a 2D version of their Method of Local Corrections algorithm for solving the Poisson equation for constant coefficients over an infinite domain [2]. Bonachea, Chapman, and Putnam built a Microarray Optimal Oligo Selection Engine for selecting optimal oligonucleotide sequences from an entire genome of simple organisms, to be used in microarray design. The most ambitious efforts have been applications frameworks for Adaptive Mesh Refinement (AMR) algorithms and Immersed Boundary Method simulations [6] by Tong Wen and Ed Giveland, respectively. In both cases, these application efforts have taken a few years and were preceded by implementations of Titanium codes for specific problem instances, e.g., AMR Poisson by Luigi Semenzato, AMR gas dynamics [11] by Peter McCorquodale and Immersed Boundaries for simulation of the heart by Armando Solar-Lezama and cochlea by Ed Giveland, with various optimization and analysis efforts by Sabrina Merchant, Jimmy Su, and Amir Kamil.

The performance results show good scalability on the applications problems on up to hundreds of separate distributed memory nodes, and performance that is in some cases comparable to applications written in C++ or FORTRAN with message passing. The compiler is a research prototype and does not have all of the static and dynamic optimizations one would expect from a commercial compiler, but even serial running-time comparisons show competitive performance. No formal productivity studies involving humans have been done, but a variety of case studies have shown that the global address space combined with a powerful multi-dimensional array abstraction and the data abstraction support derived from Java leads to code that is elegant and concise.

## Related Entries

- [Coarray Fortran](#)
- [PGAS \(Partitioned Global Address Space\) Languages](#)
- [UPC](#)

## Bibliography

1. Aiken A, Gay D (1998) Barrier inference. In: Principles of programming languages, San Diego, CA
2. Balls GT, Colella P (2002) A finite difference domain decomposition method using local corrections for the solution of Poisson's equation. *J Comput Phys* 180(1):25–53
3. Bonachea D (2002) GASNet specification. Technical report CSD-02-1207, University of California, Berkeley
4. Datta K, Bonachea D, Yelick K (2005) Titanium performance and potential: an NPB experimental study. In: 18th international workshop on languages and compilers for parallel computing (LCP). Hawthorne, NY, October 2005
5. Gay D, Aiken A (2001) Language support for regions. In: SIGPLAN conference on programming language design and implementation. Washington, DC, pp 70–80
6. Givelberg E, Yelick K Distributed immersed boundary simulation in titanium. <http://titanium.cs.berkeley.edu>, 2003
7. Hilfinger PN, Colella P (1989) FIDIL: a language for scientific processing. In: Grossman R (ed) Symbolic computation: applications to scientific computing. SIAM, Philadelphia, pp 97–138
8. Kamil A, Yelick K (2007) Hierarchical pointer analysis for distributed programs. Static Analysis Symposium (SAS), Kongens Lyngby, Denmark, August 22–24, 2007
9. Kamil A, Yelick K (2010) Enforcing textual alignment of collectives using dynamic checks. In: 22nd international workshop on languages and compilers for parallel computing (LCP), October 2009. Also appears in Lecture notes in computer science, vol 5898. Springer, Berlin, pp 368–382. DOI: 10.1007/978-3-642-13374-9

10. Liblit B, Aiken A (2000) Type systems for distributed data structures. In: The 27th ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL), Boston, January 2000
11. McCorquodale P, Colella P (1999) Implementation of a multi-level algorithm for gas dynamics in a high-performance Java dialect. In: International parallel computational fluid dynamics conference (CFD'99)
12. Pike G, Semenzato L, Colella P, Hilfinger PN (1999) Parallel 3D adaptive mesh refinement in Titanium. In: 9th SIAM conference on parallel processing for scientific computing, San Antonio, TX, March 1999
13. Su J, Yelick K (2005) Automatic support for irregular computations in a high-level language. In: 19th International Parallel and Distributed Processing Symposium (IPDPS)
14. Yelick K, Hilfinger P, Graham S, Bonachea D, Su J, Kamil A, Datta K, Colella P, Wen T (2007) Parallel languages and compilers: perspective from the titanium experience. *Int J High Perform Comput App* 21:266–290
15. Yelick K, Semenzato L, Pike G, Miyamoto C, Liblit B, Krishnamurthy A, Hilfinger P, Graham S, Gay D, Colella P, Aiken A (1998) Titanium: a high-performance Java dialect. *Concur: Pract Exp* 10:825–836

## Web Documentation Bibliography

GASNet Home Page. <http://gasnet.cs.berkeley.edu/>  
Titanium Project Home Page at <http://titanium.cs.berkeley.edu>.

## TLS

- [Speculation, Thread-Level](#)
- [Speculative Parallelization of Loops](#)

## TOP500

JACK DONGARRA, PIOTR LUSZCZEK  
University of Tennessee, Knoxville, TN, USA

## Definition

TOP500 is a list of 500 fastest supercomputers in the world ranked by their performance achieved from running the LINPACK Benchmark. The list is assembled twice a year and officially presented at two supercomputing conferences: one in Europe and one in the USA. This list has been put together since 1993.

## Discussion

Statistics on high-performance computers are of major interest to manufacturers, users, and potential users. These people wish to know not only the number of systems installed, but also the location of the various supercomputers within the high-performance computing community and the applications for which a computer system is being used. Such statistics can facilitate the establishment of collaborations, the exchange of data and software, and provide a better understanding of the high-performance computer market.

Statistical lists of supercomputers are not new. Every year since 1986, Hans Meuer has published system counts of the major vector computer manufacturers, based principally on those at the Mannheim Supercomputer Seminar. In the early 1990s, a new definition of supercomputer was needed to produce meaningful statistics. After experimenting with metrics based on processor count in 1992, the idea was born at the University of Mannheim to use a detailed listing of installed systems as the basis. In early 1993, Jack Dongarra was convinced to join the project with the LINPACK benchmark. A first test version was produced in May 1993. Today the TOP500 list is compiled by Hans Meuer of the University of Mannheim, Germany, Jack Dongarra of the University of Tennessee, Knoxville, and Erich Strohmaier and Horst Simon of NERSC/Lawrence Berkeley National Laboratory.

New statistics are required that reflect the diversification of supercomputers, the enormous performance difference between low-end and high-end models, the increasing availability of massively parallel processing (MPP) systems, and the strong increase in computing power of the high-end models of workstation suppliers (SMP).

To provide a new statistical foundation, the authors of the TOP500 decided in 1993 to assemble and maintain a list of the 500 most powerful computer systems. The list is updated twice a year. The first of these updates always coincides with the International Supercomputer Conference in June (submissions are accepted until April 15), the second one is presented in November at the IEEE Super Computer Conference in the USA (submissions are accepted until October 1st). The list is assembled with the help of high-performance computer experts, computational scientists, and manufacturers.

In the present list (called the TOP500), computers are ranked by their performance on the LINPACK Benchmark. The list is freely available at <http://www.top500.org/> where users can create additional sublists and statistics out of the TOP500 database on their own.

The main objective of the TOP500 list is to provide a ranked list of general-purpose systems that are in common use for high-end applications. A general-purpose system is expected to be able to solve a range of scientific problems.

The TOP500 list shows the 500 most powerful commercially available computer systems known. To keep the list as compact as possible, only a part of the information is shown:

- Nworld – Position within the TOP500 ranking
- Manufacturer – Manufacturer or vendor
- Computer – Type indicated by manufacturer or vendor
- Installation Site – Customer
- Location – Location and country
- Year – Year of installation/last major update
- Field of Application
- #Proc. – Number of processors (Cores)
- Rmax – Maximal LINPACK performance achieved
- Rpeak – Theoretical peak performance
- Nmax – Problem size for achieving Rmax
- N1/2 – Problem size for achieving half of Rmax

In the TOP500 List table, the computers are ordered first by their Rmax value. In the case of equal performances (Rmax value) for different computers, a choice was made to order by Rpeak. For sites that have the same computer, the order is by memory size and then alphabetically.

## Method of Solution

In an attempt to obtain uniformity across all computers in performance reporting, the algorithm used in solving the system of equations ( $Ax = b$ , for a dense matrix  $A$ ) in the benchmark procedure must conform to LU factorization with partial pivoting. In particular, the operation count for the algorithm must be  $2/3 n^3 + O(n^2)$  double point floating point operations (Rmax value is computed by dividing this count by the time taken to solve). Here a floating point operation is an addition or multiplication of 64-bit operands. This excludes the use of a fast matrix multiply algorithm like "Strassen's

Method" or algorithms which compute a solution in a precision lower than full precision (64 bit floating point arithmetic) and refine the solution using an iterative approach. This is done to provide a comparable set of performance numbers across all computers. Submitters of the results are free to implement their own solution as long as the above criteria are met. A reference implementation of the benchmark called HPL (High Performance LINPACK) is provided at: <http://www.netlib.org/benchmark/hpl/>. In addition to satisfying the rules, HPL also verifies the result with a numerical check of the obtained solution.

## Restrictions

The main objective of the TOP500 list is to provide a ranked list of general-purpose systems that are in common use for high-end applications. The authors of the TOP500 reserve the right to independently verify submitted LINPACK Benchmark [1] results, and exclude systems from the list, which are not valid or not general purpose in nature. A system is considered to be of general purpose if it is able to be used to solve a range of scientific problems. Any system designed specifically to solve the LINPACK benchmark problem or have as its major purpose the goal of a high TOP500 ranking will be disqualified. The systems in the TOP500 list are expected to be persistent and available for use for an extended period of time. In that period, it is allowed to submit new results which will supersede any prior submissions. Thus, an improvement over time is allowed. The TOP500 authors will reserve the right to deny inclusion in the list if it is suspected that the system violates these conditions.

The TOP500 List keepers can be reached by sending email to info at [top500.org](http://top500.org).

The TOP500 list can be found at [www.top500.org](http://www.top500.org).

## Related Entries

- ▶ [Benchmarks](#)
- ▶ [HPC Challenge Benchmark](#)
- ▶ [LINPACK Benchmark](#)
- ▶ [Livermore Loops](#)

## Bibliography

1. Dongarra JJ, Luszczek P, Petitet A (2003) The LINPACK benchmark: past, present, and future. *Concurr Comput Pract Exp* 15:1–18

## Topology Aware Task Mapping

ABHINAV BHATELE

University of Illinois at Urbana-Champaign, Urbana, IL, USA

### Synonyms

Graph embedding; MPI process mapping

### Definition

Topology aware task mapping refers to the mapping of communicating parallel objects, tasks, or processes in a parallel application on nearby physical processors to minimize network traffic, by considering the communication of the objects or tasks and the interconnect topology of the machine.

## Discussion

### Introduction

Processors in modern supercomputers are connected together using a variety of interconnect topologies: meshes, tori, fat-trees, and others. Increasing size of the interconnect leads to an increased sharing of resources (network links and switches) among messages and hence network contention. This can potentially lead to significant performance degradation for certain classes of parallel applications. Sharing of links can be avoided by minimizing the distance traveled by messages on the network. This is achieved by mapping communicating objects or tasks on nearby physical processors on the network topology and is referred to as topology aware task mapping. Topology aware mapping is a technique to minimize communication traffic over the network and hence optimize performance of parallel programs. It is becoming increasingly relevant for obtaining good performance on current supercomputers.

The general mapping problem is known to be NP-hard [1, 2]. Apart from parallel computing, topology aware mapping also has applications in graph embedding in mathematics and VLSI circuit design. The problem of embedding one graph on another while minimizing some metric has been well studied in mathematics. Layout of VLSI circuits to minimize length of the longest wire is another problem that requires mapping of one grid on to another. However, the problems

to be tackled are different in several aspects in parallel computing from mathematics or circuit layout. For example, in VLSI, the size of the host graph can be larger than that of the guest graph whereas, in parallel computing, typically, the host graph is equal to or smaller than the guest graph.

Research on topology aware mapping in parallel computing began in the 1980s with a paper by Bokhari [1]. Work in this area has primarily involved the development of heuristics that target different mapping scenarios. Heuristics typically provide close to optimal solutions in a reasonable time. Arunkumar et al. [3] categorize various heuristic techniques into – deterministic, randomized, and random start heuristics. Over the years, specific techniques better suited for certain architectures were developed – for hypercubes and array processors in the 1980s and meshes and tori in the 1990s. In the recent years, developers of certain parallel applications have also developed application specific mapping techniques to map their codes on to modern supercomputers [4–6].

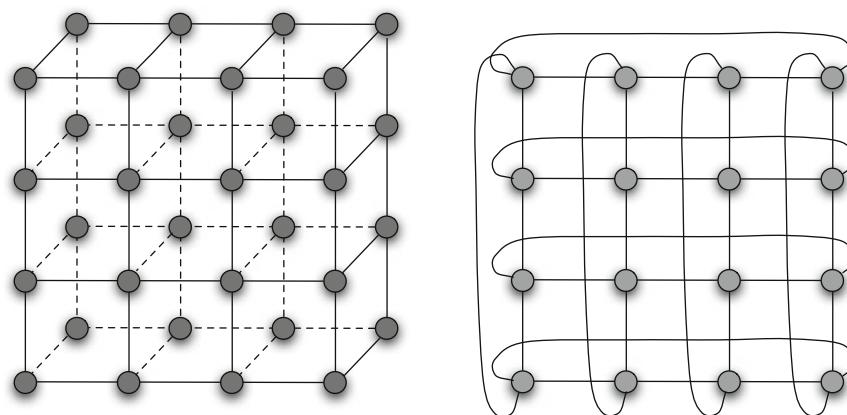
Prior to a survey of the various heuristic techniques for task mapping, a brief description of the existing

interconnect topologies and the kinds of communication graphs that are prevalent in parallel applications is essential.

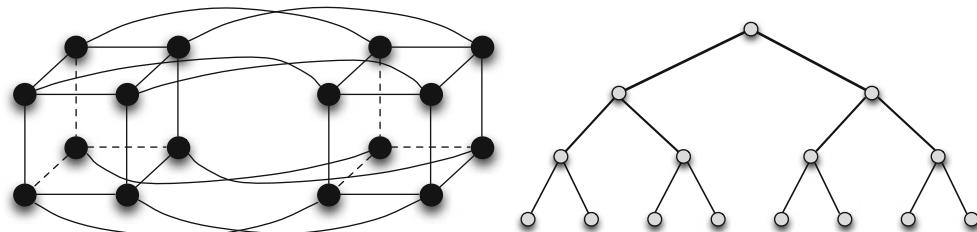
### Interconnect Topologies

Various common and radical topologies have been deployed in supercomputers, ranging from hypercubes to fat-trees to three-dimensional tori and meshes. They can be divided into two categories:

1. *Direct networks*: In direct networks, each processor is connected to a few other processors directly. A message travels from source to destination by going through several links connecting the processors. Hypercubes, tori, meshes, etc., are all examples of direct networks (see Figs. 1 and 2). Several modern supercomputers currently have a three-dimensional (3D) mesh or torus interconnect topology. IBM Blue Gene/L and Blue Gene/P machines are 3D tori built from blocks of a torus of size  $8 \times 8 \times 8$  nodes. Cray XT machines (XT5 and XE6) are also 3D tori. The primary difference between IBM and Cray machines is that on IBM machines, each allocated job partition



**Topology Aware Task Mapping. Fig. 1** A three-dimensional mesh and a two-dimensional torus



**Topology Aware Task Mapping. Fig. 2** A four-dimensional hypercube and a three-level fat-tree network

is a contiguous mesh or torus. However, on Cray machines, nodes are randomly selected for a job and do not constitute a complete torus.

2. *Indirect networks:* Indirect networks have switches which route the messages to the destination. No two processors are connected directly and messages always have to go through switches to reach their destination. Fat-tree networks are examples of indirect networks (see Fig. 2). Infiniband, IBM's Federation interconnect and SGI Altix machines are examples of fat-tree networks. LANL's RoadRunner also has a fat-tree network.

Some of these networks benefit from topology aware task mapping more than others. A significant percentage of the parallel machines in the 1980s had a hypercube interconnect and hence, much of the research then was directed toward such networks. More recent work involves optimizing applications on 3D meshes and tori.

## Communication Graphs

Tasks in parallel applications can interact in a variety of ways in terms of the specific communication partners, number of communicators, global versus localized communication, etc. All applications can be classified into a few different categories based on different parameters governing the communication patterns:

*Static versus dynamic communication:* Depending on whether the communication graph of the application changes at runtime, graph can be classified as static or dynamic. If the communication graph is static, topology aware mapping can be done offline and used for the entirety of the run. If the communication is dynamic, periodic remapping depending on the changes in the communication graph are required. Several categories of parallel applications such as Lattice QCD, Ocean Simulations, and Weather Simulations have a stencil-like communication pattern that does not change during the run. On the other hand, molecular dynamics and cosmological simulations have dynamic communication patterns.

*Regular versus irregular communication:* Communication in a parallel application can be regular (structured) or irregular (unstructured). An example of regular communication is a five-point stencil-like application where every task communicates with four of its neighbors.

When no specific pattern can be attributed to the communication graph, it is classified as irregular or unstructured. Unstructured grid computations are examples of applications with irregular communication graphs.

*Point-to-point versus collective communication:* Some applications primarily use point-to-point messages with minimal global communication. Others, however use collective operations such as broadcasts, reductions, and all-to-alls between all or a subset of processors. Different mapping algorithms are required to optimize different types of communication patterns.

Parallel applications can also be classified into computation bound or communication bound depending on the relative amount of communication involved. A large body of parallel applications spend a small portion of their overall execution time doing communication. Such applications will typically be unaffected by topology aware mapping. Communication-bound latency-sensitive applications benefit most in terms of performance from topology aware task mapping.

## The Mapping Process

An algorithm for mapping of tasks in an application requires two inputs – the communication graph of an application and the machine topology of the allocated job partition. Given these two inputs, the aim is to map communicating objects or tasks close to one another on nearby physical processors. The success of a mapping algorithm is evaluated in terms of minimizing or maximizing some function which correlates well with the contention on the network or actual application performance.

## Objective Functions

Mapping algorithms aim at minimizing some metric referred to as an objective function which should be chosen carefully. A good objective function is one that does an accurate evaluation of a mapping solution in terms of yielding better performance. Objective functions are also important to compare the optimality of different mapping solutions. Several objective functions which have been used for different mapping algorithms are listed below:

- Overlap between guest and host graph edges: One metric to determine the quality of the mapping is

the number of edges in the guest graph which fall on the host graph. This metric is referred to as the cardinality of the mapping by Bokhari [1]. The mapping which yields the highest cardinality is the best.

- Maximum dilation: This metric is used for architectures and applications where the longest edge in the communication graph determines the performance. In other words, the message that travels the maximum number of hops or links on the network determines the overall performance [7, 8].

$$\text{Maximum dilation} = \max_{i=1}^n |d_i| \quad (1)$$

The mapping which leads to the smallest dilation for any edge in the guest graph on the processor interconnect is the best.

- Hop-bytes: This is the weighted sum of all edges in the communication graph multiplied by their dilation on the processor graph as per the mapping algorithm [9, 10].

$$\text{Hop-bytes} = \sum_{i=1}^n d_i \times b_i \quad (2)$$

where  $d_i$  is the number of hops or links traveled by the message on the network and  $b_i$  is the size of the message in bytes.

Hop-bytes is a measure of the total communication traffic on the network and hence, an approximate indication of the contention. A smaller value for hop-bytes indicates less contention on the network. Average hops per byte is another way of expressing the same metric,

$$\text{Average hops per byte} = \frac{\sum_{i=1}^n d_i \times b_i}{\sum_{i=1}^n b_i} \quad (3)$$

The last two objective functions, maximum dilation and hop-bytes, are typically used today and are applicable in different scenarios. The choice of one over the other depends upon the parallel application and the architecture for which the mapping is being performed.

### Heuristic Techniques for Mapping

Owing to the general applicability of mapping in various fields, a huge body of work exists targeting this problem. Many techniques used for solving combinatorial optimization problems can be used for obtaining

solutions to the mapping problem. Simulated annealing, genetic algorithms, and neural network-based heuristics are examples of such physical optimization techniques. Other heuristic techniques are recursive partitioning, pairwise exchanges, and clustering and geometry-based mapping. Arunkumar et al. [3] categorize various heuristic techniques into – deterministic, randomized, and random start heuristics. The following sections discuss some of the mapping techniques classified into these categories.

### Deterministic Heuristics

In this class, the choice of search path is deterministic and typically a fixed search strategy is used taking the domain-specific knowledge about the parallel application into account. Yu et al. [11] present folding and embedding techniques to obtain deterministic solutions for mapping of two- and three-dimensional grids on to 3D mesh topologies. Their topology mapping library provides support for MPI virtual topology functions on IBM Blue Gene machines. Bhatele [12] uses domain-specific knowledge and communication patterns of parallel application for heuristic techniques such as “affine transformation” inspired mapping and guided graph traversals to map on to 3D tori. The mapping library developed as a result can map application graphs that are regular (n-dimensional grids) as well as those that are irregular. Several application developers such as those of Blue Matter [4], Qbox [5], and OpenAtom [6] have developed application specific mapping algorithms to map tasks on to processor topologies. Recursive graph partitioning-based strategies which partition both the application and processor graph for mapping also fall under this category [13]. Algorithms using deterministic algorithms are typically the fastest among the three categories.

### Randomized Heuristics

This category of solutions does not depend on domain-specific knowledge and uses search techniques that are randomized, yielding different solutions in successive executions. Neural networks, genetic algorithms, and simulated annealing-based heuristics are example of this class. Bokhari’s algorithm of pairwise exchanges accompanied by probabilistic jumps also falls under this category.

In genetic algorithm-based heuristics [3], possible mapping solutions are first encoded in some manner and a random population of such patterns is generated. Then different genetic operators such as crossover and mutation are applied to derive new generations from old ones. Certain criteria are used to estimate the fitness of a selection and unfit solutions are rejected. Given a termination rule, the best solution among the population is taken to be the solution at termination.

Obtaining an exact solution to the mapping problem is difficult and iterative algorithms tend to produce solutions that are not globally optimal. The technique of simulated annealing provides a mechanism to escape local optima and hence is a good fit for mapping problems. The most important considerations for a simulated annealing algorithm are deciding a good objective function and an annealing schedule. This technique has been used for processor and link assignment by Midkiff et al. [14] and Bhanot et al. [15].

### Random Start Heuristics

In some algorithms, a random initial mapping is chosen and then improved iteratively. Such solutions fall under the category of *random start* heuristics. Techniques such as pairwise exchanges and recursive partitioning fall under this category.

The technique of pairwise exchanges that starts from an initial assignment, is a simple brute force method which has been used with different variations to tackle the mapping problem [7]. The basic idea is simple: An objective function or metric to be optimized is selected and then an initial mapping of the guest graph on the host graph is determined. Then, a pair of nodes is chosen, either randomly or based on some selection criteria and their mappings are interchanged. If the metric or objective function becomes better, the exchange is preserved and the process is repeated, until some termination criterion is achieved.

Another technique in this class is task clustering followed by cluster allocation. In the clustering phase, tasks are clustered into groups equal to the number of processors using recursive min-cut algorithms. Then these clusters are allocated to the processors by starting with a random assignment and iteratively improving it by local exchanges. The first phase aims at minimizing intercluster communication without comprising load balancing while the second phase aims at minimizing

inter-processor communication. This is especially useful for models such as Charm++ where the number of tasks is much larger than the number of processors.

### Future Directions

The emergence of new architectures and network topologies requires modifying existing algorithms and developing new ones to suit them. As an example, the increase in number of cores per node adds another dimension to the network topology and should be taken into account. Algorithms also need to be developed for new parallel applications. There is a growing need for runtime support in the form of an automated mapping framework that can map applications intelligently on to the processor topology. This will reduce the burden on application developers to map individual applications and will also help reuse algorithms across similar communication graphs. Bhatele et al. [16] are making some efforts in this direction. There is an increasing demand for support in the MPI runtime for mapping of MPI virtual topology functions [11].

The increase in size of parallel machines and in the number of threads in a parallel program requires parallel and distributed techniques for mapping. Gathering the entire communication graph on one processor and applying sequential centralized techniques will not be feasible in the future. Hence, an effort should be made towards developing strategies which are distributed, scalable, and can be run in parallel. Hierarchical multilevel graph partitioning techniques are one such effort in this direction.

### Related Entries

- ▶ [Cray XT3 and Cray XT Series of Supercomputers](#)
- ▶ [Cray XT4 and Seastar 3-D Torus Interconnect](#)
- ▶ [Graph Partitioning](#)
- ▶ [Hypercubes and Meshes](#)
- ▶ [IBM Blue Gene Supercomputer](#)
- ▶ [Infiniband](#)
- ▶ [Interconnection Networks](#)
- ▶ [Load Balancing, Distributed Memory](#)
- ▶ [Locality of Reference and Parallel Processing](#)
- ▶ [Processes, Tasks, and Threads](#)
- ▶ [Routing \(Including Deadlock Avoidance\)](#)
- ▶ [Space-Filling Curves](#)
- ▶ [Task Graph Scheduling](#)

## Bibliographic Notes and Further Reading

Bokhari [1] wrote one of the first papers on task mapping for parallel programs. A good discussion of the various objective functions used for comparing mapping algorithms can be found in [7]. Fox et al. [17] divide the various mapping algorithms into physical optimization and heuristic techniques. Arunkumar et al. [3] provide another classification into deterministic, randomized, and random start heuristics.

Application developers attempting to map their parallel codes can gain insights from mapping algorithms developed by individual application groups [4–6]. Bhatele and Kale have been developing an automatic mapping framework for mapping of Charm++ and MPI applications to the processor topology [12]. They are also developing techniques for parallel and distributed topology aware mapping.

## Bibliography

1. Bokhari SH (1981) On the mapping problem. *IEEE Trans Comput* 30(3):207–214
2. Kasahara H, Narita S (1984) Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Trans Comput* 33:1023–1029
3. Arunkumar S, Chockalingam T (1992) Randomized heuristics for the mapping problem. *Int J High Speed Comput (IJHSC)* 4(4):289–300
4. Fitch BG, Rayshubskiy A, Eleftheriou M, Ward TJC, Giampapa M, Pitman MC (2006) Blue matter: approaching the limits of concurrency for classical molecular dynamics. In: SC'06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, Tampa, ACM Press, New York, 11–17 Nov 2006
5. Gygi F, Draeger EW, Schulz M, Supinski BRD, Gunnels JA, Austel V, Sexton JC, Franchetti F, Kral S, Ueberhuber C, Lorenz J (2006) Large-scale electronic structure calculations of high-Z metals on the blue gene/L platform. In: SC'06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, ACM Press, New York
6. Bhatelé A, Bohm E, Kalé LV (2011) Optimizing communication for Charm++ applications by reducing network contention. *Concurr Comput* 23(2):211–222
7. Lee S-Y, Aggarwal JK (1987) A mapping strategy for parallel processing. *IEEE Trans Comput* 36(4):433–442
8. Berman F, Snyder L (1987) On mapping parallel algorithms into parallel architectures. *J Parallel Distrib Comput* 4(5):439–458
9. Ercal F, Ramanujam J, Sadayappan P (1988) Task allocation onto a hypercube by recursive mincut bipartitioning. In: Proceedings of the 3rd conference on Hypercube concurrent computers and applications, ACM Press, New York, pp 210–221
10. Agarwal T, Sharma A, Kalé LV (2006) Topology-aware task mapping for reducing communication contention on large parallel machines, In: Proceedings of IEEE International Parallel and Distributed Processing Symposium 2006, Rhodes Island, 25–29 Apr 2006. IEEE, Piscataway
11. Yu H, Chung I-H, Moreira J (2006) Topology mapping for blue gene/L supercomputer. In: SC'06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, Tampa, 11–17 Nov 2006. ACM, New York, p II6
12. Bhatele A (2010) Automating topology aware mapping for supercomputers. Ph.D. thesis, Dept. of Computer Science, University of Illinois. <http://hdl.handle.net/2142/16578> (August 2010)
13. Kernighan BW, Lin S (1970) An efficient heuristic procedure for partitioning graphs. *Bell Syst Tech J* 49(1):291–307
14. Bollinger SW, Midkiff SF (1988) Processor and link assignment in multicomputers using simulated annealing. In: 1988 ICPP, vol 1, Aug 1988, pp 1–7
15. Bhanot G, Gara A, Heidelberger P, Lawless E, Sexton JC, Walkup R (2005) Optimizing task layout on the blue gene/L supercomputer. *IBM J Res Dev* 49(2/3):489–500
16. Bhatele A, Gupta G, Kale LV, Chung I-H (2010) Automated mapping of regular communication graphs on mesh interconnects. In: Proceedings of International Conference on High Performance Computing & Simulation (HiPCS) 2010, Caen, 28 June–2 July 2010. IEEE, Piscataway
17. Mansour N, Ponnusamy R, Choudhary A, Fox GC (1993) Graph contraction for physical optimization methods: a quality-cost tradeoff for mapping data on parallel computers. In: ICS'93: Proceedings of the 7th International Conference on Supercomputing, Tokyo, 19–23 July 1993. ACM, New York, pp 1–10

## Torus

### ► Networks, Direct

## Total Exchange

### ► Allgather

## Trace Scheduling

STEFAN M. FREUDENBERGER  
Zürich, Switzerland

## Definition

Trace scheduling is a global acyclic instruction scheduling technique in which the scheduling region consists

of a linear acyclic sequence of basic blocks embedded in the control flow graph. Trace scheduling differs from other global acyclic scheduling techniques by allowing the scheduling region to be entered after the first instruction.

Trace scheduling was the first global instruction scheduling technique that was proposed and successfully implemented in both research and commercial compilers. By demonstrating that simple microcode operations could be statically compacted and scheduled on multi-issue hardware, trace scheduling provided the basis for making large amounts of instruction-level parallelism practical. Its first commercial implementation demonstrated that commercial codes could be statically compiled for multi-issue architectures, and thus greatly influenced and contributed to the performance of superscalar architectures. Today, the ideas of trace scheduling and its descendants are implemented in most compilers.

## Discussion

### Introduction

Global scheduling techniques are needed for processors that expose instruction-level parallelism (ILP), that is, processors that allow multiple operations to execute simultaneously. This situation may independently arise for two reasons: either because a processor issues more than a single operation during each clock cycle, or because a processor allows issuing independent operations while deeply pipelined operations are still executing. The number of independent operations that need to be found for an ILP processor is a function of both the number of operations issued per clock cycle, and the latency of operations, whether computational or memory. The latency of computational operations depends upon the design of the functional units. The latency of memory operations depends upon the design and latencies of caches and main memory, as well as on the availability of prefetch and cache-bypassing operations. Global scheduling techniques are needed for these processors because the number of independent operations available in a typical basic block is too small to fully utilize their available hardware resources. By expanding the scheduling region, more operations become available for scheduling. Global scheduling techniques differ from other global code motion techniques (such as

loop-invariant code motion or partial redundancy elimination) because they take into account the available hardware resources (such as available functional units and operation issue slots).

Instruction scheduling techniques can be broadly classified based on the region that they schedule, and whether this region is cyclic or acyclic. Algorithms that schedule only single basic blocks are known as *local scheduling* algorithms; algorithms that schedule multiple basic blocks at once are known as *global scheduling* algorithms. Global scheduling algorithms that operate on entire loops of a program are known as *cyclic scheduling* algorithms, while methods that impose a scheduling barrier at the end of a loop body are known as *acyclic scheduling* algorithms. Global scheduling regions include regions consisting of a single basic block as a “degenerate” form of region, and acyclic schedulers may consider entire loops but, unlike cyclic schedulers, stop at the loops’ back edges (a back edge points to an ancestor in a depth-first traversal of the control flow graph; it captures the flow from one iteration of the loop to the start of the next iteration).

All scheduling algorithms can benefit from hardware support. When control-dependent operations that can cause side effects move above their controlling branch, they need to be either executed conditionally so that their effects only arise if the operation is executed in the original program order, or any side effects must be delayed until the point at which the operation would have been executed originally.

Hardware techniques to support this include *predication* of operations, implicit or explicit *register renaming*, and mechanisms to suppress or delay exceptions in order to prevent an incorrect exception to be signaled. Predication of operations controls whether the side effects of the predicated operations become visible to the program state through an additional predicate operand. The predicate operand can be implicit (such as the conditional execution of operations in branch delay slots depending on the outcome of the branch condition) or explicit (through an additional machine register operand); in the latter case, the predicate operand could simply be the same predicate that controls the conditional branch on which the operation was control-dependent in the original flow graph (in which case the predicated operation could move just above a single conditional branch). Register renaming refers to the technique where additional machine registers are

used to hold the results of an operation until the point where the operation would have occurred in the original program order.

Global scheduling algorithms principally consist of two phases: *region formation* and *schedule construction*. Algorithms differ in the shape of the region and the global code motions permitted during scheduling. Depending on the region and the allowed code motions, *compensation code* needs to be inserted at appropriate places in the control flow graph to maintain the original program semantics; depending on the code motions allowed during scheduling, compensation code needs to be inserted during the scheduling phase of the compiler.

Trace scheduling allows traces to be entered after the first operation and before the last operation. This complicates the determination of compensation code because the location of *rejoin points* cannot be done before a trace has been scheduled. This leads to the following overall trace scheduling loop:

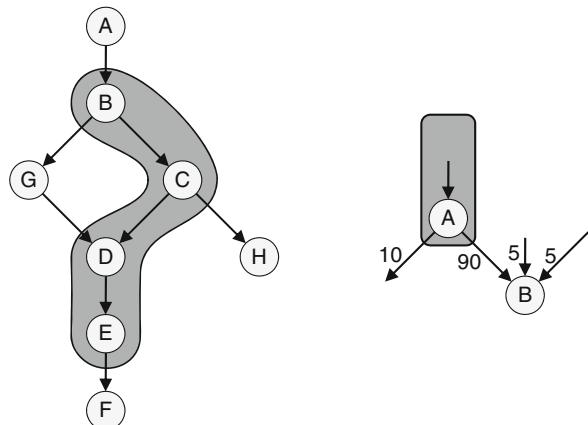
```
while (unscheduled operations remain)
{
 select trace T
 construct schedule for T
 bookkeeping -
 determine rejoin points to T
 generate compensation code
}
```

The remainder of this entry first discusses region formation and schedule construction in general and as it applies to trace scheduling, and then compares trace scheduling to other acyclic global scheduling techniques. Cyclic scheduling algorithms are discussed elsewhere.

### Region Formation – Trace Picking

Traces were the first global scheduling region proposed, and represent contiguous linear paths through the code (Fig. 1). More formally, a trace consists of the operations of a sequence of basic blocks  $B_0, B_1, \dots, B_n$  with the properties that:

- Each basic block is a predecessor of the next in the sequence (i.e., for each  $k = 0, \dots, n - 1$ ,  $B_k$  is a predecessor of  $B_{k+1}$ , and  $B_{k+1}$  is a successor of  $B_k$  in the control flow graph).



**Trace Scheduling.** Fig. 1 Trace selection. The left diagram shows the selected trace. The right diagram illustrates the mutual-most-likely trace picking heuristic: assume that  $A$  is the last operation of the current trace, and that  $B$  is one of  $A$ 's successors. Here  $B$  is the most likely successor of  $A$ , and  $A$  is the most likely predecessor of  $B$

- For any  $j, k$  there is no path  $B_j \rightarrow B_k \rightarrow B_j$  except for those that include  $B_0$  (i.e., the code is cycle free, except that the entire region can be part of some encompassing loop).

Note that this definition does not exclude forward branches within the region, nor control flow that leaves the region and reenters it at a later point. This generality has been controversial in the research community because many felt that the added complexity of its implementation was not justified by its added benefit and has led to several alternative approaches that are discussed below.

Of the many ways in which one can form traces, the most popular algorithm employs the following simple trace formation algorithm:

- Pick the as-yet unscheduled operation with the largest expected execution frequency as the seed operation of the trace.
- Grow the trace both forward in the direction of the flow graph as well as backward, picking the *mutually most-likely* successor (predecessor) operation to the currently last (first) operation on the trace.
- Stop growing a trace when either no mutually most-likely successor (predecessor) exists, or when some heuristic trace length limit has been reached.

The mutually most-likely successor  $S$  of an operation  $P$  is the operation with the properties that:

- $S$  is the most likely successor of  $P$ ;
  - $P$  is the most likely predecessor of  $S$ .

For this definition, it is immaterial whether the likelihood that  $S$  follows  $P$  ( $P$  precedes  $S$ ) is based on available profile data collected during earlier runs of the program, has been determined by a synthetic profile, or is based on source annotations in the program. Of course, the more benefit is derived from having picked the correct trace, the greater is the penalty when picking the wrong trace.

Trace picking is the region formation technique used for trace scheduling. Other acyclic region formation techniques and their relationship to trace scheduling are discussed below.

## Region Enlargement

Trace selection alone typically does not expose enough ILP for the instruction scheduler of a typical ILP processor. Once the limit on the length of a “natural” trace has been reached (e.g., the entire loop body), *region-enlargement* techniques can be employed to further increase the size of the region, albeit at the cost of a larger code size for the program. Many enlargement techniques exploit the fact that programs iterate

and grow the size of a region by making extra copies of highly iterated code, leading to a larger region that contains more ILP.

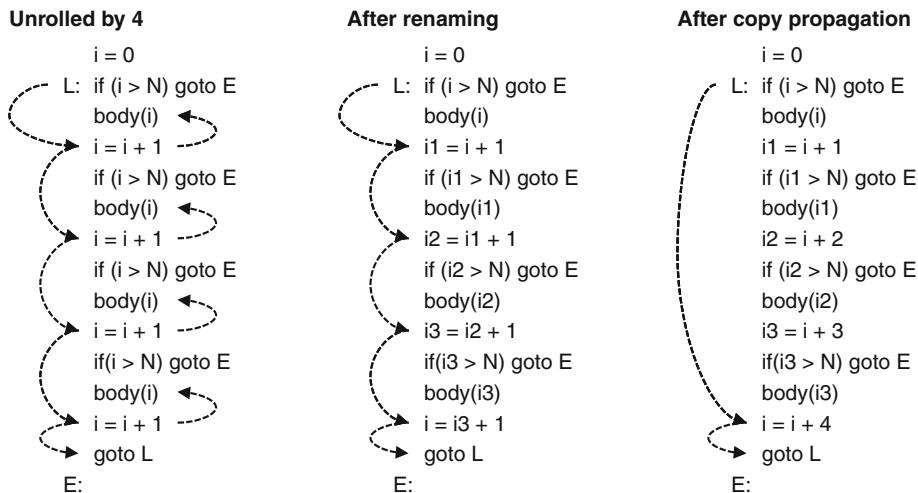
These code-replicating techniques have been criticized by advocates of other approaches, such as cyclic scheduling and loop-level parallel processing, because comparable benefits to larger schedule regions may be found using other techniques. However, no study appears to exist that quantifies such claims.

The simplest and oldest region-enlargement technique is *loop unrolling* (Fig. 2): to unroll a loop, duplicate its body several times, change the targets of the back edges of each copy but the last to point to the header of the next copy (so that the back edges of the last copy point back to the loop header of the first copy). Variants of loop unrolling include pre-/post-conditioning of a loop by  $k$  for counted *for* loops with unknown loop bounds (leading to two loops: a “fixup loop” that executes up to  $k$  iterations; and a “main loop” that is unrolled by  $k$  and has its internal exits removed; the fixup loop can precede or follow the main loop), and loop peeling by the expected small iteration count. When the iteration count of the fixup loop of a p-conditioned loop is small (which it typically is), the fixup loop is completely unrolled.

Typically, loop unrolling is done before region formation so that the enlarged region becomes available

| Original loop                      | Unrolled by 4                                                                                                                          | Pre-conditioned by 4                                                                                        | Post-conditioned by 4                                                      |
|------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------|
| L: if ... goto E<br>body<br>goto L | L: if ... goto E<br>body<br>if ... goto E<br>body<br>if ... goto E<br>body<br>if ... goto E<br>body<br>if ... goto E<br>body<br>goto L | if ... goto L<br>body<br>if ... goto L<br>body<br>if ... goto L<br>body<br>L: if ... goto E<br>body<br>body | L: if ... goto X<br>body<br>body<br>body<br>body<br>body<br>goto L         |
| E:                                 | body<br>if ... goto E<br>body<br>goto L                                                                                                | if ... goto E<br>body<br>body<br>body<br>E:                                                                 | X: if ... goto E<br>body<br>if ... goto E<br>body<br>if ... goto E<br>body |
|                                    |                                                                                                                                        | body<br>body<br>goto L                                                                                      | E:                                                                         |

**Trace Scheduling.** Fig. 2 Simplified illustration of variants of loop unrolling. “if” and “goto” represent the loop control operations; “body” represents the part of the loop without loop-related control flow. In the general case (e.g., a *while* loop) the loop exit tests remain inside the loop. This is shown in the second column (“unrolled by 4”). For counted loops (i.e., *for* loops), the compiler can condition the unrolled loop so that the loop conditions can be removed from the main body of the loop. Two variants of this are shown in the two rightmost columns. Modern compilers will typically precede the loop with a zero trip count test and place the loop condition at the bottom of the loop. This removes the unconditional branch from the loop



**Trace Scheduling. Fig. 3** Typical induction variable manipulations for loops. Downward arrows represent flow dependences; upward arrows represent anti dependences. Only the critical dependences are shown

to the region selector. This is done to keep the region selector simpler but may lead to phase-ordering issues, as loop unrolling has to guess the “optimal” unroll amount. At the same time, when loops are unrolled before region formation then the resulting code can be scalar optimized in the normal fashion; in particular height-reducing transformations that remove dependences between the individual copies of the unrolled loop body can expose a larger amount of parallelism between the individual iterations (Fig. 3). Needless to say, if no parallelism between the iterations exists or can be found, loop unrolling is ineffective.

Loop unrolling in many industrial compilers is often rather effective because a heuristically determined small amount of unrolling is sufficient to fill the resources of the target machine.

### Region Compaction – Instruction Scheduler

Once the scheduling region has been selected, the instruction scheduler assigns functional units of the target machine and time slots in the instruction schedule to each operation of the region. In doing so, the scheduler attempts to minimize an objective cost function while maintaining program semantics and obeying the resource limitations of the target architecture. Often, the objective cost function is the expected execution time,

but other objective functions are possible (for example, code size and energy efficiency could be part of an objective function).

The semantics of a program defines certain sequential constraints or *dependences* that must be maintained by a valid execution. These dependences preclude some reordering of operations within a program. The data flow of a program imposes *data dependences*, and the control flow of a program imposes *control dependences*. (Note the difference between control flow and control dependence: block  $B$  is control dependent on block  $A$  if  $A$  precedes  $B$  along some path, but  $B$  does not post-dominate  $A$ . In other words, the result of the control decision made in  $A$  directly affects whether or not  $B$  is executed.)

There are three types of data dependences: *read-after-write* dependences (also called *RAW*, *flow*, or *true* dependences), *write-after-read* dependences (also called *WAR* or *anti* dependences), and *write-after-write* dependences (also called *WAW* or *output* dependences). The latter two types are also called *false* dependences because they can be removed by renaming.

There are two types of control dependences: *split* dependences may prevent operations from moving below the exit of a basic block, and *join* dependences may prevent operations from moving above the entrance to a basic block. Control dependence does not constrain the relative order of operations within a

basic block but rather expresses constraints on moving operations between basic blocks.

Both data and control dependences represent ordering constraints on the program execution, and hence induce a partial ordering on the operations. Any partial ordering can be represented as a *directed acyclic graph* (DAG), and DAGs are indeed often used by scheduling algorithms. Variants to the simple DAG are the *data dependence graph* (DDG), and the *program dependence graph* (PDG). All these graphs represent operations as nodes and dependences as edges (some graphs only express data dependences, while others include both data and control dependences).

### Code Motion Between Adjacent Blocks

Two fundamental techniques, predication and speculation, are employed by schedulers (or earlier phases) to transform or remove control dependence. While it is sometimes possible to employ either technique, they represent independent techniques, and usually one is more natural to employ in a given situation. Speculation is used to move operations above a branch that is highly weighted in one direction; predication is used to collapse short sequences of alternative operations following a branch that is nearly equally likely in each direction. Predication can also play an important role in software pipelining.

*Speculative code motion* (or *code hoisting* and sometimes *code sinking*) moves operations above control-dominating branches (or below joins for sinking). In principle, this transformation does not always maintain the original program semantics, and in particular it may change the exception behavior of the program. If an operation may generate an exception and the exception recovery model does not allow speculative exceptions to be dismissed (ignored), then the compiler must generate recovery code that raises the exception at the original program point of the speculated operation. Unlike predication, speculation actually removes control dependences, and thus potentially reduces the length of the critical path of execution. Depending on the shape and size of recovery code, and if multiple operations are speculated, the addition of recovery code can lead to a substantial amount of code.

*Predication* is a technique where with hardware support operations have an additional input operand, the predicate operand, which determines whether any

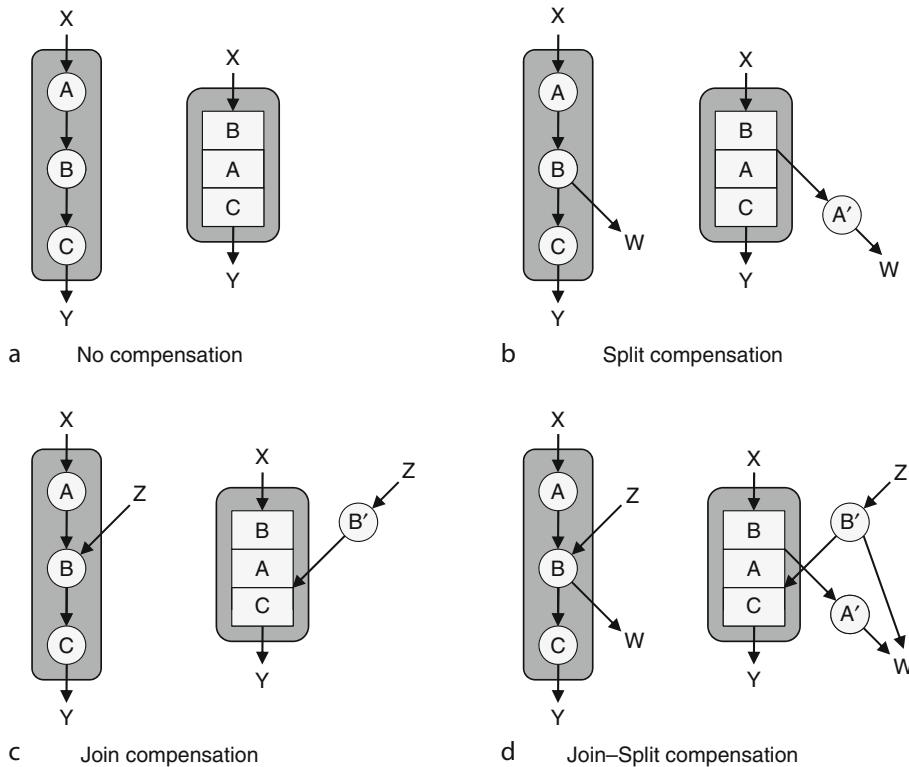
effects of executing the operations are seen by the program execution. Thus, from an execution point of view, the operation is conditionally executed under the control of the predicate input. Hence changing a control-dependent operation to its predicated equivalent that depends on a predicate that is equivalent to the condition of the control dependence turns control dependence into data dependence.

### Trace Compaction

There are many different scheduling techniques, which can broadly be classified by features into cycle versus operation scheduling, linear versus graph-based, cyclic versus acyclic, and greedy versus backtracking. However, for trace scheduling itself the scheduling technique employed is not of major concern; rather, trace scheduling distinguishes itself from other global acyclic scheduling techniques by the way the scheduling region is formed, and by the kind of code motions permitted during scheduling. Hence these techniques will not be described here, and in the following, a greedy graph-based technique, namely list scheduling, will be used.

### Compensation Code

During scheduling, typically only a very small number of operations can be moved freely between basic blocks without changing program semantics. Other operations may be moved only when additional *compensation code* is inserted at an appropriate place in order to maintain original program semantics. Trace scheduling is quite general in this regard. Recall that a trace may be entered after the first instruction, and exited before the last instruction. In addition, trace scheduling allows operations in the region (trace) to move freely during scheduling relative to entries (join points) to and exits (split points) from the current trace. A separate *bookkeeping* step restores the original program semantics after trace compaction through the introduction of compensation code. It is this freedom of code motion during scheduling, and the introduction of compensation code between the scheduling of individual regions, that represents a major difference between trace scheduling and other acyclic scheduling techniques.



**Trace Scheduling.** Fig. 4 Basic scenarios for compensation code. In each diagram, the *left* part shows the selected trace, the *right* part shows the compacted code where operation *B* has moved above operation *A*

Since trace scheduling allows operations to move above join points as well as below split points (conditional branches) in the original program order, the bookkeeping process includes the following kinds of compensation. Note that a complete discussion of all the intricacies of compensation code is well beyond the scope of this entry; however, the following is a list of the simple concepts that form the basis of many of the compensation techniques used in compilers.

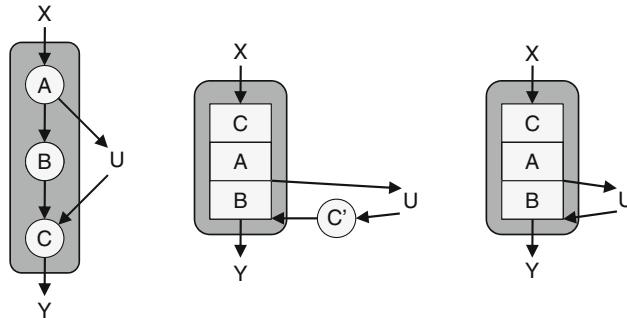
**No compensation** (Fig. 4a). If the global motion of an operation on the trace does not change the relative order of operations with respect to split and join points, no compensation code is needed. This covers the situation when an operation moves above a split, in which case the operation becomes *speculative*, and requires compensation depending on the recovery model of exceptions: in the case of *dismissible speculation*, no compensation code is needed; in the case of *recovery speculation*, the compiler has to emit a recovery block to guarantee the timely delivery of exceptions for correctly speculated operations.

**Split compensation** (Fig. 4b). When an operation *A* moves below a split operation *B* (i.e., a conditional branch), a copy of *A* (called *A'*) must be inserted on the off-trace split edge. When multiple operations move below a split operation, they are all copied on the off-trace edge in source order. These copies are unscheduled, and hence will be picked and scheduled later during the trace scheduling of the program.

**Join compensation** (Fig. 4c). When an operation *B* moves above a join point *A*, a copy of *B* (called *B'*) must be copied on the off-trace join edge. When multiple operations move above a join point, they are all copied on the off-trace edge in source order.

**Join-Split compensation** (Fig. 4d). When splits are allowed to move above join points, the situation becomes more complicated: when the split is copied on the rejoin edge, it must account for any split compensation and therefore introduce additional control paths with additional split copies.

These rules define the compensation code required to correctly maintain the semantics of the original



**Trace Scheduling. Fig. 5** Compensation copy suppression. The *left* diagram shows the selected trace. The *middle* diagram shows the compacted code where operation  $C$  has moved above operation  $A$  together with the normal join compensation. The *right* diagram shows the result of compensation copy suppression assuming that  $C$  is available at  $Y$

program. The following observations can be used to heuristically control the amount of compensation code that is generated.

To limit split compensation, the Multiflow Trace Scheduling compiler [12], the first commercial compiler to implement trace scheduling, required that all operations that precede a split on the trace precede the split on the schedule. While this limits the amount of available parallelism, the intuitive explanation is that a trace represents the most likely execution path; the on-trace performance penalty of this restriction is small; and off-trace the same operations would have to be executed in the first place. Multiflow's implementation excluded memory-store operations from this heuristic because in Multiflow's Trace architecture stores were unconditional and hence could not move above splits; they were allowed to move below splits to avoid serialization between stores and loop exits in unrolled loops. The Multiflow compiler also restricted splits to remain in source order. Not only did this reduce the amount of compensation code, it also ensured that all paths created by compensation code are subsets of paths (possibly rearranged) in the flow graph before trace scheduling.

Another observation concerns the possible suppression of compensation copies [8] (Fig. 5): sometimes an operation  $C$  that moves above a join point following an operation  $B$  actually moves to a position on the trace that dominates the join point. When this happens, and the result of  $C$  is still available at the join point, no copy of  $C$  is needed. This situation often arises when

loops with internal branches are unrolled. Without copy suppression, such loops can generate large amounts of redundant compensation code.

## Bibliographic Notes and Further Reading

The simplest form of a scheduling region is a region where all operations come from a single-entry single-exit straight-line piece of code (i.e., a basic block). Since these regions do not contain any internal control flow, they can be scheduled using simple algorithms that maintain the partial order given by data dependences. (For simplicity, it is best to require that operations that could incur an exception must end their basic block, allowing the exception to be caught by an exception handler.)

Traces and trace scheduling were the first region-scheduling techniques proposed. They were introduced by Fisher [6, 7] and described more carefully in Ellis' thesis [4]. By demonstrating that simple microcode operations could be statically compacted and scheduled on multi-issue hardware trace scheduling provided the basis for making VLIW machines practical. Trace scheduling was implemented in the Multiflow compiler [12]; by demonstrating that commercial codes could be statically compiled for multi-issue architectures, this work also greatly influenced and contributed to the performance of superscalar architectures. Today, ideas of trace scheduling and its descendants are implemented in most compilers (e.g., GCC, LLVM, Open64, Pro64, as well as commercial compilers).

Trace scheduling inspired several other global acyclic scheduling techniques. The most important linear acyclic region-scheduling techniques are presented next.

## Superblocks

Hwu and his colleagues on the IMPACT project have developed a variant of trace scheduling called *superblock scheduling*. Superblocks are traces with the added restriction that the superblock must be entered at the top [2, 3]. Hence superblocks can be joined only before the first or after the last operation in the superblock. As such, superblocks are single-entry, multiple-exit traces.

Since superblocks do not contain join points, scheduling a superblock cannot generate any join or join-split compensation. By also prohibiting motion below splits, superblock scheduling avoids the need of generating compensation code outside the schedule region, and hence does not require a separate bookkeeping step. With these restrictions, superblock formation can be completed before scheduling starts, simplifying its implementation.

Superblock formation often includes a technique called *tail duplication* to increase the size of the superblock: tail duplication copies any operations that follow a rejoin in the original control flow graph and that are part of the superblock into the rejoin edge, thus effectively lowering the rejoin point to the end of the superblock. This is done at superblock formation time, before any compaction takes place [11].

A variant of superblock scheduling that allows speculative code motion is sentinel scheduling [14].

## Hyperblocks

A different approach to global acyclic scheduling also originated with the IMPACT project. *Hyperblocks* are superblocks that have eliminated internal control flow using predication [13]. As such, hyperblocks are single-entry, multiple-exit traces (superblocks) that use predication to eliminate internal control flow.

## Treeregions

Treeregions [9, 10] consist of the operations from a list of basic blocks  $B_0, B_1, \dots, B_n$  with the properties that:

- For each  $j > 0$ ,  $B_j$  has exactly one predecessor.

- For each  $j > 0$ , the predecessor  $B_i$  of  $B_j$  is also on the list, where  $i < j$ .

Hence, treeregions represent trees of basic blocks in the control flow graph. Since treeregions do not contain any side entrances, each path through a treeregion yields a superblock. Like superblock compilers, treeregion compilers employ tail duplication and other region-enlarging techniques. More recent work by Zhou and Conte [16, 17] shows that treeregions can be made quite effective without significant code growth.

## Nonlinear Regions

Nonlinear region approaches include percolation scheduling [1] and DAG-based scheduling [15]. Trace scheduling-2 [5] extends treeregions by removing the restriction on side entrances. However, its implementation proved so difficult that its proposer eventually gave up on it, and no formal description or implementation of it is known to exist.

## Related Entries

- [Modulo Scheduling and Loop Pipelining](#)

## Bibliography

1. Aiken A, Nicolau A (1988) Optimal loop parallelization. In: Proceedings of the SIGPLAN 1988 conference on programming language design and implementation, June 1988, pp 308–317
2. Chang PP, Warter NJ, Mahlke SA, Chen WY, Hwu WW (1991) Three superblock scheduling models for superscalar and super-pipelined processors. Technical Report CRHC-91-29. Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign
3. Chang PP, Mahlke SA, Chen WY, Warter NJ, Hwu WW (1991) IMPACT: an architectural framework for multiple-instruction-issue processors. In: Proceedings of the 18th annual international symposium on computer architecture, May 1991, pp 266–275
4. Ellis JR (1985) Bulldog: a compiler for VLIW architectures. PhD thesis, Yale University
5. Fisher JA (1993) Global code generation for instruction-level parallelism: trace scheduling-2. Technical Report HPL-93-43. Hewlett-Packard Laboratories
6. Fisher JA (1981) Trace scheduling: a technique for global microcode compaction, IEEE Trans Comput, July 1981, 30(7):478–490
7. Fisher JA (1979) The optimization of horizontal microcode within and beyond basic blocks. PhD dissertation. Technical Report COO-3077-161. Courant Institute of Mathematical Sciences, New York University, New York, NY

8. Freudberger SM, Gross TR, Lowney PG (1994) Avoidance and suppression of compensation code in a trace scheduling compiler, ACM Trans Program Lang Syst, July 1994, 16(4):1156–1214
9. Havanki WA (1997) Treigon scheduling for VLIW processors. MS thesis. Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC
10. Havanki WA, Banerjia S, Conte TM (1998) Treigon scheduling for wide issue processors. In: Proceedings of the fourth international symposium on high-performance computer architecture, February 1998, pp 266–276
11. Hwu WW, Mahlke SA, Chen WY, Chang PP, Warter NJ, Bringmann RA, Ouellette RG, Hank RE, Kiyohara T, Haab GE, Holm JG, Lavery DM (May 1993) The superblock: an effective technique for VLIW and superscalar compilation. J Supercomput, 7(1–2):229–248
12. Lowney PG, Freudberger SM, Karzes TJ, Lichtenstein WD, Nix RP, O'Donnell JS, Ruttenberg JC (1993) The Multiflow trace scheduling compiler, J Supercomput, May 1993, 7(1–2):51–142
13. Mahlke SA, Lin DC, Chen WY, Hank RE, Bringmann RA (1992) Effective compiler support for predicated execution using the hyperblock. In: Proceedings of the 25th annual international symposium on microarchitecture, 1992, pp 45–54
14. Mahlke SA, Chen WY, Bringmann RA, Hank RE, Hwu WW, Rau BR, Schlansker MS (1993) Sentinel scheduling: a model for compiler-controlled speculative execution, ACM Trans Comput Syst, November 1993, 11(4):376–408
15. Moon SM, Ebcioğlu K (1997) Parallelizing nonnumerical code with selective scheduling and software pipelining, ACM Trans Program Lang Syst, November 1997, 19(6):853–898
16. Zhou H, Conte TM (2002) Code size efficiency in global scheduling for ILP processors. In: Proceedings of the sixth annual workshop on the interaction between compilers and computer architectures, February 2002, pp 79–90
17. Zhou H, Jennings MD, Conte TM (2001) Tree traversal scheduling: a global scheduling technique for VLIW/EPIC processors. In: Proceedings of the 14th annual workshop on languages and compilers for parallel computing, August 2001, pp 223–238

concurrent or parallel systems. Traces, or equivalently, elements in a free partially commutative monoid, are given by a sequence of letters (or atomic actions). Two sequences are assumed to be equal if they can be transformed into each other by equations of type  $ab = ba$ , where the pair  $(a, b)$  belongs to a predefined relation between letters. This relation is usually called *partial commutation* or *independence*. With an empty independence relation, that is, without independence, the setting coincides with the classical theory of words or strings.

## Discussion

### Introduction

The analysis of sequential programs describes a run of a program as a sequence of atomic actions. On an abstract level such a sequence is simply a string in a free monoid over some (finite) alphabet of letters. This purely abstract viewpoint embeds program analysis into a rich theory of combinatorics on words and a theory of automata and formal languages. The approach has been very fruitful from the early days where the first compilers have been written until now where research groups in academia and industry develop formal methods for verification.

Efficient compilers use autoparallelization, which provides a natural example of independence of actions resulting in a partial commutation relation. For example, let  $a; b; c; a; d; e; f$  be a sequence of arithmetic operations where:

$$\begin{array}{lll} (a) \quad x := x + 2y, & (b) \quad x := x - z, & (c) \quad y := y \cdot 5z \\ (d) \quad w := 2w, & (e) \quad z := y \cdot z, & (f) \quad z := x + y \cdot w. \end{array}$$

A concurrent-read-exclusive-write protocol yields a list of pairs of independent operations  $(a, d)$ ,  $(a, e)$ ,  $(b, c)$ ,  $(b, d)$ ,  $(c, d)$ , and  $(d, e)$ , which can be performed concurrently or in any order. The sequence can therefore be performed in four parallel steps  $\{a\}; \{b, c\}; \{a, d, e\}; \{f\}$ , but as  $d$  commutes with  $a, b, c$  the result of  $a; b; c; a; d; e; f$  is equal to  $a; d; b; c; a; e; f$ , and two processors are actually enough to guarantee minimal parallel execution time, since another possible schedule is  $\{a, d\}; \{b, c\}; \{a, e\}; \{f\}$ . Trace theory yields a tool to do such (data-independent) transformations automatically.

---

## Trace Theory

VOLKER DIEKERT<sup>1</sup>, ANCA MUSCHOLL<sup>2</sup>

<sup>1</sup>Universität Stuttgart FMI, Stuttgart, Germany

<sup>2</sup>Université Bordeaux 1, Talence, France

### Synonyms

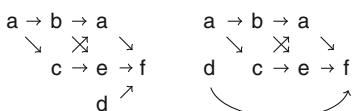
Partial computation; Theory of Mazurkiewicz-traces

### Definition

Trace Theory denotes a mathematical theory of free partially commutative monoids from the perspective of

Parallelism and concurrency demand for specific models, because a purely sequential description is neither accurate nor possible in all cases, for example, if asynchronous algorithms are studied and implemented. Several formalisms have been proposed in this context. Among these models there are Petri nets, Hoare's CSP and Milner's CCS, event structures, and branching temporal logics. The mathematical analysis of Petri nets is however quite complicated and much of the success of Hoare's and Milner's calculus is due to the fact that it stays close to the traditional concept of sequential systems relying on a unified and classical theory of words. Trace theory follows the same paradigm; it enriches the theory of words by a very restricted, but essential formalism to capture the main aspects of parallelism: In a static way a set  $I$  of independent letters  $(a, b)$  is fixed, and sequences are identified if they can be transformed into each other by using equations of type  $ab = ba$  for  $(a, b) \in I$ . In computer science this approach appeared for the first time in the paper by Keller on *Parallel Program Schemata and Maximal Parallelism* published in 1973. Based on the ideas of Keller and the behavior of elementary net systems, Mazurkiewicz introduced in 1977 the notion of *trace theory* and made its concept popular to a wider computer science community. Mazurkiewicz's approach relies on a graphical representation for a trace. This is a node-labeled directed acyclic graph, where arcs are defined by the dependence relation, which is by definition the complement of the independence relation  $I$ .

Thereby, a concurrent run has an immediate graphical visualization, which is obviously convenient for practice. The picture of the two parallel executions  $\{a\} ; \{b, c\}$ ;  $\{a, d, e\} ; \{f\}$  and  $\{a, d\} ; \{b, c\} ; \{a, e\} ; \{f\}$  can be depicted as follows, which represents (the Hasse diagrams of) isomorphic labeled partial orders:

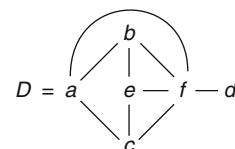


Moreover, the graphical representation yields immediately a correct notion of *infinite trace*, which is not clear when working with partial commutations. In the following years it became evident that trace theory indeed copes with some important phenomena such

as *true concurrency*. On the other hand it is still close to the classical theory of word languages describing sequential programs. In particular, it is possible to transfer the notion of finite sequential state control to the notion of asynchronous state control. This important result is due to Zielonka; it is one of the highlights of the theory. There is a satisfactory theory of recognizable languages relating finite monoids, rational operations, asynchronous automata, and logic. This leads to decidability results and various effective operations. Moreover, it is possible to develop a theory of asynchronous Büchi automata, which enables in trace theory the classical automata theory-based approach to automated verification.

## Mathematical Definitions and Normal Forms

Trace theory is founded on a rigorous mathematical approach. The underlying combinatorics for partial commutation were studied in mathematics already in 1969 in the seminal Lecture Notes in Mathematics *Problèmes combinatoires de commutation et réarrangements* by Cartier and Foata. The mathematical setting uses a finite alphabet  $\Sigma$  of letters and the specification of a symmetric and irreflexive relation  $I \subseteq \Sigma \times \Sigma$ , called the *independence* relation. Conveniently, its complement  $D = \Sigma \times \Sigma \setminus I$  is called the *dependence* relation. The dependence relation has a direct interpretation as graph as well. For the dependency used in the first example above it looks as follows:



The intended semantics is that independent letters commute, but dependent letters must be ordered. Taking  $ab = ba$  with  $(a, b) \in I$  as defining relations one obtains a quotient monoid  $M(\Sigma, I)$ , which has been called *free partial commutative monoid* or simply *trace monoid* in the literature. The elements are finite (Mazurkiewicz-)traces. For  $I = \emptyset$ , traces are just words in  $\Sigma^*$ ; for a full independence relation, that is,  $D = \text{id}_\Sigma$ , traces are vectors in some  $\mathbb{N}^k$ , hence Parikh-images of words. The general philosophy is that the extrema  $\Sigma^*$  and  $\mathbb{N}^k$  are

well understood (which is far from being true), but the interesting and difficult problems arise when  $\mathbb{M}(\Sigma, I)$  is neither free nor commutative.

For effective computations and the design of algorithms appropriate normal forms can be used. For the *lexicographic* normal form it is assumed that the alphabet  $\Sigma$  is totally ordered, say  $a < b < c < \dots < z$ . This defines a lexicographic ordering on  $\Sigma^*$  exactly the same way words are ordered in a standard dictionary. The lexicographic normal form of a trace is the minimal word in  $\Sigma^*$  representing it. For example, if  $I$  is given by  $\{(a, d), (d, a), (b, c), (c, a)\}$ , then the trace defined by the sequence *badacb* is the congruence class of six words:

$$\{baadbc, badabc, bdaabc, baadcb, badacb, bdaacb\}.$$

Its lexicographic normal form is the first word *baadbc*. An important property of lexicographic normal forms has been stated by Anisimov and Knuth. A word is in lexicographic normal form if and only if it does not contain a *forbidden pattern*, which is a factor *bua* where  $a < b \in \Sigma$  and the letter  $a$  commutes with all letters appearing in  $bu \in \Sigma^*$ . As a consequence, the set of lexicographic normal forms is a regular language.

The other main normal is due to Foata. It is a normal form that encodes a maximal parallel execution. Its definition uses *steps*, where a step means here a subset  $F \subseteq \Sigma$  of pairwise independent letters. Thus, a step requires only one parallel execution step. A step  $F$  yields a trace by taking the product  $\Pi_{a \in F} a$  over all its letters in any order. The *Foata normal form* is a sequence of steps  $F_1 \dots F_k$  such that  $F_1, \dots, F_k$  are chosen from left to right with maximal cardinality. The sequence  $\{a, d\}; \{b, c\}; \{a, e\}; \{f\}$  above has been the Foata normal form of *abcdef*.

The graphical representation of a trace due to Mazurkiewicz can be viewed as a third normal form. It is called the *dependence graph representation*; and it is closely related to the Foata normal form. Say a trace  $t$  is specified by some sequence of letters  $t = a_1 \dots a_n$ . Each index  $i \in V = \{1, \dots, n\}$  is labeled by the letter  $a_i$ . Finally, arcs  $(i, j) \in E$  are introduced if and only if both  $(a_i, a_j) \in D$  and  $i < j$ . In this way an acyclic directed graph  $G(t)$  is defined which is another unique representation of  $t$ . The information about  $t$  is also contained in the induced partial order (i.e., the transitive closure

of  $G(t)$ ) or in its Hasse-diagram (i.e., removing all transitive arcs from  $G(t)$ ).

### Computation of Normal Forms

There are efficient algorithms that compute normal forms in polynomial time. A very simple method uses a stack for each letter of the alphabet  $\Sigma$ . An input word is scanned from right to left, so the last letter is read first. When processing a letter  $a$  it is pushed on its stack and a marker is pushed on the stack of all the letters  $b$  ( $b \neq a$ ), which do not commute with  $a$ . Once the word has been processed its lexicographic normal form, the Foata normal form, and the Hasse-diagram of the dependence graph representation can be obtained straightforwardly. For example, the sequence  $a; b; c; a; d; e; f$  (with a dependence relation as depicted above) yields stacks as follows:

|          |          |          |          |          |          |          |
|----------|----------|----------|----------|----------|----------|----------|
|          |          |          |          |          |          | *        |
| <i>a</i> | *        | *        |          |          | *        | *        |
| *        | <i>b</i> | <i>c</i> |          |          | *        | *        |
| *        | *        | *        |          |          | *        | *        |
| <i>a</i> | *        | *        | <i>d</i> | <i>e</i> | *        | *        |
| *        | *        | *        | *        | *        | *        | <i>f</i> |
| <i>a</i> | <i>b</i> | <i>c</i> | <i>d</i> | <i>e</i> | <i>f</i> |          |

### Regular Sets

A fundamental concept in formal languages is the notion of a *regular set*. Kleene's Theorem says that a regular set can be specified either by a finite deterministic (resp. nondeterministic) automaton DFA (resp. NFA) or, equivalently, by a regular expression. Regular expressions are also called *rational expressions*. They are defined inductively by saying that every finite set denotes a rational expression and if  $R, S$  is rational, then  $R \cup S$ ,  $R \cdot S$ , and  $R^*$  are rational expressions, too. The semantics of a rational expression is defined in any monoid  $M$  since the semantics of  $R \cup S$ ,  $R \cdot S$  is obvious, and  $R^*$  can be viewed as the union  $\bigcup_{k \in \mathbb{N}} R^k$ . For *star-free* expressions one does not allow the star-operation, but one adds complementation, denoted, for example, by  $\overline{R}$  with the semantics  $M \setminus R$ .

In trace theory a direct translation of Kleene's Theorem fails, but it can be replaced by a generalization due to Ochmański. If  $(a, b)$  is a pair of independent letters, then  $(ab)^*$  is a rational expression, but due to  $ab = ba$  it represents all strings with an equal number

of  $a$ 's and  $b$ 's which is clearly not regular. With three pairwise independent letters  $(abc)^*$  is not even context-free. A general formal language theory distinguishes between recognizable and rational sets. A subset  $L$  of a trace monoid is called *recognizable*, if its closure is a regular word language. Here the closure refers to all words in  $\Sigma^*$ , which represent some trace in  $L$ . A subset  $L$  is called *rational*, if  $L$  can be specified by some regular (and hence rational) expression. Using the algebraic notion of homomorphism this can be rephrased as follows. Let  $\varphi$  be the canonical homomorphism of  $\Sigma^*$  onto  $\mathbb{M}(\Sigma, I)$ , which simply means the interpretation of a string as its trace. Now,  $L$  is recognizable if and only if  $\varphi^{-1}(L)$  is a regular word language, and  $L$  is rational if and only if  $L = \varphi(K)$  for some regular word language  $K$ . As a consequence of Kleene's Theorem all recognizable trace languages are rational, but the converse fails as soon as there is a pair of independent letters, that is, the trace monoid is not free.

Given a recognizable trace language  $L$ , the corresponding word language  $\varphi^{-1}(L)$  is accepted by some NFA (actually some DFA), which satisfies the so-called *I-diamond property*. This means whenever it holds  $(a, b) \in I$  and a state  $p$  leads to a state  $q$  by reading the word  $ab$ , then it is in state  $p$  also possible to read  $ba$  and this leads to state  $q$ , too. NFAs satisfying the *I-diamond property* accept closed languages only. Therefore they capture exactly the notion of recognizability for traces.

It has been shown that the concatenation of two recognizable trace languages is recognizable, in particular *star-free languages* (i.e., given by star-free expressions) are recognizable. However, the example  $(ab)^*$  above shows that the star-operation leads to non-recognizable sets as soon as the trace monoid is not free. M茅tivier and Ochma艅ski have introduced a restricted version where the star-operation is allowed only when applied to languages  $L$  where all traces  $t \in L$  are connected. This means the dependence graph  $G(t)$  is connected or, equivalently, there is no nontrivial factorization  $t = uv$  where all letters in  $u$  are independent of all letters in  $v$ . A theorem shows that  $L^*$  is still recognizable, if  $L$  is connected (i.e., all  $t \in L$  are connected) and recognizable. Ochma艅ski's Theorem yields also the converse: A trace language  $L$  is recognizable if and only if it can be specified by a rational expression where the star-operation is restricted to connected subsets. As word languages are always connected this is a proper generalization of the

classical Kleene's Theorem. Yet another characterization of recognizable trace languages is as follows: They are in one-to-one correspondence with regular subsets inside the regular set  $\text{LexNF} \subseteq \Sigma^*$  of lexicographic normal forms. The correspondence associates with  $L \subseteq \mathbb{M}(\Sigma, I)$  the set  $K = \varphi^{-1}(L) \cap \text{LexNF}$ . A rational expression for  $K$  is a rational expression for  $L$ , where the star-operation is restricted to connected languages.

## Decidability Questions

### The Star Mystery

The *Star Problem* is to decide for a given recognizable trace language  $L \subseteq \mathbb{M}(\Sigma, I)$  whether  $L^*$  is recognizable. It is not known whether the star problem is decidable, even if it is restricted to finite languages  $L$ . The surprising difficulty of this problem has been coined as the *star mystery* by Ochma艅ski. It has been shown by Richomme that the Star Problem is decidable, if  $(\Sigma, I)$  does not contain any  $C_4$  (cycle of four letters) as an induced subgraph.

### Undecidability Results for Rational Sets

For rational languages (unlike as for recognizable languages) some very basic problems are known to be undecidable. The following list contains undecidable decision problems, where the input for each instance consists of an independence alphabet  $(\Sigma, I)$  and rational trace languages  $R, T \subseteq \mathbb{M}(\Sigma, I)$  specified by rational expressions.

- **Inclusion** question: Does  $R \subseteq T$  hold?
- **Equality** question: Does  $R = T$  hold?
- **Universality** question: Does  $R = \mathbb{M}(\Sigma, I)$  hold?
- **Complementation** question: Is  $\mathbb{M}(\Sigma, I) \setminus R$  a rational?
- **Recognizability** question: Is  $R$  recognizable?
- **Intersection** question: Does  $R \cap T = \emptyset$  hold?

On the positive side, if  $I$  is transitive, then all six problems above are decidable. This is also a necessary condition for the first five problems in the list. Transitivity of the independence alphabet means in algebraic terms that the trace monoid is a free product of free and free commutative monoids, like, for example,  $\{a, b\}^* * \mathbb{N}^3$ .

The intersection problem is simpler. It is known that the problem Intersection is decidable if and only if  $(\Sigma, I)$

is a transitive forest. It is also well known that transitive forests are characterized by forbidden induced subgraphs  $C_4$  and  $P_4$  (cycle and path, resp., of four letters).

### Asynchronous Automata

Whereas recognizable trace languages can be defined as word languages accepted by DFAs or NFAs with  $I$ -diamond property, there is an equivalent distributed automaton model called *asynchronous automata*. Such an automaton is a parallel composition of finite-state processes synchronizing over shared variables, whereas a DFA satisfying the  $I$ -diamond property is still a device with a centralized control. An asynchronous automaton  $\mathcal{A}$  has, by definition, a distributed finite state control such that independent actions may be performed in parallel. The set of global states is modeled as a direct product  $Q = \prod_{p \in P} Q_p$ , where the  $Q_p$  are states of the local component  $p \in P$  and  $P$  is some finite index set (a set of processors). For each letter  $a \in \Sigma$  there is a *read domain*  $R(a) \subseteq P$  and a *write domain*  $W(a) \subseteq P$  where for simplicity  $W(a) \subseteq R(a)$ . Processors  $p$  and  $q$  share a variable  $a$  if and only if  $p, q \in R(a)$ . The transitions are given by a family of partially defined functions  $\delta_p$ , where each processor  $p$  reads the status in the local components of its read domain and changes states in local components of its write domain. Accordingly to the read-and-write-conflicts being allowed, four basic types are distinguished:

- Concurrent-Read-Exclusive-Write (*CREW*),  
if  $R(a) \cap W(b) = \emptyset$  for all  $(a, b) \in I$ .
- Concurrent-Read-Owner-Write (*CROW*),  
if  $R(a) \cap W(b) = \emptyset$  for all  $(a, b) \in I$  and  $W(a) \cap W(b) = \emptyset$  for all  $a \neq b$ .
- Exclusive-Read-Exclusive-Write (*EREW*),  
if  $R(a) \cap R(b) = \emptyset$  for all  $(a, b) \in I$ .
- Exclusive-Read-Owner-Write (*EROW*),  
if  $R(a) \cap R(b) = \emptyset$  for all  $(a, b) \in I$  and  $W(a) \cap W(b) = \emptyset$  for all  $a \neq b$ .

The local transition functions  $(\delta_p)_{p \in P}$  give rise to a partially defined transition function on global states  $\delta : (\prod_{p \in P} Q_p) \times \Sigma \rightarrow \prod_{p \in P} Q_p$ .

If  $\mathcal{A}$  is of any of the four types above, then the action of a trace  $t \in \mathbb{M}(\Sigma, I)$  on global states is well defined. This allows to see an asynchronous automaton as an  $I$ -diamond DFA. There are effective translations from

one model to the other. The most compact versions can be obtained by a CREW model, therefore it is of prior practical interest.

Zielonka has shown in his thesis (published in 1987) the following deep theorem in trace theory: Every recognizable trace language can be accepted by some finite asynchronous automaton. The proof of this theorem is very technical and complicated. Moreover, the original construction was doubly exponential in the size of an  $I$ -diamond automaton for the language  $L$ . Therefore, it is part of ongoing research to simplify its construction, in particular since efficient constructions are necessary to make the result applicable in practice. The best result to date is due to Genest et al. They provide a construction where the size of the obtained asynchronous automaton is polynomial in the size of a given DFA and simply exponential in the number of processes. They also show that the construction is optimal within the class of automata produced by Zielonka-type constructions, which yields a nontrivial lower bound on the size of asynchronous automata.

A rather direct construction of asynchronous automata is known for triangulated dependence alphabets, which means that all chordless cycles are of length 3. For example, complete graphs and forests are triangulated.

### Infinite Traces

The theory of infinite traces has its origins in the mid-1980s when Flé and Roucairol considered the problem of serializability of iterated transactions in data bases. A suitable definition of an infinite trace uses the dependence graph representation due to Mazurkiewicz. Just as in the finite case an infinite sequence  $t = a_1 a_2 \dots$  of letters yields an infinite node-labeled acyclic directed graph  $G(t)$ , where now each  $i \in V = \mathbb{N}$  is labeled by the letter  $a_i$ , and again arcs  $(i, j) \in E$  are introduced if and only if both  $(a_i, a_j) \in D$  and  $i < j$ . It is useful to consider finite and infinite objects simultaneously as an infinite trace may split into connected components where some of them might be finite. The notion of *real trace* has been introduced to denote either a finite or an infinite trace. If  $t_1, t_2, \dots$  is (finite or infinite) sequence of finite traces, then the product  $t_1 t_2 \dots$  is a well-defined real trace. It is a finite trace if almost all  $t_i$  are empty and an infinite trace otherwise. In particular, one can define the  $\omega$ -product  $L^\omega$  for every set  $L$  of finite traces and one enriches the set of rational expressions by this operation.

The set  $\mathbb{R}(\Sigma, I)$  of real traces can be embedded into a monoid of *complex traces* where the *imaginary* component is a subset of  $\Sigma$ . This alphabetic information is necessary in order to define an associative operation of concatenation. (Over complex traces  $L^\omega$  is defined for all subsets  $L$ .)

Many results from the theory of finite traces transfer to infinite traces according to the same scheme as for finite and infinite words.

## Logics

### MSO and First-Order Logic

Formulae in monadic second-order logic (MSO) are built up upon first-order variables  $x, y, \dots$  ranging over vertices and second-order variables  $X, Y, \dots$  ranging over subsets of vertices. There are Boolean constants *true* and *false*, the logical connectives  $\vee, \wedge, \neg$ , and quantification  $\exists, \forall$  for the first- and second-order variables. In addition there are four types of atomic formulae:

$$x \in X, x = y, (x, y) \in E, \text{ and } \lambda(x) = a.$$

A first-order formula is a formula without any second-order variable. A *sentence* is a closed formula, that is, a formula without free variables. The semantics of an MSO-sentence is defined for every node-labeled graph  $[V, E, \lambda]$  (here:  $V$  = set of vertices,  $E$  = set of edges,  $\lambda : V \rightarrow \Sigma$  = vertex labeling). Identifying a trace  $t$  with its dependence graph  $G(t)$ , the truth value of  $t \models \psi$  is therefore well defined for every sentence  $\psi$ . The trace language defined by a sentence  $\psi$  is  $L(\psi) = \{t \in \mathbb{R}(\Sigma, I) \mid t \models \psi\}$ . It follows a notion of first-order and second-order definability of trace languages.

### Temporal Logic

Linear temporal logic, LTL, can be inductively defined inside first order as formulae with one free variable, as soon as the transitive closure  $(x, y) \in E^*$  is expressible in first order (as it is the case for trace monoids). There are no quantifiers, but all Boolean connectives. The atomic formulae are  $\lambda(x) = a$ . If  $\varphi(x), \psi(x)$  are LTL-formulae, then  $\text{EX } \varphi(x)$  and  $(\varphi \cup \psi)(x)$  are LTL-formulae. In temporal logic  $(x, y) \in E^*$  means that  $y$  is in the future of the node  $x$ . The semantics of  $\text{EX } \varphi(x)$  is *exists next*, thus  $\varphi(y)$  holds for a direct successor of  $x$ . The semantics of  $(\varphi \cup \psi)(x)$  reflects an *until operator*, it says that in the future of  $x$  there is some  $z$  that satisfies  $\psi(z)$  and all  $y$  in the future of  $x$  but in the strict

past of  $z$  satisfy  $\varphi(y)$ . Hence, condition  $\varphi$  holds until  $\psi$  becomes true. There are dual past-tense operators, but they do not add expressivity.

For LTL one can also give a syntax without any free variable and a *global semantics* where the evaluation is based on the prefix relation of traces. The local semantics as defined above is for traces a priori expressively weaker, but it was shown that both, the global and local LTL have the same expressive power as first-order logic. This was done by Thiagarajan and Walukiewicz in 1998 for global LTL and by Diekert and Gastin in 2006 for local LTL, respectively. Both results extend a famous result of Kamp from words to traces. The complexity of the satisfiability problem (or model checking) is however quite different. In global semantics it is nonelementary, whereas in local semantics it is in PSPACE (= class of problems solvable on a Turing machine in polynomial space.)

### Fragments

For various applications fragments of first-order logics suffice. This has the advantage that simpler constructions are possible and that the complexity of model checking is possibly reduced. A prominent fragment is first-order logic with at most two names for variables. Two-variable logics capture the core features of XML navigational languages like XPath. Over words and over traces two variable logic  $\text{FO}^2[E]$  can be characterized algebraically via the variety of monoids DA (referring to the fact that regular  $\mathcal{D}$ -classes are aperiodic semigroups), in logic by *Next-Future* and *Yesterday-Past* operators, and in terms of rational expressions via unambiguous polynomials. It turns out that the satisfiability problem for two-variable logic is NP-complete (if the independence alphabet is not part of the input). The extension of these results from words to traces is due to Kufleitner.

### Logics, Algebra, and Automata

The connection between logic and recognizability uses algebraic tools from the theory of finite monoids. If  $h : \mathbb{M}(\Sigma, I) \rightarrow M$  is a homomorphism to a finite monoid  $M$  and  $L \subseteq \mathbb{R}(\Sigma, I)$  is a set of real traces, then one says that  $h$  recognizes  $L$ , if for all  $t \in L$  and factorizations  $t = t_1 t_2 \dots$  into finite traces  $t_i$  the following inclusion holds:  $h^{-1}(t_1)h^{-1}(t_2) \dots \subseteq L$ . This allows to speak of

*aperiodic languages* if some recognizing monoid is aperiodic. A monoid  $M$  is *aperiodic*, if for all  $x \in M$  there is some  $n \in \mathbb{N}$  such that  $x^{n+1} = x^n$ . A deep result states that a language is first-order definable if and only if it is recognized by a homomorphism to a finite aperiodic monoid. Algebraic characterizations lead to decidability of fragments. For example, it is decidable whether a recognizable language is aperiodic or whether it can be expressed in two-variable first-order logic.

Another way to define recognizability is via Büchi automata. A Büchi automaton for real traces is an  $I$ -diamond NFA with a set of final states  $F$  and a set of repeated states  $R$ . It accepts a trace if the run stops in  $F$  or if repeated states are visited infinitely often. If its transformation monoid is aperiodic it is called aperiodic, too. There is also a notion of asynchronous (cellular) Büchi automaton, and it is known that every  $I$ -diamond Büchi automaton can be transformed into an equivalent asynchronous cellular Büchi automaton.

The main result connecting logic, recognizability, rational expressions, and algebra can be summarized by saying that the following statements in the first block (second block resp.) are equivalent for all trace languages  $L \subseteq \mathbb{R}(\Sigma, I)$ :

#### MSO definability:

1.  $L$  is definable in monadic second-order logic.
2.  $L$  is recognizable by some finite monoid.
3.  $L$  is given as a rational expression where the star is restricted to connected languages.
4.  $L$  is accepted by some asynchronous Büchi automaton.

#### First-order definability:

1.  $L$  is definable in first-order logic.
2.  $L$  is definable in LTL (with global or local semantics).
3.  $L$  is recognizable by some finite and aperiodic monoid.
4.  $L$  is star-free.

#### Automata-Based Verification

The automata theoretical approach to verification uses the fact that systems and specifications are both modeled with finite automata. More precisely, a system is given as a finite transition system  $\mathcal{A}$ , which is typically realized as an NFA without final states. So, the system

allows finite and infinite runs. The specification is written in some logical formalism, say in the linear temporal logic LTL. So the specification is given by some formula  $\varphi$ , and its semantics  $L(\varphi)$  defines the runs that obey the specification. Model checking means to verify the inclusion  $L(\mathcal{A}) \subseteq L(\varphi)$ . This is equivalent to  $L(\mathcal{A}) \cap L(\neg\varphi) = \emptyset$ . Once an automaton  $\mathcal{B}$  with  $L(\mathcal{B}) = L(\neg\varphi)$  has been constructed, standard methods yield a product automaton for  $L(\mathcal{A}) \cap L(\mathcal{B})$ . The check for emptiness becomes a reachability problem in directed graphs.

A main obstacle is the combinatorial explosion when constructing the automaton  $\mathcal{B}$ . But this works in practice nevertheless reasonable well, because typical specifications are simple enough to be understood (hopefully) by the designer, so they are short. From a theoretical viewpoint the complexity of model checking for MSO and first order is nonelementary, but for (local) LTL is still in PSPACE. This approach is mostly applied and very successful where runs can be modeled as sequences. Trace theory provides the necessary tools to extend these methods to asynchronous systems. A first step in this direction has been implemented in the framework of *partial order* reduction. Another application of trace theory is the analysis of communication protocols.

#### Traces and Asynchronous Communication

Trace automata like asynchronous ones model concurrency in the same spirit as Petri nets, using shared variables. A more complex model arises when concurrent processes cooperate over unbounded, fifo communication channels.

A *communicating automaton* is defined over a set  $P$  of processes, together with point-to-point communication channels  $Ch \subseteq \{(p, q) \in P^2 \mid p \neq q\}$ . It consists of a tuple of NFAs  $\mathcal{A}_p$ , one for each process  $p \in P$ . Each NFA  $\mathcal{A}_p$  has a set of local states  $Q_p$  and transition relation  $\delta_p \subseteq Q_p \times \Sigma_p \times Q_p$ . The set  $\Sigma_p$  of local actions of process  $p$  consists of send-actions  $p!q(m)$  (of message  $m$  to process  $q$ ,  $(p, q) \in Ch$ ) and receive-actions  $p?r(m)$  (of message  $m$  from process  $r$ ,  $(r, p) \in Ch$ ), respectively. The semantics of such an automaton is defined through configurations consisting of a tuple of local states (one for each process) and a tuple of word contents (one for each channel). In terms of partial orders the semantics of runs corresponds to *message sequence charts*.

(MSCs), a graphical notation for fifo message exchange. In contrast with asynchronous automata, communicating automata have an infinite state space and are actually Turing powerful; thus, most algorithmic questions about them are undecidable.

The theory of recognizable trace languages enjoys various nice results known from word languages, for example, in terms of logics and automata. Since communicating automata are Turing powerful, one needs restrictions in order to obtain, for example, logical characterizations. A natural restriction consists in imposing bounds on the size of the channels. Such bounds come in two versions, namely, as *universal* and *existential* bounds, respectively. The existential version of channel bounds is optimistic and considers all those runs that can be rescheduled on bounded channels. The universal version is pessimistic and considers only those runs that, independent of the scheduling, can be executed with bounded channels. Thus, communicating automata with an universal channel bound are finite state, whereas with an existential channel bound they are infinite state systems.

Kuske proposed an encoding of runs of communicating automata with bounded channels into trace languages. Using this encoding, the set of runs (MSCs) of a communicating automaton is the projection of a recognizable trace language (for a universal bound), respectively the set of MSCs generated by the projection of a recognizable trace language (for an existential bound). This correspondence has the same flavor as the distinction between recognizable and rational trace languages, respectively.

The logic MSO over MSCs is defined with an additional binary message-predicate relating matching send and receive events. Henriksen et al. and Genest et al., respectively, have shown that the equivalence between MSO and automata extends to communicating automata with universal and existential channel bound, respectively. Another equivalent characterization exists in terms of MSC-graphs, similar to star-connected expressions for trace languages. These expressiveness results are complemented by decidable instances of the model-checking problem.

## Related Entries

- ▶ [Asynchronous Iterative Algorithms](#)
- ▶ [CSP \(Communicating Sequential Processes\)](#)

- ▶ [Formal Methods-Based Tools for Race, Deadlock, and Other Errors](#)
- ▶ [Multi-Threaded Processors](#)
- ▶ [Parallel Computing](#)
- ▶ [Parallelization, Automatic](#)
- ▶ [Peer-to-Peer](#)
- ▶ [Petri Nets](#)
- ▶ [Reordering](#)
- ▶ [Synchronization](#)
- ▶ [Trace Scheduling](#)
- ▶ [Verification of Parallel Shared-Memory Programs, Owicki-Gries Method of Axiomatic](#)

## Bibliographic Notes and Further Reading

Trace theory has its origin in enumerative combinatorics when Cartier and Foata found a new proof of the MacMahon Master Theorem in the framework of partial commutation by combining algebraic and bijective ideas [2]. The Foata normal form was defined in this Lecture Note. In computer science the key idea to use partial commutation as tool to investigate parallel systems was laid by Keller [10], but it was only by the influence of the technical report of Mazurkiewicz [11] when these ideas were spread to a wider computer science community, in particular to the Petri-net community. It was also Mazurkiewicz who coined the notion *Trace theory* and who introduced the notion of dependence graphs as a visualization of traces. The characterization of lexicographic normal forms by forbidden pattern is due to Anisimov and Knuth [1].

The investigation of recognizable (regular, rational resp.) languages is central in the theory of traces. The characterization of recognizable languages in terms of star-connected regular expressions is due to Ochmański [13]. The notion of *asynchronous automaton* is due to Zielonka. The major theorem showing that all recognizable languages can be accepted by asynchronous automata is his work (built on his thesis) [15]. The research on asynchronous automata is still an important and active area. The best constructions so far are due to Genest et al., where also nontrivial lower bounds were established [8].

The theory of infinite traces has its origin in the mid-1980s. A definition of a real trace as a prefix-closed and directed subset of real traces and its characterization by dependence graphs is given in a survey by

Mazurkiewicz [12]. The theory of recognizable real trace languages has been initiated by Gastin in 1990. The generalization of the Kleene–Büchi–Ochmański Theorem to real traces is due to Gastin, Petit, and Zielonka [7]. Diekert and Muscholl gave a construction for deterministic asynchronous Muller automata accepting a given recognizable real trace language.

Ebinger initiated the study of LTL for traces in his thesis in 1994. But it took quite an effort until Diekert and Gastin were able to show that LTL (in local semantics) has the same expressive power as first-order logic [3]. The advantage of a local LTL is that model checking in PSPACE, whereas in its global semantics it becomes nonelementary by a result of Walukiewicz [14]. The PSPACE-containment has been shown for a much wider class of logics by Gastin and Kuske [6]. Diekert, Horsch, and Kufleitner [4] give a survey on fragments of first-order logic in trace theory. The Büchi-like equivalence between automata and MSO for existentially bounded communicating automata has been shown by Genest, Kuske, and Muscholl [9]. The translation from MSO into automata uses the equivalence for trace languages, but needs some additional, quite technical construction specific to communicating automata.

Very much of the material used in the present discussion can be found in *The Book of Traces*, which was edited by Diekert and Rozenberg [5]. The book surveys also a notion of *semi-commutation* (introduced by Clerbout and Latteux), and it provides many hints for further reading. Current research efforts concentrate on the topic of distributed games and controller synthesis for asynchronous automata.

## Bibliography

1. Anisimov AV, Knuth DE (1979) Inhomogeneous sorting. *Int J Comput Inf Sci* 8:255–260
2. Cartier P, Foata D (1969) Problèmes combinatoires de commutation et réarrangements. Lecture notes in mathematics, vol 85. Springer, Heidelberg
3. Diekert V, Gastin P (2006) Pure future local temporal logics are expressively complete for Mazurkiewicz traces. *Inf Comput* 204:1597–1619. Conference version in LATIN 2004, LNCS 2976: 170–182, 2004
4. Diekert V, Horsch M, Kufleitner M (2007) On first-order fragments for Mazurkiewicz traces. *Fundamenta Informaticae* 80:1–29
5. Diekert V, Rozenberg G (eds) (1995) *The book of traces*. World Scientific, Singapore
6. Gastin P, Kuske D (2007) Uniform satisfiability in pspace for local temporal logics over Mazurkiewicz traces. *Fundam Inf* 80(1–3): 169–197

7. Gastin P, Petit A, Zielonka WL (2007) An extension of Kleene's and Ochmański's theorems to infinite traces. *Theoret Comput Sci* 125:167–204, x
8. Genest B, Gimbert H, Muscholl A, Walukiewicz I (2010) Optimal Zielonka-type construction of deterministic asynchronous automata. In: Abramsky S, Gavoille C, Kirchner C, Meyer auf der Heide F, Spirakis PG (eds) ICALP (2). Lecture notes in computer science, vol 6199. Springer, pp 52–63
9. Genest B, Kuske D, Muscholl A (2006) A Kleene theorem and model checking algorithms for existentially bounded communicating automata. *Inf Comput* 204:926–956. <http://dx.doi.org/10.1016/j.ic.2006.01.005>; DBLP, <http://dblp.uni-trier.de>
10. Keller RM (1973) Parallel program schemata and maximal parallelism I. Fundamental results. *J Assoc Comput Mach* 20(3):514–537
11. Mazurkiewicz A (1977) Concurrent program schemes and their interpretations. DAIMI Rep. PB 78, Aarhus University, Aarhus
12. Mazurkiewicz A (1987) Trace theory. In: Brauer W et al. (eds) Petri nets, applications and relationship to other models of concurrency. Lecture notes in computer science, vol 255. Springer, Heidelberg, pp 279–324
13. Ochmański E (Oct 1985) Regular behaviour of concurrent systems. *Bull Eur Assoc Theor Comput Sci (EATCS)* 27:56–67
14. Walukiewicz I (1998) Difficult configurations – on the complexity of LTrL. In: Larsen KG, et al. (eds) Proceedings of the 25th International Colloquium Automata, Languages and Programming (ICALP'98), Aalborg (Denmark). Lecture notes in computer science, vol 1443. Springer, Heidelberg, pp 140–151
15. Zielonka WL (1987) Notes on finite asynchronous automata. R.A.I.R.O. Informatique Théorique et Applications 21:99–135

## Tracing

- [Performance Analysis Tools](#)
- [Scalasca](#)
- [TAU](#)

## Transactional Memories

T

MAURICE HERLIHY  
Brown University, Providence, RI, USA

## Synonyms

[Locks](#); [Monitors](#); [Multiprocessor synchronization](#)

## Introduction

Transactional memory (TM) is an approach to structuring concurrent programs that seeks to provide better scalability and ease-of-use than conventional approaches based on locks and conditions. The term is

commonly used to refer to ideas that range from programming language constructs to hardware architecture. This entry will survey how transactional memory affects each of these domains.

The major chip manufacturers have, for the time being, given up trying to make processors run faster. Moore's law has not been repealed: Each year, more and more transistors fit into the same space, but their clock speed cannot be increased without overheating. Instead, attention has turned toward *chip multiprocesssing* (CMP), in which multiple computing cores are included on each processor chip. In the medium term, advances in technology will provide increased parallelism, but not increased single-thread performance. As a result, system designers and software engineers can no longer rely on increasing clock speed to hide software bloat. Instead, they must learn to make more effective use of increasing parallelism.

This adaptation will not be easy. Conventional programming practices typically rely on combinations of locks and conditions, such as monitors [1], to prevent threads from concurrently accessing shared data. Locking makes concurrent programming possible because it allows programmers to reason about certain code sections as if they were executed atomically. Nevertheless, the conventional approach suffers from a number of shortcomings.

First, programmers must decide between *coarse-grained* locking, in which a large data structure is protected by a single lock, and *fine-grained* locking, in which a lock is associated with each component of the data structure. Coarse-grained locking is relatively easy to use, but permits little or no concurrency, thereby preventing the program from exploiting multiple cores. By contrast, fine-grained locking is substantially more complicated because of the need to ensure that threads acquire all necessary locks (and only those, for good performance), and because of the need to avoid deadlock when acquiring multiple locks. Such designs are further complicated because the most efficient engineering solution may be platform dependent, varying with different machine sizes, workloads, and so on, making it difficult to write code that is both scalable and portable.

Second, locking provides poor support for code composition and reuse. For example, consider a lock-based queue that provides atomic `enq()` and `deq()`

methods. Ideally, it should be easy to transfer an item atomically from one queue to another, but such elementary composition simply does not work. It is necessary to lock both queues at the same time to make the transfer atomic. If the queue methods synchronize internally, then there is no way to acquire and hold both locks simultaneously. If the queues export their locks, then modularity and safety are compromised, because the integrity of the objects depends on whether their users follow *ad hoc* conventions correctly.

Finally, such basic issues as the mapping from locks to data, that is, which locks protect which data, and the order in which locks must be acquired and released, are all based on convention, and violations are notoriously difficult to detect and debug. For these and other reasons, today's software practices make concurrent programs too difficult to develop, debug, understand, and maintain.

## The Transactional Model

A *transaction* is a sequence of steps executed by a single thread. Transactions are *atomic*: Each transaction either commits (it takes effect) or aborts (its effects are discarded). Transactions are *linearizable* [2]: They appear to take effect in a one-at-a-time order. Transactional memory supports a computational model in which each thread announces the start of a transaction, executes a sequence of operations on shared objects, and then tries to commit the transaction. If the commit succeeds, the transaction's operations take effect; otherwise, they are discarded.

Sometimes we refer to these transactions as *memory transactions*. Memory transactions satisfy the same formal serializability and atomicity properties as the transactions used in conventional database systems, but they are intended to address different problems.

Unlike database transactions, memory transactions are short-lived activities that access a relatively small number of objects in primary memory. Database transactions are *persistent*: When a transaction commits, its changes are backed up on a disk. Memory transactions need not be persistent, and involve no explicit disk I/O.

To illustrate why memory transactions are attractive from a software engineering perspective, consider the problem of constructing a concurrent FIFO queue that permits one thread to enqueue items at the tail of

the queue at the same time another thread dequeues items from the head of the queue, at least while the queue is non-empty. Any problem so easy to state, and that arises so naturally in practice, should have an easily devised, understandable solution. In fact, solving this problem with locks is quite difficult. In 1996, Michael and Scott published a clever and subtle solution [3]. It speaks poorly for fine-grained locking as a methodology that solutions to such simple problems are challenging enough to be publishable.

By contrast, it is almost trivial to solve this problem using transactions. [Figure 1](#) shows how the queue's enqueue method might look in a language that provides direct support for transactions. It consists of little more than enclosing sequential code in a transaction

```
class Queue<T> {
 QNode head;
 Qnode tail ;
 public void enq(T x) {
 atomic {
 Qnode q = new Qnode(x);
 if (tail == null) { // empty queue
 head = tail = q;
 } else {
 tail .next = q;
 tail = q;
 }
 }
 public T deq() {
 atomic {
 if (head == null)
 retry ;
 T item = head.item;
 head = head.next;
 if (head == null)
 tail = null;
 return item;
 }
 }
 ...
 }
}
```

**Transactional Memories. Fig. 1** Transactional queue code fragment

```
atomic {
 x = q0.deq();
} orElse {
 x = q1.deq();
}
```

**Transactional Memories. Fig. 2** The orElse statement: waiting on multiple conditions

block. In practice, of course, a complete implementation would include more details (such as how to respond to an empty queue), but even so, this concurrent queue implementation is a remarkable achievement: It is not, by itself, a publishable result.

Conditional synchronization can be accomplished in the transactional model by means of the `retry` construct [4]. As illustrated in [Fig. 1](#), if a thread attempts to dequeue from an empty queue, it executes `retry`, which rolls back the partial effects of the atomic block, and re-executes that block later when the object's state has changed. The `retry` construct is attractive because it is not subject to the *lost wake-up* bug that can arise using monitor conditions.

Transactions also admit compositions that would be impossible using locks and conditions. Waiting for one of several conditions to become *true* is impossible using objects with internal monitor condition variables. A novel aspect of `retry` is that such composition becomes easy. [Figure 2](#) shows a code snippet illustrating the `orElse` statement, which joins two or more code blocks. Here, the thread executes the first block. If that block calls `retry`, then that subtransaction is rolled back, and the thread executes the second block. If that block also calls `retry`, then the `orElse` as a whole pauses, and later reruns each of the blocks (when something changes) until one completes.

T

## Motivation

TM is commonly used to address three distinct problems: first, a simple desire to make highly concurrent data structures easy to implement; second, a more ambitious desire to support well-structured large-scale concurrent programs; and third, a pragmatic desire to make conventional locking more concurrent. Here is a survey of each area.

## Lock-Free Data Structures

A data structure is *lock-free* if it guarantees that infinitely often *some* method call finishes in a finite number of steps, even if some subset of the threads halt in arbitrary places. A data structure that relies on locking cannot be lock-free because a thread that acquires a lock and then halts can prevent non-faulty threads from making progress.

Lock-free data structures are often awkward to implement using today's architectures which typically rely on *compare-and-swap* for synchronization. The *compare-and-swap* instruction takes three arguments, and *address a*, an *expected* value *e*, and an *update* value *u*. If the value stored at *a* is equal to *e*, then it is atomically replaced with *u*, and otherwise it is unchanged. Either way, the instruction sets a flag indicating whether the value was changed.

Often, the most natural way to define a lock-free data structure is to make an atomic change to several fields. Unfortunately, because *compare-and-swap* allows only one word (or perhaps a small number of contiguous words) to be changed atomically, designers of lock-free data structures are forced to introduce complex multistep protocols or additional levels of indirection that create unwelcome overhead and conceptual complexity. The original TM paper [5] was primarily motivated by a desire to circumvent these restrictions.

## Software Engineering

TM is appealing as a way to help programmers structure concurrent programs because it allows the programmer to focus on what the program should be doing, rather than on the detailed synchronization mechanisms needed. For example, TM relieves the programmer of tasks such as devising specialized locking protocols for avoiding deadlocks, and conventions associating locks with data.

A number of programming languages and libraries have emerged to support TM. These include Clojure [6], .Net [7], Haskell [4], Java [8, 9], C++ [10], and others.

Several groups have reported experiences converting programs from locks to TM. The TxLinux [11] project replaced most of the locks in the Linux kernel with transactions. Syntactically, each transaction appears to be a lock-based critical section, but that code is executed speculatively as a transaction (see Section 3.3). If an I/O call is detected, the transaction is rolled

back and restarted using locks. Using transactions primarily as an alternative way to implement locks minimized the need to rewrite and restructure the original application.

Damron et al. [12] transactionalized the Berkeley DB lock manager. They found the transformation more difficult than expected because simply changing critical sections into atomic blocks often resulted in a disappointing level of concurrency. Critical sections often shared data unnecessarily, usually in the form of global statistics or shared memory pools. Later on, we will see other work that reinforces the notion the need to avoid *gratuitous conflicts* means that concurrent transactional programs must be structured differently than concurrent lock-based programs.

Pankratius et al. [13] conducted a user study where twelve students, working in pairs, wrote a parallel desktop search engine. Three randomly chosen groups used a compiler supporting TM, and three used conventional locks. The best TM group were much faster to produce a prototype, the final program performed substantially better, and they reported less time spent on debugging. However, the TM teams found performance harder to predict and to tune. Overall, the TM code was deemed easier to understand, but the TM teams did still make some synchronization errors.

Rossbach et al. [14] conducted a user study in which 147 undergraduates implemented the same programs using coarse-grained and fine-grained locks, monitors, and transactions. Many students reported they found transactions harder to use than coarse-grain locks, but slightly easier than fine-grained locks. Code inspection showed that students using transactions made many fewer synchronization errors: Over 70% of students made errors with fine-grained locking, while less than 10% made errors using transactions.

## Lock Elision

Transactions can also be used as a way to implement locking. In *lock elision* [15], when a thread requests a lock, rather than waiting to acquire that lock, the thread starts a speculative transaction. If the transaction commits, then the critical section is complete. If the transaction aborts because of a synchronization conflict, then the thread can either retry the transaction, or it can actually acquire the lock.

Here is why lock elision is attractive. Locking is conservative: A thread must acquire a lock if it *might* conflict with another thread, even if such conflicts are rare. Replacing lock acquisition with speculative execution enhances concurrency if actual conflicts are rare. If conflicts persist, the thread can abandon speculative execution and revert to using locks. Lock elision has the added advantage that it does not require code to be restructured. Indeed, it can often be made to work with legacy code.

Azul Systems [16] has a JVM that uses (hardware) lock elision for contended Java locks, with the goal of accelerating “dusty deck” Java programs. The run-time system keeps track of how well the hardware transactional memory (HTM) is doing, and decides when to use lock elision and when to use conventional locks. The results work well for some applications, modestly well for others, and poorly for a few. The principal limitation seems to be the same as observed by Damron et al. [12]: many critical sections are written in a way that introduces gratuitous conflicts, usually by updating performance counters. Although these are not real conflicts, the HTM has no way to tell. Rewriting such code can be effective, but requires abandoning the goal of speeding up “dusty deck” programs.

## Hardware Transactional Memory

Most hardware transactional memory (HTM) proposals are based on straightforward modifications to standard multiprocessor cache-coherence protocols. When a thread reads or writes a memory location on behalf of a transaction, that cache entry is flagged as being transactional. Transactional writes are accumulated in the cache or write buffer, but are not written back to memory while the transaction is active. If another thread invalidates a transactional entry, a data conflict has occurred, that transaction is aborted and restarted. If a transaction finishes without having had any of its entries invalidated, then the transaction commits by marking its transactional entries as valid or as dirty, and allowing the dirty entries to be written back to memory in the usual way.

One limitation of HTM is that in-cache transactions are limited in size and scope. Most hardware transactional memory proposals require programmers to be aware of platform-specific resource limitations such as cache and buffer sizes, scheduling quanta,

and the effects of context switches and process migrations. Different platforms provide different cache sizes and architectures, and cache sizes are likely to change over time. Transactions that exceed resource limits or are repeatedly interrupted will never commit. Ideally, programmers should be shielded from such complex, platform-specific details. Instead, TM systems should provide full support even for transactions that cannot execute directly in hardware.

Techniques that substantially increase the size of hardware transactions include signatures [17] and permissions-only caches [18]. Other proposals support (effectively) unbounded transactions by allowing transactional metadata to overflow caches, and for transactions to migrate from one core to another. These proposals include TCC [19], VTM [20], OneTM [18], UTM [21], TxLinux [11], and LogTM [17].

## Software Transactional Memory

*Software transactional memory* (STM) is an alternative to direct hardware support for TM. STM is a software system that provides programmers with a transactional model through a library or compiler interface. In this section, we describe some of the questions that arise when designing an STM system. Some of these questions concern *semantics*, that is, how the STM behaves, and other concern *implementation*, that is, how the STM is structured internally.

### Weak vs Strong Isolation

How should threads that execute transactions interact with threads executing non-transactional code? One possibility is *strong isolation* [22] (sometimes called *strong atomicity*), which guarantees that transactions are atomic with respect to non-transactional accesses. The alternative, *weak isolation* (or *weak atomicity*), makes no such guarantees. HTM systems naturally provide strong atomicity. For STM systems, however, strong isolation may be too expensive.

The distinction between strong and weak isolation leaves unanswered a number of other questions about STM behavior. For example, what does it mean for an unhandled exception to exit an atomic block? What does I/O mean if executed inside a transaction? One appealing approach is to say that transactions behave as if they were protected by a *single global lock* (SGL) [19, 23, 24].

One limitation of the SGL semantics is that it does not specify the behavior of *zombie* transactions: transactions that are doomed to abort because of synchronization conflicts, but continue to run for some duration before the conflict is discovered. In some STM implementations, zombie transactions may see an inconsistent state before aborting. When a zombie aborts, its effects are rolled back, but while it runs, observed inconsistencies could provoke it to pathological behavior that may be difficult for the STM system to protect against, such as dereferencing a null pointer or entering an infinite loop. *Opacity* [25] is a correctness condition that guarantees that all uncommitted transactions, including zombies, see consistent states.

## I/O and System Calls

What does it mean for a transaction to make a system call (such as I/O) that may affect the outside world? Recall that transactions are often executed speculatively, and a transaction that encounters a synchronization conflict may be rolled back and restarted. If a transaction creates a file, opens a window, or has some other external side effect, then it may be difficult or impossible to roll everything back.

One approach is to allow *irrevocable* transactions [11, 18, 26] that are not executed speculatively, and so never need to be undone. An irrevocable transaction cannot explicitly abort itself, and only one such transaction can run at a time, because of the danger that multiple irrevocable transactions could deadlock.

An alternative approach is to provide a mechanism to escape from the transactional system. Escape actions [27] and open nested transactions citeNi07 allow a thread to execute statements outside the transaction system, scheduling application-specific commit and abort handlers to be called if the enclosing transaction commits or aborts. For example, an escape action might create a file, and register a handler to abort that file if the transaction aborts. Escape mechanisms can be misused, and often their semantics are not clearly defined. Using open nested transactions, for example, care must be taken to ensure that abort handlers do not deadlock.

## Exploiting Object Semantics

STM systems typically synchronize on the basis of *read/write conflicts*. As a transaction executes, it records

the data items it read in a *read set*, and the data items it wrote in a *write set*. Two transactions *conflict* if one transaction's read or write set intersects the other's write set. Conflicting transactions cannot both commit. Synchronizing via read/write conflicts has one substantial advantage: it can be done automatically without programmer participation. It also has a substantial disadvantage: It can severely and unnecessarily restrict concurrency for certain shared objects. If these objects are subject to high levels of contention (that is, they are “hot-spots”), then the performance of the system as a whole may suffer.

This problem can be addressed by open nested transactions, as described above in Section 5.2, but open nested transactions are difficult to use correctly, and lack the expressive power to deal with certain common cases [28].

Another approach is to use type-specific synchronization and recovery to exploit concurrency inherent in an object's high-level specification. One such mechanism is *transactional boosting* [28], which allows thread-safe (but non-transactional) object implementations to be transformed into highly concurrent transactional implementations by allowing method calls to proceed in parallel as long as their high-level specifications are *commutative*.

## Eager vs Lazy Update

There are two basic ways to organize transactional data. In an *eager* update system, data objects are modified in place, and each transaction maintains an *undo log* allowing it to undo its changes if it aborts. The dual approach is *lazy* (or deferred) update, where each transaction computes optimistically on its local copy of the data, installing the changes if it commits, and discarding them if it aborts. An eager system makes committing a transaction more efficient, but makes it harder to ensure that zombie transactions see consistent states.

## Eager vs Lazy Conflict Detection

STM systems differ according to when they detect conflicts. In *eager* conflict detection schemes, conflicts are detected before they arise. When one transaction is about to create a conflict with another, it may consult a contention manager, defined below, to decide whether to pause, giving the other transaction a chance to finish, or to proceed and cause the other to abort. By contrast,

a *lazy* conflict detection scheme detects conflicts when a transaction tries to commit. Eager detection may abort transactions that could have committed lazily, but lazy detection discards more computation, because transactions are aborted later.

## Contention Managers

In many STM proposals, conflict resolution is the responsibility of a *contention manager* [29] module. Two transactions *conflict* if they access the same object and one access is a write. If one transaction discovers it is about to conflict with another, then it can pause, giving the other a chance to finish, or it can proceed, forcing the other to abort. Faced with this decision, the transaction consults a contention management module that encapsulates the STM’s conflict resolution policy.

The literature includes a number of contention manager proposals [29–32], ranging from exponential backoff to priority-based schemes. Empirical studies have shown that the choice of a contention manager algorithm can affect transaction throughput, sometimes substantially.

## Visible vs Invisible Reads

Early STM systems [29] used either *invisible reads*, in which each transaction maintains per-read metadata to be revalidated after each subsequent read, or *visible reads*, in which each reader registers its operations in shared memory, allowing a conflicting writer to identify when it is about to create a conflict. Invisible read schemes are expensive because of the need for repeated validation, while visible read schemes were complex, expensive, and not scalable.

More recent STM systems such as TL2 [33] or SKYSTM [34] use a compromise solution, called *semi-visible reads*, in which read operations are tracked imprecisely. Semi-visible reads conservatively indicate to the writer that a read-write conflict might exist, avoiding expensive validation in the vast majority of cases.

## Privatization

It is sometimes useful for a thread to *privatize* [35] a shared data structure by making it inaccessible to other threads. Once the data structure has been privatized, the owning thread can work on the data structure directly, without incurring synchronization costs. In principle,

privatization works correctly under SGL semantics, in which every transaction executes as if it were holding a “single global lock.” Unfortunately, care is required to ensure that privatization works correctly. Here are two possible hazards. First, the thread that privatizes the data structure must observe all changes made to that data by previously committed transactions, which is not necessarily guaranteed in an STM system where updates are lazy. Second, a doomed (“zombie”) transaction must not be allowed to perform updates to the data structure after it has been privatized.

## Bibliographic Notes and Further Reading

The most comprehensive TM survey is the book *Transactional Memory* by Larus and Rajwar [23]. Of course, this area changes rapidly, and the best way to keep up with current developments is to consult the the *Transactional Memory Online* web page at: <http://www.cs.wisc.edu/trans-memory/>.

## Bibliography

1. Hoare CAR (1974) Monitors: an operating system structuring concept. Commun ACM 17(10):549–557
2. Herlihy MP, Wing JM (1990) Linearizability: a correctness condition for concurrent objects. ACM T Progr Lang Sys 12(3):463–492
3. Michael MM, Scott ML (1996) Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC, Philadelphia. ACM, New York, pp 267–275
4. Harris T, Marlow S, Peyton-Jones S, Herlihy M (2005) Composable memory transactions. In: PPoPP ’05: Proceedings of the tenth ACM SIGPLAN symposium on principles and practice of parallel programming, Chicago. ACM, New York, pp 48–60
5. Herlihy M, Moss JEB (May 1993) Transactional memory: architectural support for lock-free data structures. In: International symposium on computer architecture, San Diego
6. Hickey R (2008) The clojure programming language. In: DLS ’08: Proceedings of the 2008 symposium on dynamic languages, Paphos. ACM, New York, pp 1–1
7. Microsoft Corporation. Stm.net. <http://msdn.microsoft.com/en-us/devlabs/ee334183.aspx>
8. Korland G Deuce STM. <http://www.deucestm.org/>
9. S. Microsystems. DSTM2. <http://www.sun.com/download/products.xml?id=453fb28e>
10. Intel Corporation. C++ STM compiler. <http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition-20/>
11. Rossbach CJ, Hofmann OS, Porter DE, Ramadan HE, Aditya B, Witchel E (2007) TxLinux: using and managing hardware transactional memory in an operating system. In: SOSP ’07: Proceedings of twenty-first ACM SIGOPS symposium on operating systems principles, Stevenson. ACM, New York, pp 87–102

12. Damron P, Fedorova A, Lev Y, Luchangco V, Moir M, Nussbaum D (2006) Hybrid transactional memory. In: ASPLOS XII: Proceedings of the 12th international conference on architectural support for programming languages and operating systems, Boston. ACM, New York, pp 336–346
13. Pankratius V, Adl-Tabatabai A-R, Otto F (Sept 2009) Does transactional memory keep its promises? Results from an empirical study. Technical Report 2009-12, University of Karlsruhe
14. Rossbach CJ, Hofmann OS, Witchel E (Jun 2009) Is transactional memory programming actually easier? In: Proceedings of the 8th annual workshop on duplicating, deconstructing, and debunking (WDDD), Austin
15. Rajwar R, Goodman JR (2001) Speculative lock elision: enabling highly concurrent multithreaded execution. In: MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on microarchitecture, Austin. IEEE Computer Society, Washington, DC, pp 294–305
16. Click C (Feb 2009) Experiences with hardware transactional memory. <http://blogs.azulsystems.com/cliff/2009/02/and-now-some-hardware-transactional-memory-comments.html>
17. Yen L, Bobba J, Marty MR, Moore KE, Volos H, Hill MD, Swift MM, Wood DA (2007) LogTM-SE: decoupling hardware transactional memory from caches. In: HPCA '07: Proceedings of the 2007 IEEE 13th international symposium on high performance computer architecture, Phoenix. IEEE Computer Society, Washington, DC, pp 261–272
18. Blundell C, Devietti J, Lewis EC, Martin M (Jun 2007) Making the fast case common and the uncommon case simple in unbounded transactional memory. In: International symposium on computer architecture, San Diego
19. Hammond L, Carlstrom BD, Wong V, Hertzberg B, Chen M, Kozyrakis C, Olukotun K (2004) Programming with transactional coherence and consistency (TCC). ACM SIGOPS Oper Syst Rev 38(5):1–13
20. Rajwar R, Herlihy M, Lai K (Jun 2005) Virtualizing transactional memory. In: International symposium on computer architecture, Madison
21. Ananian CS, Asanović K, Kuszmaul BC, Leiserson CE, Lie S (Feb 2005) Unbounded transactional memory. In: Proceedings of the 11th international symposium on high-performance computer architecture (HPCA'05), San Francisco, pp 316–327
22. Blundell C, Lewis EC, Martin MMK (Jun 2005) Deconstructing transactions: the subtleties of atomicity. In: Fourth annual workshop on duplicating, deconstructing, and debunking, Wisconsin
23. Larus J, Rajwar R (2007) Transactional memory (Synthesis lectures on computer architecture). Morgan & Claypool, San Rafael
24. Menon V, Balensiefer S, Shpeisman T, Adl-Tabatabai A-R, Hudson RL, Saha B, Welc A (2008) Single global lock semantics in a weakly atomic STM. SIGPLAN Notices 43(5):15–26
25. Guerraoui R, Kapalka M (2008) On the correctness of transactional memory. In: PPoPP '08: Proceedings of the 13th ACM SIGPLAN symposium on principles and practice of parallel programming, Salt Lake City. ACM, New York, pp 175–184
26. Welc A, Saha B, Adl-Tabatabai A-R (2008) Irrevocable transactions and their applications. In: SPAA '08: Proceedings of the twentieth annual symposium on parallelism in algorithms and architectures, Munich. ACM, New York, pp 285–296
27. Moravan MJ, Bobba J, Moore KE, Yen L, Hill MD, Liblit B, Swift MM, Wood DA (2006) Supporting nested transactional memory in logTM. SIGPLAN Notices 41(11):359–370
28. Herlihy M, Koskinen E (2008) Transactional boosting: a methodology for highly-concurrent transactional objects. In: PPoPP '08: Proceedings of the 13th ACM SIGPLAN symposium on principles and practice of parallel programming, Salt Lake City. ACM, New York, pp 207–216
29. Herlihy M, Luchangco V, Moir M, Scherer W (Jul 2003) Software transactional memory for dynamic-sized data structures. In: Symposium on principles of distributed computing, Boston
30. Guerraoui R, Herlihy M, Pochon B (2005) Toward a theory of transactional contention managers. In: PODC '05: Proceedings of the twenty-fourth annual ACM symposium on principles of distributed computing, Las Vegas. ACM, New York, pp 258–264
31. Scherer WN III, Scott ML (Jul 2004) Contention management in dynamic software transactional memory. In: PODC workshop on concurrency and synchronization in java programs, St. John's
32. Attiya H, Epstein L, Shachnai H, Tamir T (2006) Transactional contention management as a non-clairvoyant scheduling problem. In: PODC '06: Proceedings of the twenty-fifth annual ACM symposium on principles of distributed computing, Denver. ACM, New York, pp 308–315
33. Dice D, Shalev O, Shavit N (2006) Transactional locking II. In: Proceedings of the 20th international symposium on distributed computing, Stockholm
34. Lev Y, Luchangco V, Marathe V, Moir M, Nussbaum D, Olszewski M (2009) Anatomy of a scalable software transactional memory. In: TRANSACT 2009, Raleigh
35. Spear MF, Marathe VJ, Dalessandro L, Scott ML (2007) Privatization techniques for software transactional memory. In: PODC '07: Proceedings of the twenty-sixth annual ACM symposium on principles of distributed computing, Portland. ACM, New York, pp 338–339

## Transactions, Nested

J. ELIOT B. MOSS

University of Massachusetts, Amherst, MA, USA

## Synonyms

[Multi-level transactions](#); [Nested spheres of control](#)

## Definition

*Nested Transactions* extend the traditional semantics of transactions by allowing meaningful nesting of one or

more child transactions within a parent transaction. Considering the traditional ACID properties of transactions, atomicity, consistency, isolation, and durability, a child transaction possesses these properties in a relative way, such that a parent transaction effectively provides a universe within which its children act similarly to ordinary transactions in a non-nested system. In parallel computation, the traditional property of durability in the face of various kinds of system failures may not be required.

## Discussion

### Transactions

Transactions are a way of guaranteeing atomicity of more or less arbitrary sequences of code in a parallel computation. Unlike locks, which identify computations that may need serialization according to the identity of held and requested locks, transaction serialization is based on the specific data accessed by a transaction while it runs. In some cases, it is possible usefully to pre-declare the maximal set of data that a transaction might access, but it is not possible in general. Transactions *specify semantics*, while locks express an implementation of serialization. The usual semantics of transactions are that concurrent execution of a collection of transactions must be equivalent to execution of those transactions one at a time, in some order. This property is called *serializability*.

The ACID properties capture transaction semantics in a slightly different way. *Atomicity* requires that transactions are all-or-nothing: Either all of a transaction's effects occur, or the transaction fails and has no effect. *Consistency* requires that each transaction take the state of the world from one consistent state to another. *Isolation* requires that no transaction perceive any state in the middle of execution of another transaction. *Durability* requires that the effects of any transaction, once the transaction is accepted by the system, not disappear.

In general, in parallel computing, as opposed to database systems where transactions had their origins, the consistency and durability properties are often less important. Consistency may be ignored in that most commonly there are no explicitly stated consistency

constraints and no mechanism to enforce them. However, a correct transaction system is still required not to leave effects of partially executed or failed transactions. Durability is often ignored in that a parallel computing system may have no permanent state. If the system has distributed memory, then it may achieve significant durability by keeping multiple copies of data in different units of the distributed memory. Of course, parallel systems can also use one or more nonvolatile copies to achieve durability, according to a system's durability requirements.

It is important to distinguish transactions from concurrency-safe data structures. A concurrency-safe data structure generally offers a guarantee of *linearizability*: If two actions on the data structure by different threads overlap in their execution, then the effect is as if the actions are executed in one order or the other. That is, a set of concurrent actions by different threads appears to occur in some linear order. This is *similar* to serializability, but what transactions and serializability add is the possibility for a given thread to execute a whole *sequence* of actions  $a_1, a_2, \dots, a_n$  without any intervening actions of other threads. A concurrency-safe data structure guarantees only that the individual  $a_i$  execute correctly as defined by the data type, but permits actions of other threads to interleave between the  $a_i$ .

In discussing transactions and their nesting, some additional terms will be useful. Transactions are said to *commit* (succeed) or *abort* (fail). They may fail for many reasons, one of them being serialization conflicts with other transactions. Transactional concurrency control may be *pessimistic*, also called early conflict detection, or *optimistic*, also called late conflict detection. Pessimistic conflict detection usually employs some kind of locks, while optimistic generally uses some kind of version numbers or timestamps on data and transactions to determine conflicts. It is even possible to maintain multiple versions so as to allow more transactions to commit, while still enforcing serializability. In general, locking schemes require some kind of deadlock avoidance or detection protocol. However, if a set of transactions is in deadlock, the system has a way out: It can abort one of the transactions to break the deadlock. Thus, deadlock is not a fatal problem as it is when using just locks for synchronization.

A system may update data *in place*, which requires an *undo log* to support removing the effects of a failed transaction. Alternatively, a system may create *new copies* of data, and install them only if a transaction commits, which in general requires a *redo log*. Updating *in-place* requires early conflict detection if the system guarantees that a transaction will not see effects of other uncommitted transactions. It is possible to allow such effects to be visible, but serializability then requires that the observing transaction commit only if the transaction it observed also commits. However, the system must still prevent two transactions from observing each other's effects, since then they cannot be serialized. More advanced models have also been explored but are not discussed here.

## Semantics of Nesting

The simplest motivation for nesting is to make it easy to compose software components into larger systems. If a library routine uses transactions, and the programmer wishes to use that routine within an application transaction, then there will be nesting of transaction begin/end pairs. One can simply treat this as one large transaction, effectively ignoring the inner transaction begin/end pairs. However, it is also possible to attribute transactional semantics to them, as follows.

Consider a transaction T and a transaction U contained with it, i.e., whose begin and end are between the begin and end of T. T is a *parent transaction* and U a *child transaction* or *subtransaction* of T. If T is not contained within any enclosing transaction, it is *top-level*. Only proper nesting of begin/end transaction pairs is legal, so a top-level transaction and subtransactions at all depths form a tree.

Conflict semantics of non-nested transactions extend to nested transactions straightforwardly in terms of relationships in the forest of transaction trees. If action A conflicts with action B when executed by two different non-nested transactions T1 and T2, then A conflicts with B in the nested setting if neither of T1 and T2 is an ancestor of the other. Why is there no conflict in the case where, say, T1 is an ancestor of T2? It is because T1 is providing an environment or universe *within which* additional transactions can run, compete, and be serialized.

It is simplest, however, to envision nesting where a parent does not execute actions directly, but rather always creates a child transaction to perform them. Alternatively, a parent might perform actions directly, but only when it has no active subtransactions. This model leads to serializability among the subtransactions of any given transaction T. However, as viewed from outside of the transaction tree that includes T and its descendants, T and its subtransactions form a single transaction that must itself be serializable with transactions outside of T.

As with non-nested transactions, a nested transaction can fail (abort). In that case, it is as if the failing transaction, and all of its descendants, never ran. Thus, commit of a child transaction is not final, but only relative to its parent, while abort of the parent is final and aborts all descendants, even those that have (provisionally) committed.

## Example Closed Nesting Implementation Approach

Consider adding support for nesting to a non-nested transaction implementation that employs *in-place* update and early conflict detection based on locking. As each transaction runs, it accumulates a set of locks and a list of undos. If the transaction aborts, the system applies the transaction's undos (in reverse order) and then discards the transaction's locks. Note that a transaction T can be granted a lock L provided that the only conflicting holders of L are ancestors of T. Also note that discarding a child's lock does not discard any ancestor's lock on the same item. If a child transaction commits, then the system adds the child's locks to those held by the parent, and appends the child's undo list to that of the parent. If a top-level transaction commits, the system simply discards its held locks and its undo list. Moss [5, 6] described this protocol. It is also possible to devise timestamp-based approaches, possibly supporting late conflict detection, as articulated by Reed [10].

Notice that these nesting schemes use the same set of possible actions at each level of nesting. They provide *temporal grouping* of actions, and in a distributed

system can also be used for *spatial grouping* within temporal groups.

### Motivations for Closed Nesting

There are two primary advantages of closed nesting. One is that failure of a child does not require immediate failure of its parent. Thus, if a transaction desires to execute an action that has higher than usual likelihood of causing failure, it can execute that action within a child transaction and avoid immediate failure of itself should the action cause an abort. In a centralized system aborts might most likely be caused by conflicts with other transactions, but in a decentralized system, failure of a remote node or communication link is also possible. Thus, remote calls are natural candidates to execute as subtransactions. If a child does fail, the parent can retry it, which may often make sense, or the parent can perform some alternate action. For example, in a distributed system, if one node of a replicated database is down, the parent could try another one.

A possibly stronger motivation for closed nesting is safe transaction execution when the application desires to exploit concurrency *within* a transaction. It is easy to see this by considering that if there is concurrency within a transaction, then proper semantics and synchronization or serialization within the transaction present the same issues that led to proposing transaction mechanisms in the first place. Even if, at first blush, it appears that the space of data that concurrent actions might update is disjoint, and thus that there can be no conflict, that property can be a delicate one, and difficult to enforce in complex software systems having many layers. For example, transaction T at node A might make apparently disjoint concurrent remote calls to nodes B and C. However, B and C, unknown to T, use a common service at node D, and should have their actions properly serialized there. If the work at B and C is not performed in distinct concurrent child transactions of T, the work at D might not be properly serialized.

### Open Nesting

While closed nested transactions indeed support safe concurrency within transactions, and also offer limited

recoverability from partial failure, they have a significant limitation: Transactions that are “big,” either in terms of how long they run or the volume of data they access, tend to conflict with other transactions. While this cannot always be avoided, many conflicts are *false conflicts* at the level of application semantics. For example, consider a transaction T that adds a number of new records to a data structure organized as a B-tree. Logically speaking, if other transactions do not access these records or otherwise inquire directly or indirectly about their presence or absence in the data structure, then they do not conflict with T. However, straightforward mechanisms for guaranteeing safe transactional access to the B-tree might acquire locks on B-tree nodes and hold them until T commits. Thus, other transactions could be locked out of whole nodes of the tree, even though they are not (logically) affected by the changes T is making.

The solution discovered in the context of databases applies also to the case of parallel computing. It is to make a distinction between different *levels of semantics*, and requires recognizing certain data as being part of a coherent and distinct data abstraction. For example, in the case of a B-tree, each B-tree node is part of a given B-tree, and should be visible and manipulated only by actions on that B-tree. That is, the B-tree nodes are *encapsulated* within their owning B-tree. This allows B-tree actions to apply conflict management and undo or redo to B-tree nodes, during execution of those actions, and for the system to switch to *abstract* concurrency control and *abstract* undo or redo once a B-tree action is complete.

How does this solve the problem? The concurrency control and undo/redo on B-tree nodes allows safe concurrent (transactional) execution of B-tree actions themselves. This could also be achieved by non-transactional locking, lock-free or wait-free algorithms, or any other means that guarantees linearizability, but open nesting is generally taken to refer to the recursive use of transaction-like mechanisms, while wrapping a not necessarily transactional data type with abstract concurrency control and recovery is called *transactional boosting* [4]. More significantly, though, the conflicts between full B-tree *actions* will be much fewer than the (internal, temporary) conflicts on B-tree nodes during

those actions. For example, looking up record  $r_1$  and adding record  $r_2$  do not conflict logically, but if they lie in the same B-tree node, there will be a (physical) conflict on that node. While open nesting is by no means restricted to use with such collection data types, it is certainly very useful in allowing higher concurrency for them.

### An Example Open Nesting Protocol

A fleshed-out example protocol for open nesting may be helpful to build understanding. This example uses in-place update and employs locks for early conflict detection, but other protocols are possible for other approaches to update and conflict detection. For understanding the protocol, a specific example data structure is instructive. Consider a `Set` abstraction implemented using a linked list. Suppose it supports actions `add(x)` and `remove(x)` to manipulate whether  $x$  is in the set, `size()` to return the set's cardinality, and `contains(x)` to test whether  $x$  is in the set.

Suppose the items  $a$ ,  $b$ , and  $c$  have been added to the set in that order, and that `add` appends new elements to the end (since it must scan to the end anyway in order to avoid entering duplicates). Assume these elements are committed and there are no transactions pending against the set. Now suppose that a transaction adds a new member  $d$  and continues with other work. During the `add` operation, the transaction observes  $a$ ,  $b$ , and  $c$  and the list links, then it creates a new list node containing  $d$  and modifies  $c$ 's link to refer to the new node. Suppose that another transaction concurrently queries whether the set contains  $e$ . At the physical level this `contains` query conflicts with the uncommitted `add`, because the query will read the link value in  $c$ 's list node, etc. Likewise a transaction that tries to `remove b` will conflict with both the `add` and the `contains` actions because it will try to modify the link in  $a$ 's node, to unchain  $b$ 's node from the list. To guarantee correct manipulation of the list each operation can acquire transactional locks on list nodes, acquiring an exclusive (X) mode lock when modifying a node and a share (S) mode lock when only observing it. Two S mode locks on the same object do not conflict, but all other mode combinations conflict. The pointer to the first node is likewise protected with the same kind of

locks. This locking protocol works fine for closed nesting, but it is easy to see that it leads to many needless conflicts.

Open nesting requires identifying the *abstract* conflicts between operations. This example protocol uses *abstract locks*. These locks include an S and X mode lock for each possible element of the set, and an additional lock with modes Read (R) and Modify (M) for the cardinality of the set. Two R mode locks do not conflict, and two M mode locks also do not conflict, but R and M mode conflict with each other. Here is a table showing the abstract locks acquired by each action on a `Set`:

|                          |                                       |
|--------------------------|---------------------------------------|
| <code>add(x)</code>      | X mode on $x$ ; M mode on cardinality |
| <code>remove(x)</code>   | X mode on $x$ ; M mode on cardinality |
| <code>contains(x)</code> | S mode on $x$                         |
| <code>size()</code>      | R mode on cardinality                 |

This can be refined to acquire only an S mode lock on  $x$  if `add` or `remove` does not actually change the membership of the set, and in that case also not to acquire the M mode lock on the cardinality.

Assuming that each action on the set is run as an open nested transaction, then before an action completes it must acquire the specified abstract locks. If it cannot do that, it is in conflict and some transaction must be aborted. Once the action is complete *and* holds the abstract locks, the nested transaction commits and releases the lower-level locks on list nodes. The parent transaction will hold the abstract locks until it itself commits. Similar to closed nesting, if an uncommitted open nested transaction aborts, it can simply unwind back to where it started and try again. Often such cases arise because of temporary conflicts on the physical data structure. However, if the conflict is because of an abstract lock, then retry is not likely to help – either other transactions need to complete or abort to get out of this transaction's way, or this transaction needs to abort higher up in the nested transaction tree.

Abstract locks are just one way of implementing detection of abstract conflicts. In general what is required is an encoding of *abstract conflict predicates* into conflict checking code. These conflict predicates

indicate which actions on a data type conflict with other actions. Here is an example table for Set:

|             | add(y) | remove(y) | contains(y) | size() |
|-------------|--------|-----------|-------------|--------|
| add(x)      | x = y  | x = y     | x = y       | true   |
| remove(x)   | x = y  | x = y     | x = y       | true   |
| contains(x) | x = y  | x = y     | false       | false  |
| size()      | true   | true      | false       | false  |

In this table, the left action is considered to have been performed by one transaction, and the right action is requested by another. The entry in the table indicates the condition under which the new request conflicts with the older, not yet committed, action. The table above is expressed in terms of the operations and their arguments. However, it is possible to refine these predicates if they can refer to the *state* of the set. In general this might include the state after the first operation as well as the state before it. The refined table below uses references to the state S before the first operation:

|             | add(y)                    | remove(y)              | contains(y)               | size()       |
|-------------|---------------------------|------------------------|---------------------------|--------------|
| add(x)      | $x = y \wedge x \notin S$ | x = y                  | $x = y \wedge x \notin S$ | $x \notin S$ |
| remove(x)   | x = y                     | $x = y \wedge x \in S$ | $x = y \wedge x \in S$    | $x \in S$    |
| contains(x) | $x = y \wedge x \notin S$ | $x = y \wedge x \in S$ | false                     | false        |
| size()      | $y \notin S$              | $y \in S$              | false                     | false        |

Open nesting involves more than just conflict detection. Until an open nested action commits, aborting it works like aborting a closed nested action: Simply apply its (lower level) undos in reverse order and release its (lower level) locks. However, once an open nested action commits, it does not work to undo it using the list of lower level undos it accumulated while it ran. The lower level undos are guaranteed to work properly only if the lower level locks are still held. To undo a *committed* open nested action, the system applies an *abstract undo*, also called an *inverse* or *compensating action*. Here is a table of inverses for actions on the Set abstraction; a — entry means that no inverse is needed (the action did not change the state):

| Action  | add(x)                            | remove(x)                   | contains(x) | size() |
|---------|-----------------------------------|-----------------------------|-------------|--------|
| Inverse | if $x \notin S$ then<br>remove(x) | if $x \in S$ then<br>add(x) | —           | —      |

Notice that the appropriate inverse can depend on the state in which the original action ran. If add or remove does not actually change the state, then they can simply omit adding an inverse.

These inverses are added to the parent transaction's undo list, to apply if the parent transaction needs to abort. The abstract concurrency control will guarantee that these inverses still make sense when they are applied. They should be run as open nested transactions, and if they fail, it will only be because of temporary conflicts on the physical data structure, so they should simply be retried until they succeed.

### Coarse-Grained Transactions

Transactions at the more abstract level, employing abstract concurrency control, and, if using in-place update, abstract undos, can be more generally termed *coarse-grained transactions*. As previously noted, the individual actions need not be run as transactions under a transaction mechanism – all that is required is that they are linearizable. However, using a transaction mechanism does offer the advantage of being able to abort and retry an action in case of conflict, while other approaches must guarantee absence of conflict. Thus, if the system does not use nested transactions to implement the actions, then, if using in-place update, it will need to acquire abstract locks *before* running the action. This implies that it cannot base the lock acquisition on the state of the data abstraction or on the result of the action. However, if executing the action reveals that the originally acquired abstract lock is stronger than necessary, the implementation can then downgrade the lock.

As noted before, the underlying implementation might use non-transactional locks to synchronize (for example, one mutual exclusion lock on the whole data structure will work, at the cost of reduced concurrency), or might use lock-free, obstruction-free, or wait-free techniques to obtain linearizability.

Upon first consideration, in-place updates may appear more complicated, since they require specifying, implementing, tracking, and applying undos. However, providing new copies of an abstract data structure has its own problems. One difficulty is being clear as to what needs to be copied. A second problem is cost. To reduce cost, a coarse-grained transaction implementation might use *Bloom filters*, which record

and examine an ongoing transaction's changes in a side data structure, private to the transaction. Transactions must also record additional information even for read-only actions, in order to check for conflicts later.

### Correct Abstract Concurrency Control

The tables above gave conflict predicates without indicating how to derive or verify them. A conflict predicate is *safe* if the actions commute when the predicate is false. [Some make a distinction between actions moving to the left and to the right, but in most practical cases actions either commute (move both ways) or they do not (neither way).] Two actions commute if executing them in either order allows them to return the same results, and also does not affect the outcome of any future actions. Assuming that all relevant aspects of the state are observable, then this can be rephrased as: Actions commute if, when executed in either order, they produce the same results and the same final state.

There are some subtleties lurking in this definition. First, the “same state” means the same *abstract* state. For example, in the case of *Set* implemented as a linked list, the abstract state consists in what elements are in, and not in, the set. The order in which the members occur on the linked list does not matter. Therefore, even though `add(x)` and `add(y)` result in a different linked list when executed in the opposite order, *abstractly* there is no difference. Thus it is important to have clarity about what the abstract state *is*.

Second, interfaces vary in what they reveal. For example, `add(x)` might return nothing, not revealing whether `x` was previously in the set. As far as concurrency control goes, the less revealing interface reduces conflicts: two transactions could both do `add(x)` without conflict as perceived via this interface.

Third, if the system uses undos, then the undo added to a transaction's undo list is part of the result to consider when determining conflicts. So, if `x` is not initially in a set, and then two transactions each invoke `add(x)`, even if the `add` actions return no result, the actions conflict since the undo for the first one is `remove(x)` and the undo for the second is “do nothing.” In this respect, late conflict detection sometimes allows more concurrency. However, in general it

requires making a copy (at least an effective copy) of the data structure, and if any transaction commits changes to the data structure while transaction T is running, T's actions must be redone on the primary copy of the data structure rather than directly installing the new state that T constructed.

Fourth, certain non-mutating operations entail concurrency control obligations that may at first seem surprising. For example, if `x` is not in a set and transaction T runs the query `contains(x)`, the set must guarantee that any other transaction that adds `x` will conflict with T. Thus, if the system uses abstract locking, `contains(x)` must in this case lock the *absence* of `x`. Hence, an abstract lock is not necessarily a lock attached to some piece of the original data structure. (In some database implementations the locks are mixed with the actual records, and the system creates a new record for a lock like this, a record that goes away at the end of the transaction. This is called a *phantom record*.) A similar case occurs with an ordered set abstraction when a call to `getNextHigher(x)` returns `y`: The transaction must lock the fact that the ordered set has no value between `x` and `y`. Thus, read-only actions still require checking and recording, and this applies equally to late conflict detection as to early detection.

### Extended Semantics

Another use for open nesting is to break out of strict serializability (at the programmer's risk, of course). It is sometimes useful, even necessary, to keep some effects of a transaction even if it is aborted. For example, in processing a commercial transaction, a system might discover that the credit card presented is on a list of stolen cards. While most effects of the purchase should be undone, information about the attempted use of the card should go to a log that will definitely *not* be undone. This is easy to do by giving the log action's inverse as “do nothing.” (This is harder to do in a system that is not doing in-place updates, and would require a special notation.) In this way open nesting can be abused to achieve irrevocable effects. Similarly, a programmer can underestimate conflict predicates and allow communication between transactions. The “extended semantics” of open nesting abused in these ways may depend on the underlying implementation.

## Nesting in Transactional Memory

Both closed and open nesting have been proposed for use with transactional memory (TM), for both software (STM) and hardware (HTM) approaches. The primary difficulty in implementing closed nesting for TM is its more complex conflict rule. It no longer suffices to check for equality or inequality of transaction identifiers – the test must distinguish an ancestor transaction from a non-ancestor. (This assumes that only transactions that are currently leaves of the transaction tree can execute.) HTM designs must also deal with the reality that hardware resources are always limited, and thus, there may be hard limits on the nesting depth, for example. HTM will also not be aware of abstract locks and abstract concurrency control; they will always be implemented in software. However, the number of conflict checks required for abstract concurrency control is strictly less than for physical units such as words or cache lines.

It is particularly more complex to check for conflict between concurrent subtransactions running under nested TM. However, if a transaction has at most one child at once, and only leaf transactions can execute, then the implementation is only slightly more complex than for non-nested transactions. Because the transaction tree in this case consists of a single line of descent from a top-level transaction, it is called *linear nesting*. Linear nesting admittedly forgoes one of the strong advantages of nesting, namely concurrent sibling subtransactions, but it retains partial rollback and thus remains potentially more useful than non-nested transactions.

## Bibliographic Notes and Further Reading

The early exposition of nested transactions is marked by Davies [2], Reed [10], and Moss [5, 6]. Open nesting (also called *multi-level transactions*) was articulated by Beeri et al. [1], Moss et al. [8], and Weikum and Schek [11]. Nested transactions for hardware transactional memory are explored in Yen et al. [12] and Moss and Hosking [7], and Ni et al. [9] describe a prototype that supports open nesting in software transactional memory. Transactional boosting was introduced by Herlihy and Koskinen [4]. Gray and Reuter [3] provide

comprehensive coverage of transaction processing concepts and techniques.

## Bibliography

1. Beeri C, Bernstein PA, Goodman N (1983) A concurrency control theory for nested transactions. In: Proceedings of the ACM Symposium on Principles of Distributed Computing. ACM Press, New York, pp 45–62
2. Davies CT Jr (1973) Recovery semantics for a DB/DC system. In: ACM '73 Proceedings of the ACM Annual Conference. ACM, New York, pp 136–141. doi: <http://doi.acm.org/10.1145/800192.805694>
3. Gray J, Reuter A (1993) Transaction processing: concepts and techniques. Data Management Systems. Morgan Kaufmann, Los Altos, CA
4. Herlihy M, Koskinen E (2008) Transactional boosting: a methodology for highly-concurrent transactional objects. In: Chatterjee S, Scott ML (eds) Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, 20–23 Feb 2008. ACM, New York, pp 207–216, ISBN 978-1-59593-795-7
5. Moss JEB (1985) Nested transactions: an approach to reliable distributed computing. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA (Also published as MIT Laboratory for Computer Science Technical Report 260)
6. Moss JEB (1985) Nested transactions: an approach to reliable distributed computing. MIT Press, Cambridge, MA
7. Moss JEB, Hosking AL (2006) Nested transactional memory: model and architecture sketches. Sci Comput Progr 63: 186–201
8. Moss JEB, Griffeth ND, Graham MH (1986) Abstraction in recovery management. In: Proceedings of the ACM Conference on Management of Data. Washington, DC. ACM SIGMOD, ACM Press, New York, pp 72–83
9. Ni Y, Menon V, Adl-Tabatabai AR, Hosking AL, Hudson RL, Moss JEB, Saha B, and Shpeisman T (2007) Open nesting in software transactional memory. In: ACM SIGPLAN 2007 Symposium on Principles and Practice of Parallel Programming, San Jose, CA. ACM, New York, pp 68–78
10. Reed DP (1978) Naming and synchronization in a decentralized computer system. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA (Also published as MIT Laboratory for Computer Science Technical Report 205)
11. Weikum G, Schek HJ (1992) Concepts and applications of multilevel transactions and open nested transactions. Morgan Kaufmann, Los Altos, CA, pp 515–553, ISBN 1-55860-214-3
12. Yen L, Bobba J, Marty MR, Moore KE, Volos H, Hill MD, Swift MM, Wood DA (2007) LogTM-SE: decoupling hardware transactional memory from caches. In: Proceedings of the 13th International Conference on High-Performance Computer Architecture (HPCA-13 2007), 10–14 February 2007, Phoenix, Arizona, USA, IEEE Computer Society, Washington, DC, pp 261–272

**Transpose**

► [All-to-All](#)

**Tuning and Analysis Utilities**

► [TAU](#)

**TStreams**

► [Concurrent Collections Programming Model](#)

# U

## Ultracomputer, NYU

ALLAN GOTTLIEB  
New York University, New York, NY, USA

### Definition

The NYU Ultracomputer is a class of highly parallel, shared-memory multiprocessors featuring two innovations: the *fetch-and-add* process coordination primitive, and a hardware method to *combine* (nearly simultaneous) *memory references* directed at the same location. These references can be loads, stores, or fetch-and-adds. The combining procedure generalizes to fetch-and-phi for any associative operator phi, for example, max or min.

Combining often enables all simultaneous memory requests from many processors to be completed in essentially the time required for just one such request. This property, together with the power of fetch-and-add/phi, has enabled construction of coordination algorithms with the important property of imposing no serialization beyond that specified in the problem itself. For example, fetch-and-add-based readers-writers algorithms, during periods of no writer activity, can satisfy multiple requests in essentially constant time. Similarly, fetch-and-add-based queue algorithms, during periods when the queue is neither full nor empty, can satisfy multiple inserts and deletes in essentially constant time. Such implementations are sometimes called “bottleneck free.”

The Ultracomputer project constructed a few small-scale prototype implementations, the largest being a fully functional, 16-processor prototype with a processor-to-memory interconnection network whose full-custom VLSI switches successfully combined memory references, including fetch-and-adds.

Implemented software included two generations of the Unix operating system tailored to run on the multiprocessor. The first version required nearly all kernel

activity to occur on one processor; the second was itself a parallel program that made extensive use of locally developed, highly parallel (aka scalable) process coordination algorithms. All the software was compiled by the locally developed port of GCC (the GNU Compiler Collection, formerly the GNU C Compiler).

### Discussion

#### Early History

The NYU Ultracomputer project was launched in 1979 when Jack Schwartz wrote a highly influential paper [14] itself entitled *Ultracomputers*. However, the primary architecture described in that paper is not what became the NYU Ultracomputer. Instead, influenced by the Burroughs proposal for NASA’s Numerical Aerodynamic Simulation Facility, Schwartz and his NYU colleagues increased their consideration of shared-memory MIMD (Multiple Instruction stream Multiple Data stream) designs. This lead to the discovery of both fetch-and-add together with several effective process coordination algorithms using this primitive, and the hardware design enabling nearly simultaneous fetch-and-adds to be combined into a single memory operation.

In fact fetch-and-add had been discovered previously. Dijkstra [4] considered essentially the same primitive and showed that the natural fetch-and-add implementation of a semaphore had a fatal race condition. The NYU group, ignorant of this earlier work, not only found the same primitive but also implemented essentially the same flawed semaphore, thinking it was correct. However, they subsequently discovered the same race condition, and found another semaphore implementation, which they proved correct.

#### Fetch-and-Add (and Friends)

*Fetch-and-add* is essentially an indivisible add to memory operation. Its format is F&A ( $X, v$ ), where  $X$  is an integer variable and  $v$  is an integer value. This function

with side effects is defined to both return the (old) value of  $X$  and replace the memory value of  $X$  by  $X+v$ . That is,  $\text{F\&A}(X, v)$  fetches  $X$  and adds  $v$  to it.

Simultaneous fetch-and-add's directed at the same integer variable  $X$ , are required to satisfy the *serialization principle* (cf. Eswaran et al. [6]), meaning that their effect must be the same as if they were executed serially in some unspecified order. That is, fetch-and-add is *atomic*.

### Fetch-and- $\phi$ and Replace-Add

The fetch-and-add function can easily be generalized to fetch-and- $\phi$ , where  $\phi$  is a binary operator. For example `fetch-and-max(X, v)` would return the old value of  $X$  while atomically setting  $X := \max(X, v)$ . As shall be seen below, the ability to combine simultaneous fetch-and- $\phi$ 's requires that  $\phi$  be associative.

The early versions of fetch-and-add actually used a slightly different function `Replace-add(X, v)` which returned the new (incremented) version of  $X$ . Thus, it was an atomic version of `++X` rather than `X++`. For an invertible function such as addition, there is little to choose between the two variants since

$\text{F\&A}(X, v)$  is equivalent to

$\text{Replace-Add}(X, v) - v$

$\text{Replace-Add}(X, v)$  is equivalent to

$\text{F\&A}(X, v) + v$

However, when generalizing to non-invertible  $\phi$ 's such as `max`, the fetch-and- $\phi$  version is superior to the replace- $\phi$  version since there is no analogue of the first equivalence for `max`. This observation becomes especially important when simultaneous `fetch-and- $\phi$ (X, v)`'s are performed on the same variable  $X$ .

### A Fetch-and-Add Semaphore

Perhaps the simplest coordination algorithm is the busy-waiting, binary semaphore (aka. spin-lock) used to enforce mutual exclusion for critical sections.

```
loop
 P(S)
 critical section
 V(S)
end loop
```

The idea behind a fetch-and-add-based semaphore  $S$  is to treat  $S$  as an integer variable, letting  $S = 1$  represent

"open" and  $S \leq 0$  represent "closed."  $S$  is then atomically incremented and decremented with `fetch-and-add`.

A successful  $P(S)$  operation finds  $S = 1$ . It executes  $\text{F\&A}(S, -1)$  receiving 1 and setting  $S$  to 0. The  $V(S)$  operation executes  $\text{F\&A}(S, +1)$  thereby undoing the effect of its successful  $P$ .

But what about an unsuccessful  $P(S)$ , one that finds  $S < 1$  when it does  $\text{F\&A}(S, -1)$ ? The unsuccessful  $P(S)$  simply does  $\text{F\&A}(S, +1)$  to undo its decrement and tries again, hoping that some  $V(S)$  will have occurred in the interim.

Code implementing this (faulty) pair of algorithms is presented in Section Code Examples; the corresponding functions are named `NaiveP(S)` and `V(S)`. These two functions are essentially the same code that Dijkstra correctly criticized for having a fatal race condition in which no process is in the critical section, but several process continually try and fail to enter. For example, consider the following scenario, beginning with the semaphore initially open (i.e.,  $S = 1$ ) and three processes A, B, and C arriving at `NaiveP(S)`.

Assume all three execute the  $\text{F\&A}(S, -1)$  simultaneously and the serial order effected is A, B, C. Thus A receives +1 and enters the critical section. B and C receive 0 and -1, respectively; they each add back a 1 and retry. As long as A is inside the critical section (i.e., has not executed  $V(S)$ ),  $S$  remains less than 1 correctly preventing B and C from succeeding.

When A does execute  $V(S)$ , its initial decrement is undone and either B or C should be able to succeed. Indeed, if both B and C have just executed  $\text{F\&A}(S, +1)$ , the semaphore will be restored to its initial value of +1 and the next process to execute  $\text{F\&A}(S, -1)$  will receive +1 and succeed. However, this may never happen since the following endless scenario may occur instead.

Assume that, after A increments  $S$ , both B and C are at their increment instruction (at this point  $S = -1$ ). Suppose that the effected serialization is that B increments and then decrements prior to C executing. The result is that  $S$  moves to 0 and then back to -1, and B receives 0 from  $\text{F\&A}(S, -1)$ ; thus B was not successful. Now the same sequence can occur for C, and then again for B, etc.

Although this race condition is quite unlikely to occur for three processes, it is *possible*, and becomes steadily more probable as the number of processes increases.

If  $\text{Naive } P(S)$  is replaced by  $\text{True } P(S)$  also from the section Code Examples, the race is no longer possible. Indeed, using  $\text{True } P(S)$ , a correct implementation of  $P(S)$ , Gottlieb et al. [11] proved that if, at time  $T$ ,  $N > 0$  processes are executing the  $P(S) - V(S)$  loop above and no process ceases execution, then there is a time  $t > T$  when exactly one process is executing its critical section.

At first glance,  $\text{True } P(S)$  appears to simply add a redundant test to  $\text{Naive } P(S)$ . As just shown, however, the extra test is needed. In fact the resulting test-decrement-retest paradigm has proved fruitful for other algorithms as well (Gottlieb et al. [11]). Tighter coding of several of these algorithms can be found in [7] and application to a highly parallel version of Unix can be found in [5].

## The Ultracomputer Architecture

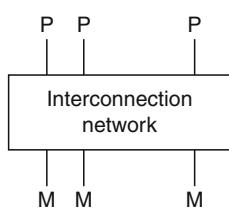
The basic Ultracomputer design is that of a symmetric multiprocessor, that is, there are multiple processors sharing a central memory. Although the memory can be packaged together with the processors, the prototype hardware did not do this and neither do the diagrams in this article. Figure 1 in the Section Diagrams shows the basic block diagram with multiple processors connected to multiple memory modules.

## The Processors

The Ultracomputer processors are fairly standard; the primary novelty is that they can issue fetch-and-add instructions. The prototype hardware used AMD 29000 microprocessors with locally developed support hardware to implement fetch-and-add.

## The Memory Modules

The memory modules were also straightforward. An adder was included at each module to enable fetch-and-add.



Ultracomputer, NYU. Fig. 1 Ultracomputer block diagram

## The Interconnection Network

The primary novelty of the hardware design is the ability of the interconnection network to combine (nearly) simultaneous requests to the same memory location, in particular, the ability to combine multiple fetch-and-add operations into a single memory operation.

**The Topology** The interconnection topology used was an  $N \times N$   $\Omega$ -Network composed of  $2 \times 2$  crossbar switches. See Fig. 2 for an example with  $N = 8$  and, for the moment ignore the colors. The very left and right columns contain 0–7 written in binary, they correspond to the eight processors and memory modules; the rectangles are bidirectional crossbar switches that can connect either left port with either right port; and the inter-column wires implement the so-called shuffle map.

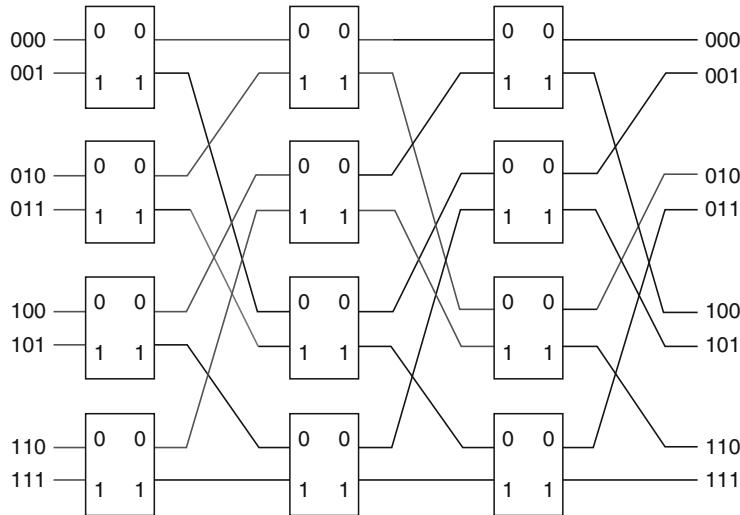
Consider the red wire. Counting from zero it connects the third horizontal line on the left to the fifth on the right. Note that  $3 = 011$ ,  $5 = 101$ , and  $\text{RightRotate}(011) = 101$ . That is the defining property of the shuffle map; it holds for all the wires.

To see that any of the eight far left ports can be connected to any of eight far right, first note that any crossbar can connect  $xyz$  to either  $xy0$  or  $xy1$ , that is, each crossbar can determine the low-order bit of the right-hand row independent of the low-order bit of the left-hand row. So any column can “correct” the low-order bit and the shuffle map ensures that each successive column operates on a new low-order bit. Thus, after  $\log N$  columns and shuffles, the message is at the desired row.

When traversed “backward” (i.e., from right to left), the network can again correct the low-order bit in each column, but now gets the next low-order bit via the inverse shuffle, which corresponds to a left rotate. Thus, each of the processors can access each memory module and vice versa.

Now consider the blue lines in Fig. 2. They show the paths from each processor to memory module number 2. Note that  $2 = 010$  and that every path successively exits switch ports 0, 1, and 0. Thus the  $\Omega$ -network is the superposition of  $N$  trees, one rooted at each memory module. It is also the superposition of  $N$  other trees, one rooted at each processor.

It will be important for combining to note that the network has a *unique* path  $\pi$  from a given processor to a



Ultracomputer, NYU. Fig. 2 8 × 8 Omega network

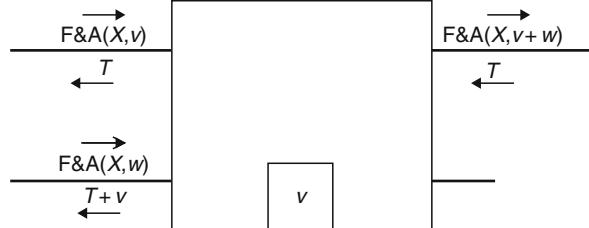
given memory module and that the (unique) path from that memory to that processor is just the reverse of  $\pi$ . It is possible to use a network with multiple paths but then forward request must generate enough information to ensure that the reply traverses the reverse path.

**The Switches** As mentioned, each Ultracomputer switch is a  $2 \times 2$  crossbar. The switches are buffered so that if two requests entering the left side wish to exit the same right-side port, one is stored at the switch (assuming the available buffer space has not been exhausted). In addition to providing the throughput advantages of buffered switches, buffering is essential for the chosen combining implementation.

### Combining Loads and Stores

It is fairly easy to see how to combine two loads or two stores that are directed at the same memory location and are present in a single switch at the same time. For two loads, the switch sends one forward and retains the other. When the memory response arrives, two replies are sent back, one for each request.

For two stores, the switch again sends just one forward. When this store is performed at memory, the effect is as if the non-forwarded store was executed immediately before. When the memory acknowledgment for the forwarded store arrives, the switch acknowledges both requests. It is important for program



Ultracomputer, NYU. Fig. 3 Combining two fetch-and-adds

semantics not to acknowledge either request before the memory acknowledgment has arrived.

Note that combined loads and combined stores appear to the next stage switch just like ordinary loads and stores and thus can be further combined. In the best case, each processor can issue a request to the same memory location and all  $N$  requests will be combined along the tree to the memory module so that only a single request need be serviced at the memory itself.

### Combining Fetch-and-Adds

Combining fetch-and-adds is not as easy as combining loads or stores [13]. Figure 3 in section Diagrams shows the actions performed when  $F\&A(X, v)$  meets  $F\&A(X, w)$  at a switch. The switch computes the sum  $v + w$ , transmits the combined request  $F\&A(X, v+w)$ , and stores the value  $v$  in its local memory. Eventually, a value  $T$  will be returned to the switch in response to

$\text{F\&A}(X, v+w)$ . At this point, the switch returns  $T$  to satisfy the request  $\text{F\&A}(X, v)$ , returns  $T + v$  to satisfy the request  $\text{F\&A}(X, w)$ , and deletes  $v$  from its local memory.

If the combined request  $\text{F\&A}(X, v+w)$  was not itself combined, then  $T$  is simply  $X$ . In this case, the values returned are  $X$  and  $X+v$  which is consistent with the serialization order “ $\text{F\&A}(X, v)$  followed immediately by  $\text{F\&A}(X, w)$ .” The memory location has been incremented to  $X + (v + w)$ . By the associativity of addition, this value equals  $(X + v) + w$ , which is again consistent with “ $\text{F\&A}(X, v)$  followed immediately by  $\text{F\&A}(X, w)$ .”

It is not hard to apply this argument recursively and see that, thanks to associativity, if combined requests are further combined while on their way to memory, the results returned to each requesting processor are consistent with some serialization of the constituent fetch-and-adds. For example, Fig. 4 shows four fetch-and-adds with increments 1, 2, 4, and 8 being combined and de-combined to achieve the serialization order  $\text{F\&A}(X, 8)$ ,  $\text{F\&A}(X, 4)$ ,  $\text{F\&A}(X, 1)$ ,  $\text{F\&A}(X, 2)$ .

### Combining Heterogeneous Requests

Having seen how to combine loads, stores, and fetch-and-adds, it is natural to ask if the switch can combine two *different* request types. Since `load X` can be

viewed as  $\text{F\&A}(X, 0)$ , the only heterogeneous case that need be considered is combining a fetch-and-add with a store, which can be combined as follows.

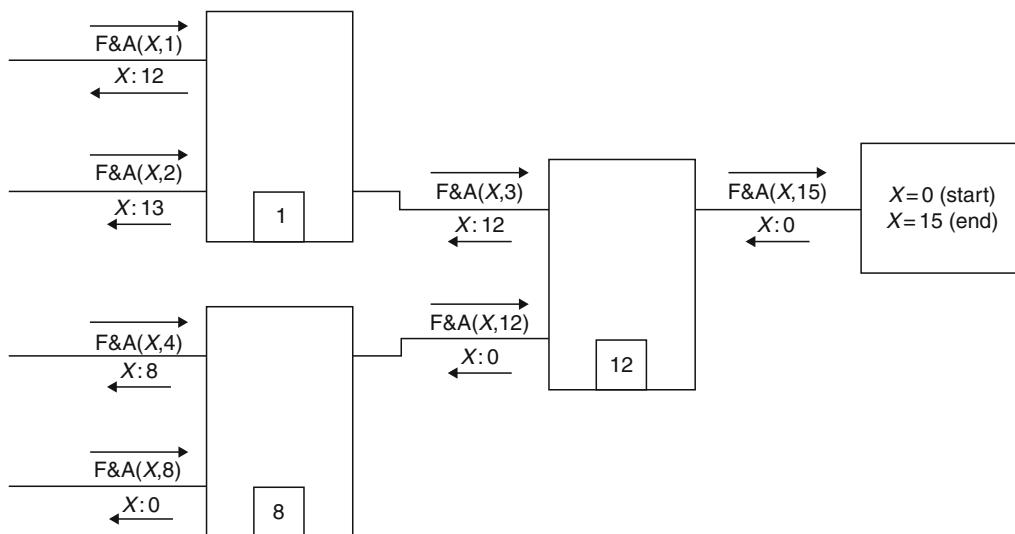
When  $\text{F\&A}(X, v)$  meets  $\text{Store}(X, w)$ , the switch transmits  $\text{Store}(w+v)$  and remembers  $w$ . When the acknowledgment of this “combined” store is received, the switch acknowledges the original store, returns  $w$  to satisfy  $\text{F\&A}(X, v)$ , and deletes  $w$  from its local memory. These actions have the same effect as “ $\text{Store}(X, w)$  followed immediately by  $\text{F\&A}(X, v)$ .”

### Combining Switch Design

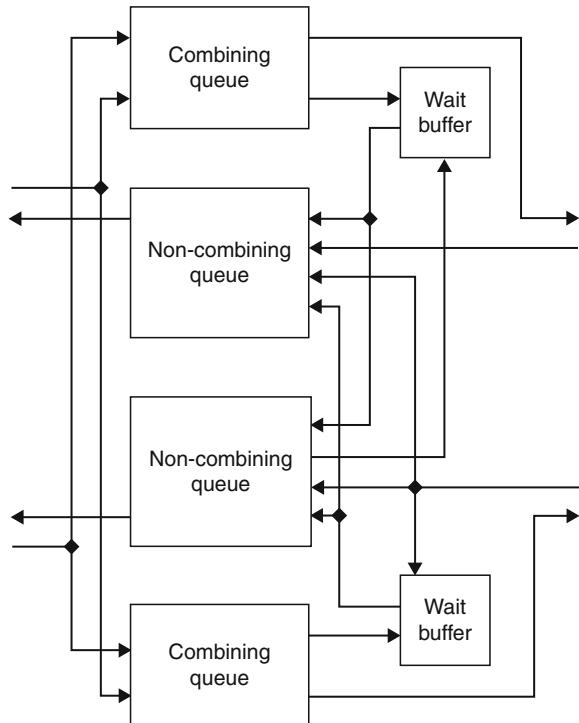
As shown above, the physical combining of memory requests occurs at the network switches. The first externally available description of combining switches was [15], a later description can be found in [2], and a detailed description together with analysis and simulation results occurs in Dickey’s thesis [3].

A block diagram of a single  $2 \times 2$  switch is shown in Fig. 5. Unlike previous diagrams in which lines represented bidirectional data flow, each line in this figure is unidirectional. The processors would be to the left and memory to the right.

In terms of the combining description given in section Combining Fetch-and-Adds,  $\text{F\&A}(X, v)$  meets  $\text{F\&A}(X, w)$  in the Combining Queue associated with the right-side port on the path to  $X$ . The value  $v$



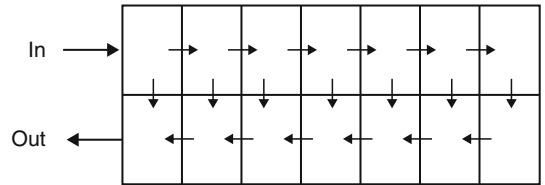
Ultracomputer, NYU. Fig. 4 Combining four fetch-and-adds

Ultracomputer, NYU. Fig. 5  $2 \times 2$  Switch

is stored in the Wait Buffer, together with additional information identifying the requests. The eventual response from memory is compared with entries in the Wait Buffer and matching entries are sent to the Non-combining Queue(s) associated with left-side port(s) on the paths to the processors that issued the original requests.

There are challenges designing each of the components and in partitioning the design to meet the constraints imposed by the low pin-count chip packages available at the time. However, the most interesting component is likely the enhanced systolic queue used to implement the combining queue, so that component is described in more detail.

**(Semi-)Systolic (Combining) Queues** Before turning our attention to the actual queue designs, note that Fig. 5 shows combining queues with two inputs. Designs of this kind were studied, but the actual implementation of the two-input queues consisted of two one-input queues followed by a two-input multiplexer (see Ref. [3] for a detailed comparison).

Ultracomputer, NYU. Fig. 6  $2 \times 2$  Systolic queue

**Systolic Queues** The point of departure for the NYU combining queue, was the systolic (non-combining) queue design of Guibas and Liang [12], the basic idea of which is shown in Fig. 6. This queue implementation is called systolic since communication within the queue is local, unlike a solution using a RAM with head and tail pointers. Avoiding global signals is advantageous for VLSI.

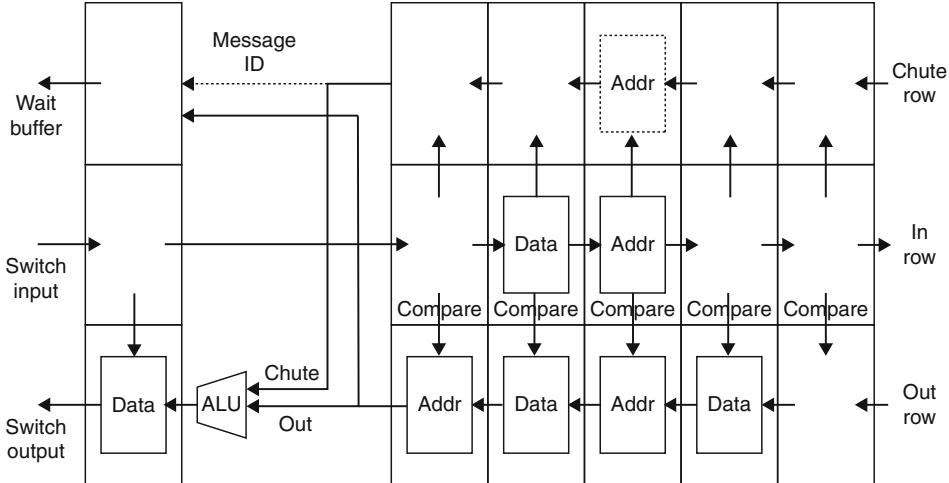
Items enter the *In* row and exit the *Out* row. If the output destination can always accept an item, then only the first column is used. In a single cycle, an item enters *In*, goes to the first column, and then drops to the bottom row, from which it exits on the subsequent cycle.

More interesting is the behavior when the downstream queue is full and blocks the current queue. In this case the item will find that it cannot go down (since that slot is full and the current occupant cannot leave) and it remains in the top row. In the next cycle, the item in question again moves right and then moves down, providing the correspond slot is available. Thus, if downstream remains full, the current queue will itself become full as all columns become occupied.

As the careful reader will have noticed, the single cycle referred to above is (internal to the queue) divided into two subcycles, one each for horizontal and vertical motion.

The NYU designs for both the non-combining queues just described and the combining queues to be described next used two global control signals *in\_moving* and *out\_moving* and thus are properly called quasi- or semi-systolic. The global signals can be precomputed at the expense of an extra column to accept an incoming message from each predecessor after the switch announces that it is full. The signals can be implemented efficiently as qualified clocks (see Ref. [3]).

The semi-systolic design used in the implementation permits insertions and deletions to occur every



Ultracomputer, NYU. Fig. 7 2 × 2 Combining queue

cycle, an important property that is not supported by the purely systolic design of [12].

**Semi-Systolic Combining Queues** The key observation enabling a variation of the systolic queues to support combining is that as an item moves left along *Out*, it passes directly under all subsequent items as they move right along *In*. When one item is directly above another, local logic can compare them and determine if they are destined for the same address. (Since both *In* and *Out* can move during the same cycle, it would appear that items can pass by without having a chance for comparison. However, all Ultracomputer memory references use an even number of packets, the first of which contains the address. Thus, with some extra logic, the system can assure that “address packets” are vertically adjacent at the end of a cycle.)

Figure 7, adapted from [3], is a block diagram of the Ultracomputer combining queue. When this diagram is compared to Fig. 6, the non-combining queue, three additions become apparent: the new *Chute* row, the comparators between the *In* and *Out* rows, and the “stuff on the left.” Each is discussed in turn.

When combinable messages are detected (one traversing *Out*, the other *In*), the message traversing *In* is moved to the *Chute*. As a result it will never be moved to *Out* and will not be sent to the memory. For example, Fig. 7 shows this event occurring in column 3. The *Chute* moves in synchrony with *Out* and thus both

the message traversing *Out* and the corresponding message traversing *Chute* will leave their respective rows together.

If the needed slot in *Chute* is already full when combinable messages are detected, the second combining opportunity is lost. (The *Out* message has already been combined with another message from *In*.) Thus, having only one *Chute* supports only pairwise combining in a single switch. As noted above, these pairwise combined messages can themselves combine as they pass through subsequent switches. Adding a second *Chute* (together with enhancing the ALU to accept three inputs, and other changes) would permit three-way combining within a switch.

The comparators between the *In* and *Out* rows detect references to the same address. As mentioned previously, while the address packet of a message traverses *Out*, it will sit directly below all address packets traversing *In*. When the comparator in a column detects equal addresses in the *In* and *Out* rows, logic in the column moves the address packet (and subsequently the remaining packets) to the *Chute* (unless this chute slot is full due to a previous combine, as described earlier in this section.).

The logic on the left is essentially an extra column, *without* the comparators, together with an ALU needed to sum the increments when two fetch-and-adds are combined. The lack of a comparator in this leftmost column does mean that some combining opportunities are lost. It was omitted to prevent the increased cycle

time that would result from one column implementing both addition and comparison.

When a combined message moves from the rectangular array of cells to the left-side logic, the memory address plus some identifying information is stored in the wait buffer. If the operation was fetch-and-add, the *Out*-row increment is also stored. This enables the eventual memory response to be decombined into responses for the two originating requests.

## Code Examples

```
procedure NaiveP(S) -- faulty
 implementation of P(S)
 OK := False;
 repeat
 if F&A(S,-1) > 0
 OK := True;
 else
 F&A(S,+1);
 end if;
 until OK;
end NaiveP;

procedure V(S) -- correct
 implementation
 F&A(S,+1);
end V;

procedure TrueP(S) -- correct
 implementation
 OK := False;
 repeat
 if S > 0 then
 if F&A(S,-1) > 0 then
 OK := True;
 else
 F&A(S,+1);
 end if;
 end if;
 until OK;
end P;
```

## Related Entries

- [Buses and Crossbars](#)
- [Cedar Multiprocessor](#)
- [Flynn's Taxonomy](#)

- [Interconnection Networks](#)
- [Memory Models](#)
- [MIMD \(Multiple Instruction, Multiple Data\) Machines](#)
- [Networks, Multistage](#)
- [Non-Blocking Algorithms](#)
- [Nonuniform Memory Access \(NUMA\) Machines](#)
- [Parallel Computing](#)
- [PRAM \(Parallel Random Access Machines\)](#)
- [Reduce and Scan](#)
- [Scan for Distributed Memory, Message-Passing Systems](#)
- [Shared-Memory Multiprocessors](#)
- [Switch Architecture](#)
- [Switching Techniques](#)
- [Synchronization](#)

## Bibliographic Notes and Further Reading

Perhaps the most polished account of the NYU Ultracomputer can be found in the book by Almasi and Gottlieb [1]. More information, plus an extensive bibliography, can be found in [8]. An early Ultracomputer paper [10] was selected to appear in a collection of ISCA papers together with a retrospective on the project [9].

## Bibliography

1. Almasi G, Gottlieb A (1994) Highly parallel computing, 2nd edn. Benjamin/Cummings, Redwood City
2. Dickey S, Kenner R, Snir M (1986) An implementation of a combining network for the NYU ultracomputer. Ultracomputer Note 93, NYU
3. Dickey SR (1994) Systolic combining switch designs. PhD thesis, NYU, May 1994
4. Dijkstra EW (1972) Hierarchical orderings of sequential processes. In: Hoare CAR, Perrot RH (eds) Operating systems techniques. Academic Press, New York
5. Edler J (1995) Practical structures for parallel operating systems. PhD thesis, NYU, May 1995
6. Eswaran KP, Gray JN, Lorie RA, Traiger IL (Nov 1976) The notion of consistency and predicate locks in a database system. Commun ACM 19(11):624–633
7. Freudenthal E (2003) Comparing and improving centralized and distributed techniques for coordinating massively parallel shared-memory systems. PhD thesis, NYU, May 2003
8. Gottlieb A (1987) An overview of the NYU ultracomputer project. In: Dongarra JJ (ed) Experimental parallel computing architectures. North Holland, Amsterdam, pp 25–95
9. Gottlieb A (1998) A personal retrospective on the NYU ultracomputer. In: Sohi GS (ed) 25 years of the international symposia

- on computer architecture (selected papers), Barcelona, ACM, New York, pp 29–31
10. Gottlieb A, Grishman R, Kruskal CP, McAuliffe KP, Rudolph L, Snir M (1982) The NYU ultracomputer|designing a MIMD shared memory parallel machine. In: International symposium on computer architecture, Austin, Apr 1982, pp 27–42
  11. Gottlieb A, Lubachevsky BD, Rudolph L (Apr 1983) Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *Trans Program Lang Syst* 5(2):164–189
  12. Guibas LJ, Liang F (1982) Systolic stacks, queues, and counters. In: MIT conference on research in VLSI, Cambridge, pp 155–162
  13. Rudolph LS (1982) Software structures for ultraparallel computing. PhD thesis, NYU, Feb 1982
  14. Schwartz JT (Oct 1980) Ultracomputers. *ACM Toplas* 2(4): 484–521
  15. Snir M, Solworth J (1984) The ultraswitch—a VLSI network node for parallel processing. *Ultracomputer Note* 39, NYU, Jan 1984

## Discussion

### Introduction

This essay gives the basic introduction to Unimodular Transformations. More advanced results on this topic are given in the essay ► *Loop Nest Parallelization* in this encyclopedia, where other transformations are considered as well.

The program model is a perfect nest  $\mathbf{L}$  of  $m$  sequential loops. An *iteration* of  $\mathbf{L}$  is an instance of the body of  $\mathbf{L}$ . The program consists of a certain set of iterations that are to be executed in a certain sequential order. This execution order imposes a *dependence structure* on the set of iterations, based on how they access different memory locations. A loop in  $\mathbf{L}$  carries a dependence if the dependence between two iterations is due to the unfolding of that loop. A loop can run in parallel if it carries no dependence.

A *unimodular matrix* is a square integer matrix with determinant 1 or -1. An  $m \times m$  unimodular matrix  $\mathbf{U}$  can be used to transform  $\mathbf{L}$  into a perfect nest  $\mathbf{L}'$  of  $m$  loops, with the same set of iterations executing in a different sequential order. This is a *unimodular transformation*. The new nest  $\mathbf{L}'$  is *equivalent* to the old nest  $\mathbf{L}$  (i.e., they give the same results), if whenever an iteration depends on another iteration in  $\mathbf{L}$ , the first iteration is executed after the second in  $\mathbf{L}'$ . If a loop in  $\mathbf{L}'$  carries no dependence, then that loop can run in parallel. It turns out that a unimodular transformation always exists, such that  $\mathbf{L}'$  is equivalent to  $\mathbf{L}$  and all loops in  $\mathbf{L}'$ , except possibly one, can run in parallel.

There is a section on mathematical preliminaries to prepare the reader for what comes next. It is followed by a section on basic concepts. The general unimodular transformation is then explained in detail by examples. Finally, some important special features of special unimodular transformations like loop permutation and loop skewing are pointed out.

## Uncertainty Quantification

### ► Terrestrial Ecosystem Carbon Modeling

## Underdetermined Systems

### ► Linear Least Squares and Orthogonal Factorization

## Unified Parallel C

### ► UPC

## Unimodular Transformations

UTPAL BANERJEE

University of California at Irvine, Irvine, CA, USA

### Definition

A *Unimodular Transformation* is a general loop transformation based on a unimodular matrix. It changes a perfect nest of sequential loops into a similar nest with the same set of iterations executing in a different sequential order.

### Mathematical Preliminaries

For basic Linear Algebra concepts used in this essay, see any standard text. The set of real numbers is denoted by  $\mathbf{R}$  and the set of integers by  $\mathbf{Z}$ . For any positive integer  $m$ ,  $\mathbf{R}^m$  denotes the  $m$ -dimensional vector space over  $\mathbf{R}$  consisting of all real  $m$ -vectors, where vector addition and scalar multiplication are defined coordinate-wise in the usual way. The zero vector  $(0, 0, \dots, 0)$  is written as  $\mathbf{0}$ .

The subset of  $\mathbf{R}^m$  consisting of all integer  $m$ -vectors is denoted by  $\mathbf{Z}^m$ .

### Lexicographic Order

For  $1 \leq \ell \leq m$ , define a relation  $\prec_\ell$  in  $\mathbf{Z}^m$  as follows: If  $\mathbf{i} = (i_1, i_2, \dots, i_m)$  and  $\mathbf{j} = (j_1, j_2, \dots, j_m)$  are vectors in  $\mathbf{Z}^m$ , then  $\mathbf{i} \prec_\ell \mathbf{j}$  if

$$i_1 = j_1, i_2 = j_2, \dots, i_{\ell-1} = j_{\ell-1}, \text{ and } i_\ell < j_\ell.$$

The *lexicographic order*  $\prec$  in  $\mathbf{Z}^m$  is then defined by requiring that  $\mathbf{i} \prec \mathbf{j}$ , if  $\mathbf{i} \prec_\ell \mathbf{j}$  for some  $\ell$  in  $1 \leq \ell \leq m$ . The notation  $\mathbf{i} \leq \mathbf{j}$  means either  $\mathbf{i} \prec \mathbf{j}$  or  $\mathbf{i} = \mathbf{j}$ . Note that  $\leq$  is a total order in  $\mathbf{Z}^m$ .

The associated relations  $>$  and  $\geq$  are defined in the usual way:  $\mathbf{j} > \mathbf{i}$  means  $\mathbf{i} \prec \mathbf{j}$ , and  $\mathbf{j} \geq \mathbf{i}$  means  $\mathbf{i} \leq \mathbf{j}$ . ( $\geq$  is also a total order in  $\mathbf{Z}^m$ .) An integer vector  $\mathbf{i}$  is *positive* if  $\mathbf{i} > \mathbf{0}$ , *nonnegative* if  $\mathbf{i} \geq \mathbf{0}$ , and *negative* if  $\mathbf{i} < \mathbf{0}$ .

For  $x \in \mathbf{R}$ , the value of  $\text{sgn}(x)$  is 1, -1, or 0, according as  $x > 0$ ,  $x < 0$ , or  $x = 0$ , respectively. The *direction vector* of a vector  $(i_1, i_2, \dots, i_m) \in \mathbf{Z}^m$  is the vector of signs:  $(\text{sgn}(i_1), \text{sgn}(i_2), \dots, \text{sgn}(i_m))$ . Note that a vector is positive (negative) if and only if its direction vector is positive (negative).

For more details on lexicographic order, see [4].

### Fourier's Method of Elimination

Fourier's method of elimination can be used to decide if a system of linear inequalities in real variables has a solution [12]. Consider a set of  $n$  inequalities in  $m$  real variables  $x_1, x_2, \dots, x_m$ , of the form

$$a_{1j}x_1 + a_{2j}x_2 + \dots + a_{mj}x_m \leq c_j \quad (1 \leq j \leq n), \quad (1)$$

where the  $a_{ij}$ 's and the  $c_j$ 's are real constants. To solve this system, eliminate the variables one at a time in the order:  $x_m, x_{m-1}, \dots, x_1$ . If an inequality  $c \leq 0$  turns up during the elimination process where  $c$  is a positive constant, then the system has no solution. Otherwise, the final result is a system of inequalities of the form:

$$b_i(x_1, x_2, \dots, x_{i-1}) \leq x_i \leq B_i(x_1, x_2, \dots, x_{i-1}) \quad (1 \leq i \leq m), \quad (2)$$

where each  $b_i$  is either  $-\infty$  or the maximum of a set of linear functions of  $x_1, x_2, \dots, x_{i-1}$ , and each  $B_i$  is either  $\infty$  or the minimum of a set of linear functions of  $x_1, x_2, \dots, x_{i-1}$ . In this case,  $b_1 \leq B_1$ . The set of all real solutions  $(x_1, x_2, \dots, x_m)$  to (1) is given by (2): Pick any

$x_1$  such that  $b_1 \leq x_1 \leq B_1$ , then pick any  $x_2$  such that  $b_2(x_1) \leq x_2 \leq B_2(x_1)$ , and so on.

For the purpose of finding loop limits after a unimodular transformation, Fourier's method is extended to check for any integer solutions to a system of the form (1), where the coefficients  $a_{ij}$ 's and  $c_j$ 's are integer constants. If (1) has no real solution, then obviously it has no integer solution. Assume then (1) has a set of real solutions  $(x_1, x_2, \dots, x_m)$  given by (2). The set of all integer solutions  $(z_1, z_2, \dots, z_m)$  to (1) is then given by

$$\alpha_i(z_1, z_2, \dots, z_{i-1}) \leq z_i \leq \beta_i(z_1, z_2, \dots, z_{i-1}) \quad (1 \leq i \leq m), \quad (3)$$

where  $\alpha_i$  and  $\beta_i$  are defined by  $\alpha_i(z_1, z_2, \dots, z_{i-1}) = [b_i(z_1, z_2, \dots, z_{i-1})]$  and  $\beta_i(z_1, z_2, \dots, z_{i-1}) = [B_i(z_1, z_2, \dots, z_{i-1})]$  for each  $i$ . If  $\alpha_1 > \beta_1$ , then there is no integer solution. Otherwise, proceed by enumeration: Take any integer  $z_1$  such that  $\alpha_1 \leq z_1 \leq \beta_1$ , then check if there is an integer  $z_2$  such that  $\alpha_2(z_1) \leq z_2 \leq \beta_2(z_1)$ , and so on.

The process described above is illustrated by the following example. For a detailed description of Fourier's method including an algorithm, see [4].

*Example 1* Consider the system of linear inequalities

$$\left. \begin{array}{rcl} 2x_1 - 11x_2 & \leq & 3 \\ -3x_1 + 2x_2 & \leq & -5 \\ x_1 + 3x_2 & \leq & 4 \\ -2x_1 & & \leq -3. \end{array} \right\} \quad (4)$$

Rearrange the inequalities so that the ones with a positive coefficient for  $x_2$  are at the top; they are followed by the ones where the coefficient for  $x_2$  is negative; and then come the inequalities where  $x_2$  is absent:

$$\left. \begin{array}{rcl} -3x_1 + 2x_2 & \leq & -5 \\ x_1 + 3x_2 & \leq & 4 \\ 2x_1 - 11x_2 & \leq & 3 \\ -2x_1 & & \leq -3. \end{array} \right\} \quad (5)$$

Divide each of the first three inequalities by the corresponding coefficient of  $x_2$ :

$$\left. \begin{array}{l} -\frac{3}{2}x_1 + x_2 \leq -\frac{5}{2} \\ \frac{1}{3}x_1 + x_2 \leq \frac{4}{3} \\ -\frac{2}{11}x_1 + x_2 \geq -\frac{3}{11} \end{array} \right\}$$

Note how the direction of the third inequality is reversed after division, as the coefficient of  $x_2$  in it was negative. Isolate  $x_2$  in this system to get

$$\left. \begin{array}{l} x_2 \leq \frac{3}{2}x_1 - \frac{5}{2} \\ x_2 \leq -\frac{1}{3}x_1 + \frac{4}{3} \\ \frac{2}{11}x_1 - \frac{3}{11} \leq x_2 \end{array} \right\}$$

Thus, when  $x_1$  is given,  $x_2$  must satisfy

$$b_2(x_1) \leq x_2 \leq B_2(x_1), \quad (6)$$

where

$$\begin{aligned} b_2(x_1) &= \frac{2}{11}x_1 - \frac{3}{11} \\ B_2(x_1) &= \min\left(\frac{3}{2}x_1 - \frac{5}{2}, -\frac{1}{3}x_1 + \frac{4}{3}\right). \end{aligned}$$

Now, eliminate  $x_2$  by setting each of its lower bounds less than or equal to each of its upper bounds:

$$\left. \begin{array}{l} \frac{2}{11}x_1 - \frac{3}{11} \leq \frac{3}{2}x_1 - \frac{5}{2} \\ \frac{2}{11}x_1 - \frac{3}{11} \leq -\frac{1}{3}x_1 + \frac{4}{3} \end{array} \right\}$$

Simplify these inequalities and bring in the last member of (5):

$$\left. \begin{array}{l} -\frac{29}{22}x_1 \leq -\frac{49}{22} \\ \frac{17}{33}x_1 \leq \frac{53}{33} \\ -2x_1 \leq -3 \end{array} \right\}$$

There is no inequality where  $x_1$  is absent. Rearrange the system so that the single inequality with a positive

coefficient for  $x_1$  goes to the top:

$$\left. \begin{array}{l} \frac{17}{33}x_1 \leq \frac{53}{33} \\ -\frac{29}{22}x_1 \leq -\frac{49}{22} \\ -2x_1 \leq -3 \end{array} \right\}$$

Divide each inequality by the corresponding coefficient of  $x_1$  (and reverse the direction if that coefficient is negative):

$$\left. \begin{array}{l} x_1 \leq \frac{53}{17} \\ x_1 \geq \frac{49}{29} \\ x_1 \geq \frac{3}{2} \end{array} \right\}$$

The minimum value of  $x_1$  is  $\max\left(\frac{49}{29}, \frac{3}{2}\right)$  or  $\frac{49}{29}$ , and the maximum value is  $\frac{53}{17}$ . Since the range

$$\frac{49}{29} \leq x_1 \leq \frac{53}{17} \quad (7)$$

is nonempty, the original system (4) has a real solution. All solutions have the form  $(x_1, x_2)$ , where  $x_1$  is a real number in the range (7) and  $x_2$  satisfies (6) for that value of  $x_1$ . Clearly,  $(x_1, x_2) = (2, \frac{1}{2})$  is one such solution.

Next, explore if (4) has an integer solution. The set of all integer vectors  $(z_1, z_2)$  satisfying the four inequalities in (4) is given by

$$\alpha_1 \leq z_1 \leq \beta_1$$

$$\alpha_2(z_1) \leq z_2 \leq \beta_2(z_1),$$

where

$$\alpha_1 = \left\lceil \frac{49}{29} \right\rceil = 2 \quad \text{and} \quad \beta_1 = \left\lfloor \frac{53}{17} \right\rfloor = 3,$$

$$\alpha_2(z_1) = \left\lceil \frac{2}{11}z_1 - \frac{3}{11} \right\rceil \quad \text{and}$$

$$\beta_2(z_1) = \left\lfloor \min\left(\frac{3}{2}z_1 - \frac{5}{2}, -\frac{1}{3}z_1 + \frac{4}{3}\right) \right\rfloor.$$

The only possible values of  $z_1$  are 2 and 3. It is easy to show that for either value of  $z_1$ , a corresponding integral value for  $z_2$  does not exist. Thus, there are real solutions to the system of inequalities (4), but no integer solutions.

### Unimodular Matrices

An  $m \times m$  real matrix  $\mathbf{A} = (a_{rt})$  represents a linear mapping of  $\mathbb{R}^m$  into itself that maps a vector  $\mathbf{x} = (x_1, x_2, \dots, x_m)$  into the vector  $\mathbf{x}\mathbf{A} = (y_1, y_2, \dots, y_m)$ , where

$$y_t = \sum_{r=1}^m a_{rt}x_r \quad (1 \leq t \leq m).$$

This mapping is a bijection (one-to-one and onto) if and only if  $\mathbf{A}$  is invertible, that is,  $\det \mathbf{A} \neq 0$ .

A *unimodular matrix* is a square integer matrix whose determinant is 1 or  $-1$ . The product of two unimodular matrices is unimodular. A unimodular matrix is invertible and its inverse is also unimodular. Thus, an  $m \times m$  unimodular matrix  $\mathbf{U}$  maps each integer vector  $\mathbf{x} \in \mathbb{R}^m$  into an integer vector  $\mathbf{x}\mathbf{U} \in \mathbb{R}^m$ , and for each integer vector  $\mathbf{y} \in \mathbb{R}^m$ , there is a unique integer vector  $\mathbf{x} \in \mathbb{R}^m$  such that  $\mathbf{y} = \mathbf{x}\mathbf{U}$ . In other words, an  $m \times m$  unimodular matrix defines a bijection of  $\mathbb{Z}^m$  onto itself. See [4, 11] for detailed discussions on unimodular matrices.

For any  $m$ , the  $m \times m$  unit matrix is unimodular. More unimodular matrices can be easily constructed from a unit matrix by elementary row (or column) operations. One gets a *reversal matrix* by multiplying a row of a unit matrix by  $-1$ , an *interchange matrix* by interchanging two rows of a unit matrix, and a *skewing matrix* by adding an integer multiple of one row to another row. These are called the *elementary matrices*.

The  $3 \times 3$  matrices

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \text{ and } \begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

are examples of a reversal, an interchange, and a skewing matrix, respectively. Elementary matrices are all unimodular. And each unimodular matrix can be expressed as the product of a finite number of elementary matrices. (See Lemma 2.3 in [4].)

A *permutation* of a finite set is a one-to-one mapping of the set onto itself. A permutation matrix is any matrix that can be obtained by permuting the columns (or rows) of a unit matrix. More formally, a *permutation matrix* is a square matrix of 0's and 1's, such that each row has exactly one 1 and each column has exactly one 1. An interchange matrix is a permutation matrix. A product of interchange matrices is clearly a permutation

matrix. The converse is also true: every permutation matrix can be expressed as a finite product of interchange matrices. In fact, every permutation matrix can be expressed as a finite product of interchange matrices each of which interchanges two adjacent columns (or rows). A permutation matrix is unimodular. (See Sect. 2.5 in [4].)

Let  $\mathbf{U}$  denote any  $m \times m$  permutation matrix. For  $1 \leq i \leq m$ , let  $\pi(i)$  denote the number of the row containing the unique 1 in column  $i$ . Then, the function  $\pi : i \mapsto \pi(i)$  is a permutation of the set  $\{1, 2, \dots, m\}$ , and it completely specifies the matrix  $\mathbf{U}$ . For any  $m$ -vector  $\mathbf{x} = (x_1, x_2, \dots, x_m)$ , one has

$$\mathbf{x}\mathbf{U} = (x_{\pi(1)}, x_{\pi(2)}, \dots, x_{\pi(m)}). \quad (8)$$

### Basic Concepts

The program model is a perfect nest of loops  $\mathbf{L} = (L_1, L_2, \dots, L_m)$  shown in Fig. 1. For  $1 \leq r \leq m$ , the index variable  $I_r$  of  $L_r$  runs from  $p_r$  to  $q_r$  in steps of 1, where  $p_r$  and  $q_r$  are integer-valued linear functions of  $I_1, I_2, \dots, I_{r-1}$ . The *index vector* of the loop nest is  $\mathbf{I} = (I_1, I_2, \dots, I_m)$ . An *index point* or *index value* of the nest is a possible value of the index vector, that is, a vector  $\mathbf{i} = (i_1, i_2, \dots, i_m) \in \mathbb{Z}^m$  such that  $p_r \leq i_r \leq q_r$  for  $1 \leq r \leq m$ . The subset  $\mathcal{R}$  of  $\mathbb{Z}^m$  consisting of all index points is the *index space* of the loop nest. During sequential execution of the nest, the index vector traverses the entire index space in the lexicographic order.

The *body* of the loop nest  $\mathbf{L}$  is denoted by  $H(\mathbf{I})$ . A given index value  $\mathbf{i}$  defines a particular *instance*  $H(\mathbf{i})$  of  $H(\mathbf{I})$ , which is an *iteration* of  $\mathbf{L}$ . The program of Fig. 1 consists of the set of iterations  $\{H(\mathbf{i}) : \mathbf{i} \in \mathcal{R}\}$ , and they are executed sequentially in such a way that an iteration  $H(\mathbf{i})$  comes before an iteration  $H(\mathbf{j})$  if and only if  $\mathbf{i} < \mathbf{j}$ .

There is *dependence* between two distinct iterations  $H(\mathbf{i})$  and  $H(\mathbf{j})$  if there is a memory location that is referenced (read or written) by both. (Sometimes, it may be required that at least one of the references be a “write.”) Suppose there is dependence between  $H(\mathbf{i})$  and  $H(\mathbf{j})$ .

```

 $L_1 : \quad \text{do } I_1 = p_1, q_1$
 $L_2 : \quad \text{do } I_2 = p_2, q_2$
 $\vdots \quad \quad \quad \vdots$
 $L_m : \quad \text{do } I_m = p_m, q_m$
 $H(\mathbf{I})$

```

Unimodular Transformations. Fig. 1 Loop nest  $\mathbf{L}$

Then  $H(\mathbf{j})$  depends on  $H(\mathbf{i})$  if  $H(\mathbf{j})$  is executed after  $H(\mathbf{i})$ , that is, if  $\mathbf{i} < \mathbf{j}$ .

If  $H(\mathbf{j})$  depends on  $H(\mathbf{i})$ , then  $\mathbf{d} = \mathbf{j} - \mathbf{i}$  is a (*dependence*) *distance vector* of the loop nest. Since  $\mathbf{i} < \mathbf{j}$ , a distance vector is necessarily positive. To keep matters simple, it is assumed that each distance vector  $\mathbf{d}$  is *uniform* in the sense that whenever  $\mathbf{i}$  and  $\mathbf{i} + \mathbf{d}$  are two index points, the iteration  $H(\mathbf{i} + \mathbf{d})$  depends on the iteration  $H(\mathbf{i})$ . Let  $N$  denote the total number of distinct distance vectors. Then the *distance matrix* of  $\mathbf{L}$  is an  $N \times m$  matrix  $\mathbf{D}$  whose rows are the distance vectors. The distance matrix is unique up to a permutation of its rows.

If  $\mathbf{d}$  is a distance vector of  $\mathbf{L}$ , then the direction vector of  $\mathbf{d}$  is a (*dependence*) *direction vector* of the loop nest. A direction vector of  $\mathbf{L}$  is necessarily positive. The *direction matrix* of  $\mathbf{L}$  is a matrix  $\Delta$  with  $m$  columns whose rows are the direction vectors. The direction matrix is unique up to a permutation of its rows.

A loop  $L_r$  in the nest  $\mathbf{L}$ , where  $1 \leq r \leq m$ , carries a *dependence* if there is a distance vector  $\mathbf{d}$  with  $\mathbf{d} >_r \mathbf{0}$ . A loop can *run in parallel* if it carries no dependence. According to this definition, a trivial loop with a single iteration can run in parallel. This is helpful in avoiding special cases.

Let  $\mathbf{U}$  denote an  $m \times m$  unimodular matrix. The image of  $\mathcal{R}$  under  $\mathbf{U}$  is the set  $\{\mathbf{IU} : \mathbf{I} \in \mathcal{R}\}$ . By using Fourier's method of elimination, one can get a suitable description for this image. Note that the index space  $\mathcal{R}$  consists of all integer  $m$ -vectors  $\mathbf{I} = (I_1, I_2, \dots, I_m)$  that satisfy the system of inequalities:

$$p_r \leq I_r \leq q_r \quad (1 \leq r \leq m), \quad (9)$$

where  $p_r$  and  $q_r$  are integer-valued linear functions of  $I_1, I_2, \dots, I_{r-1}$ . Let  $\mathbf{K} = \mathbf{IU}$ , so that  $\mathbf{I} = \mathbf{KU}^{-1}$ . If  $\mathbf{K} = (K_1, K_2, \dots, K_m)$  and  $\mathbf{U}^{-1} = (v_{tr})$ , then  $I_r = \sum_{t=1}^m v_{tr} K_t$  for each  $r$ . Replacing each  $I_r$  in (9) with the corresponding expression in  $K_1, K_2, \dots, K_m$ , one gets a system of linear inequalities in  $K_1, K_2, \dots, K_m$ . Fourier's method then yields an equivalent system of inequalities of the following type:

$$\alpha_r \leq K_r \leq \beta_r \quad (1 \leq r \leq m), \quad (10)$$

where  $\alpha_r$  and  $\beta_r$  are integer-valued functions of  $K_1, K_2, \dots, K_{r-1}$ . The image of  $\mathcal{R}$  under  $\mathbf{U}$  consists of all integer  $m$ -vectors  $\mathbf{K} = (K_1, K_2, \dots, K_m)$  that satisfy (10).

The loop nest  $\mathbf{L}'$  of Fig. 2, where  $H'(\mathbf{K}) = H(\mathbf{KU}^{-1})$ , defines the *unimodular transformation* of the loop nest  $\mathbf{L}$

|          |                                      |
|----------|--------------------------------------|
| $L'_1 :$ | $\text{do } K_1 = \alpha_1, \beta_1$ |
| $L'_2 :$ | $\text{do } K_2 = \alpha_2, \beta_2$ |
| $\vdots$ | $\vdots$                             |
| $L'_m :$ | $\text{do } K_m = \alpha_m, \beta_m$ |
|          | $H'(\mathbf{K})$                     |

Unimodular Transformations. Fig. 2 Loop nest  $\mathbf{L}'$

induced by the unimodular matrix  $\mathbf{U}$ . The transformed program has the same set of iterations as the original program, executing in a different sequential order. Two iterations  $H(\mathbf{i})$  and  $H(\mathbf{j})$  in the original nest  $\mathbf{L}$  become iterations  $H'(\mathbf{k})$  and  $H'(\mathbf{l})$ , respectively, in the transformed nest  $\mathbf{L}'$ , where  $\mathbf{k} = \mathbf{iU}$  and  $\mathbf{l} = \mathbf{jU}$ . In  $\mathbf{L}$ ,  $H(\mathbf{i})$  precedes  $H(\mathbf{j})$  if  $\mathbf{i} < \mathbf{j}$ . But in  $\mathbf{L}'$ ,  $H'(\mathbf{k})$  precedes  $H'(\mathbf{l})$  if  $\mathbf{k} < \mathbf{l}$ , that is, if  $\mathbf{iU} < \mathbf{jU}$ . The loop nest  $\mathbf{L}'$  is *equivalent* to  $\mathbf{L}$ , if whenever  $H(\mathbf{j})$  depends on  $H(\mathbf{i})$  in  $\mathbf{L}$ ,  $H'(\mathbf{k})$  precedes  $H'(\mathbf{l})$  in  $\mathbf{L}'$ , that is,  $\mathbf{iU} < \mathbf{jU}$ . The transformation  $\mathbf{L} \Rightarrow \mathbf{L}'$  is *valid* if  $\mathbf{L}'$  is equivalent to  $\mathbf{L}$ .

**Theorem 1** *The transformation  $\mathbf{L} \Rightarrow \mathbf{L}'$  induced by a unimodular matrix  $\mathbf{U}$  is valid if and only if  $\mathbf{dU} > \mathbf{0}$  for each distance vector  $\mathbf{d}$  of  $\mathbf{L}$ .*

*Proof* The “if” Part. Suppose  $\mathbf{dU} > \mathbf{0}$  for each distance vector  $\mathbf{d}$  of  $\mathbf{L}$ . Let an iteration  $H(\mathbf{j})$  depend on an iteration  $H(\mathbf{i})$ . Since  $\mathbf{j} - \mathbf{i}$  is a distance vector of  $\mathbf{L}$ , one gets  $\mathbf{jU} - \mathbf{iU} = (\mathbf{j} - \mathbf{i})\mathbf{U} > \mathbf{0}$ . Hence,  $\mathbf{iU} < \mathbf{jU}$ . Therefore,  $\mathbf{L}'$  is equivalent to  $\mathbf{L}$ , and the transformation  $\mathbf{L} \Rightarrow \mathbf{L}'$  is valid.

The “only if” Part. Suppose the transformation  $\mathbf{L} \Rightarrow \mathbf{L}'$  is valid. Let  $\mathbf{d}$  denote any distance vector of  $\mathbf{L}$ . There exist index points  $\mathbf{i}$  and  $\mathbf{j}$  of  $\mathbf{L}$ , such that  $\mathbf{d} = \mathbf{j} - \mathbf{i}$  and the iteration  $H(\mathbf{j})$  depends on the iteration  $H(\mathbf{i})$ . By hypothesis,  $\mathbf{iU} < \mathbf{jU}$ , so that  $\mathbf{dU} = \mathbf{jU} - \mathbf{iU} > \mathbf{0}$ . ■

**Corollary 1** *If the transformation  $\mathbf{L} \Rightarrow \mathbf{L}'$  by a unimodular matrix  $\mathbf{U}$  is valid, then the distance vectors of  $\mathbf{L}'$  are the vectors  $\mathbf{dU}$  where  $\mathbf{d}$  is any distance vector of  $\mathbf{L}$ .*

*Proof* There is dependence between two iterations  $H'(\mathbf{k})$  and  $H'(\mathbf{l})$  of  $\mathbf{L}'$  if and only if there is dependence between the iterations  $H(\mathbf{i})$  and  $H(\mathbf{j})$  of  $\mathbf{L}$ , where  $\mathbf{k} = \mathbf{iU}$  and  $\mathbf{l} = \mathbf{jU}$ . Since  $\mathbf{l} - \mathbf{k} = (\mathbf{j} - \mathbf{i})\mathbf{U}$ , the distance vectors of  $\mathbf{L}'$  are vectors of the form  $\pm \mathbf{dU}$  where  $\mathbf{d}$  is a distance vector of  $\mathbf{L}$ . Now a distance vector has to be positive. Since  $\mathbf{dU} > \mathbf{0}$  for each  $\mathbf{d}$  (Theorem 1), it follows that the distance vectors of  $\mathbf{L}'$  are precisely the vectors  $\mathbf{dU}$ . ■

## Parallelization by Unimodular Transformation

The goal of this section is to explain by examples how a loop nest can be transformed by a unimodular matrix, such that the innermost or outermost loops in the transformed program can run in parallel. Detailed algorithms are given in the essay *Loop Nest Parallelization* in this encyclopedia.

*Example 2* (Inner Loop Parallelization) Consider a nest  $\mathbf{L}$  of two loops:

$$L_1 : \quad \text{do } I_1 = 0, 3$$

$$L_2 : \quad \text{do } I_2 = 0, 3$$

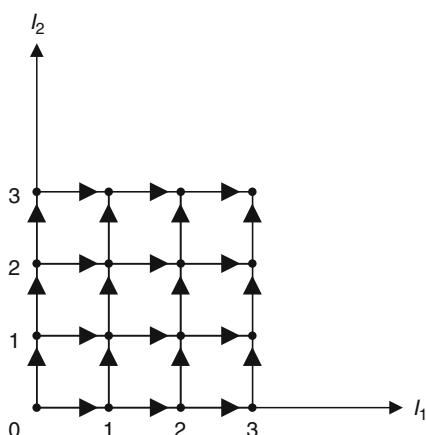
$$H(\mathbf{I}) : \quad A(I_1, I_2) = A(I_1 - 1, I_2) + A(I_1, I_2 - 1)$$

The index space  $\mathcal{R}$  of  $\mathbf{L}$  is given by

$$\mathcal{R} = \{(I_1, I_2) : 0 \leq I_1 \leq 3, 0 \leq I_2 \leq 3\}.$$

The index space and the dependence between iterations are shown in Fig. 3. The only distance vectors of  $\mathbf{L}$  are  $(1, 0)$  and  $(0, 1)$ . Each of the two loops carries a dependence, and hence cannot run in parallel.

Let  $\mathbf{U} = (u_{rt})$  denote any  $2 \times 2$  unimodular matrix. Let the unimodular transformation of  $\mathbf{L}$  induced by  $\mathbf{U}$  result in a loop nest  $\mathbf{L}' = (L'_1, L'_2)$ . By Theorem 1,  $\mathbf{L}'$  is equivalent to  $\mathbf{L}$  if  $(1, 0)\mathbf{U} > (0, 0)$  and  $(0, 1)\mathbf{U} > (0, 0)$ . Assume  $\mathbf{L}'$  is equivalent to  $\mathbf{L}$ . Then, by Corollary 1 to Theorem 1, the distance vectors of  $\mathbf{L}'$  are  $(1, 0)\mathbf{U}$  and  $(0, 1)\mathbf{U}$ . The inner loop  $L'_2$  of  $\mathbf{L}'$  can run in parallel if it carries no dependence, that is, if  $(1, 0)\mathbf{U} >_2 (0, 0)$  and



Unimodular Transformations. Fig. 3 Index space of  $\mathbf{L}$

$(0, 1)\mathbf{U} >_2 (0, 0)$  are both false. Thus, to get an equivalent nest  $\mathbf{L}'$  whose inner loop can run in parallel, one must have  $(1, 0)\mathbf{U} >_1 (0, 0)$  and  $(0, 1)\mathbf{U} >_1 (0, 0)$ , that is,  $(u_{11}, u_{12}) >_1 (0, 0)$  and  $(u_{21}, u_{22}) >_1 (0, 0)$ , or  $u_{11} > 0$  and  $u_{21} > 0$ .

A simple unimodular matrix  $\mathbf{U}$  that fits these constraints, and its inverse are shown below:

$$\mathbf{U} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \quad \text{and} \quad \mathbf{U}^{-1} = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}.$$

The equation  $(I_1, I_2) = (K_1, K_2)\mathbf{U}^{-1}$  gives  $I_1 = K_1 - K_2$  and  $I_2 = K_2$ . Since  $0 \leq I_1 \leq 3$  and  $0 \leq I_2 \leq 3$ , one gets the inequalities  $0 \leq K_1 - K_2 \leq 3$  and  $0 \leq K_2 \leq 3$  that lead to the ranges (by Fourier's method):

$$\left. \begin{array}{l} 0 \leq K_1 \leq 6 \\ \max(0, K_1 - 3) \leq K_2 \leq \min(3, K_1). \end{array} \right\}$$

Thus, the given program  $\mathbf{L}$  is equivalent to the loop nest  $\mathbf{L}'$ :

$$L'_1 : \quad \text{do } K_1 = 0, 6$$

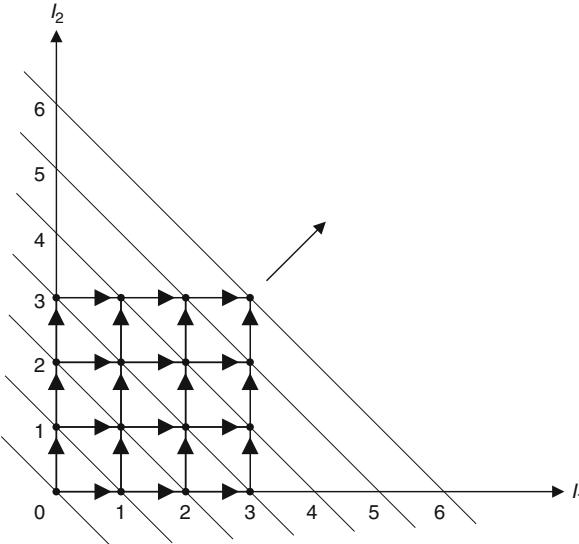
$$L'_2 : \quad \text{do } K_2 = \max(0, K_1 - 3), \min(3, K_1)$$

$$H'(\mathbf{K}) : A(K_1 - K_2, K_2) = A(K_1 - K_2 - 1, K_2) \\ + A(K_1 - K_2, K_2 - 1)$$

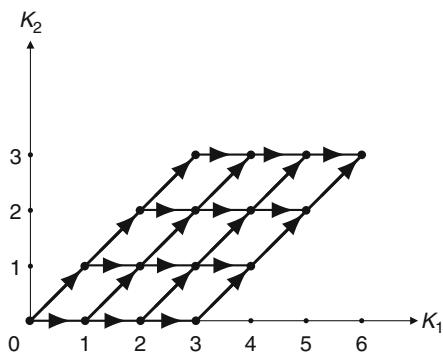
where the new body  $H'(\mathbf{K})$  is obtained from  $H(\mathbf{I})$  by replacing  $I_1$  with  $K_1 - K_2$  and  $I_2$  with  $K_2$ . The index space of the transformed loop nest  $\mathbf{L}'$  is shown in Fig. 5. The distance vectors of  $\mathbf{L}'$  are  $(1, 0)$  and  $(1, 1)$ . The inner loop  $L'_2$  can now run in parallel since it carries no dependence.

In Fig. 4, consider the set of 7 parallel lines  $I_1 + I_2 = c$ , where  $0 \leq c \leq 6$ . They correspond to the 7 values of  $K_1$ . On any given line, the index points correspond to the values of  $K_2$  for a fixed  $K_1$ . Executing  $L'_1$  sequentially and  $L'_2$  in parallel means that these lines are taken sequentially from  $c = 0$  to  $c = 6$ , while iterations for index points on each given line are executed in parallel. Since the lines appear to constitute a “wave” through the index space of  $\mathbf{L}$ , the name *wavefront method* is often given to this process of executing a loop nest.

In the essay ►*Loop Nest Parallelization*, a general algorithm is given that finds a valid unimodular transformation for any given loop nest such that all but



**Unimodular Transformations. Fig. 4** Wave through the index space of  $\mathbf{L}$



**Unimodular Transformations. Fig. 5** Index space of  $\mathbf{L}'$  in Example 2

the outermost loop in the transformed nest can run in parallel. See also Remark 2.

*Example 3 (Outer Loop Parallelization)* Consider the loop nest  $\mathbf{L}$ :

$L_1 :$       **do**  $I_1 = 5, 100$   
 $L_2 :$       **do**  $I_2 = 16, 80$   
 $H(\mathbf{I}) :$      $X(I_1, I_2) = X(I_1 - 2, I_2 - 4)$   
                       $+ X(I_1 - 3, I_2 - 6)$

The only distance vectors of  $\mathbf{L}$  are  $(2, 4)$  and  $(3, 6)$ . Since the outer loop carries a dependence, it cannot run in parallel.

Let  $\mathbf{U} = (u_{rt})$  denote any  $2 \times 2$  unimodular matrix. Let the unimodular transformation of  $\mathbf{L}$  induced by  $\mathbf{U}$  result in a loop nest  $\mathbf{L}' = (L'_1, L'_2)$ . By Theorem 1,  $\mathbf{L}'$  is equivalent to  $\mathbf{L}$  if  $(2, 4)\mathbf{U} > (0, 0)$  and  $(3, 6)\mathbf{U} > (0, 0)$ . Assume  $\mathbf{L}'$  is equivalent to  $\mathbf{L}$ . Then, by Corollary 1 to Theorem 1, the distance vectors of  $\mathbf{L}'$  are  $(2, 4)\mathbf{U}$  and  $(3, 6)\mathbf{U}$ . The outer loop  $L'_1$  of  $\mathbf{L}'$  can run in parallel if it carries no dependence, that is, if  $(2, 4)\mathbf{U} >_1 (0, 0)$  and  $(3, 6)\mathbf{U} >_1 (0, 0)$  are both false. Thus, to get an equivalent nest  $\mathbf{L}'$  whose outer loop can run in parallel, one must have  $(2, 4)\mathbf{U} >_2 (0, 0)$  and  $(3, 6)\mathbf{U} >_2 (0, 0)$ . The conditions on the elements of  $\mathbf{U}$  are then

$$\left. \begin{array}{l} 2u_{11} + 4u_{21} = 0 \\ 2u_{12} + 4u_{22} > 0 \\ 3u_{11} + 6u_{21} = 0 \\ 3u_{12} + 6u_{22} > 0. \end{array} \right\}$$

A simple unimodular matrix  $\mathbf{U}$  that fits these constraints, and its inverse are shown below:

$$\mathbf{U} = \begin{pmatrix} -2 & 1 \\ 1 & 0 \end{pmatrix} \quad \text{and} \quad \mathbf{U}^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & 2 \end{pmatrix}.$$

The limits on  $I_1$  and  $I_2$  are given by  $5 \leq I_1 \leq 100$  and  $16 \leq I_2 \leq 80$ . Since

$$(I_1, I_2) = (K_1, K_2)\mathbf{U}^{-1} = (K_2, K_1 + 2K_2),$$

the constraints in terms of  $K_1$  and  $K_2$  are

$$5 \leq K_2 \leq 100$$

$$16 \leq K_1 + 2K_2 \leq 80.$$

By Fourier's method of elimination, one gets

$$-184 \leq K_1 \leq 70$$

$$[\max(5, 8 - K_1/2)] \leq K_2 \leq [\min(100, 40 - K_1/2)].$$

Thus, the given program  $\mathbf{L}$  is equivalent to the loop nest  $\mathbf{L}'$ :

$L'_1 :$       **do**  $K_1 = -184, 70$   
 $L'_2 :$       **do**  $K_2 = [\max(5, 8 - K_1/2)],$   
                       $[\min(100, 40 - K_1/2)]$   
 $H'(\mathbf{K}) :$   $X(K_2, K_1 + 2K_2) = X(K_2 - 2, K_1 + 2K_2 - 4)$   
                       $+ X(K_2 - 3, K_1 + 2K_2 - 6)$

where the new body  $H'(\mathbf{K})$  is obtained from  $H(\mathbf{I})$  by replacing  $I_1$  with  $K_2$  and  $I_2$  with  $K_1 + 2K_2$ . The distance vectors of  $\mathbf{L}'$  are  $(0, 2)$  and  $(0, 3)$ . The outer loop  $L'_1$  can now run in parallel since it carries no dependence.

In general, if the rank of the distance matrix for  $\mathbf{L}$  is  $\rho$ , then a unimodular matrix  $\mathbf{U}$  can always be found such that the outermost  $(m - \rho)$  loops  $L'_1, L'_2, \dots, L'_{m-\rho}$  of  $\mathbf{L}'$  can run in parallel. Using inner loop parallelization techniques on top of this, one may get an equivalent loop nest where all loops except  $L'_{m-\rho+1}$  can run in parallel. Details are given in the essay ▶ [Loop Nest Parallelization](#).

There are some special unimodular transformations that can be handled more easily than a general transformation. They are described next.

### Loop Permutation

Again, the model program is the loop nest  $\mathbf{L}$  of Fig. 1, and the transformation of  $\mathbf{L}$  induced by a unimodular matrix  $\mathbf{U}$  results in the loop nest  $\mathbf{L}'$  of Fig. 2. Let  $\mathbf{U}$  denote a permutation matrix. Then the transformation  $\mathbf{L} \Rightarrow \mathbf{L}'$  is a *loop permutation*. If  $\pi$  is the corresponding permutation of the set  $\{1, 2, \dots, m\}$ , then the index vector  $\mathbf{K} = (K_1, K_2, \dots, K_m)$  of  $\mathbf{L}'$  is given by  $K_r = I_{\pi(r)}$  for  $1 \leq r \leq m$ . (See (8).)

One important feature of loop permutations is that their validity can be determined from a knowledge of direction vectors alone. This is quite useful since the number of direction vectors is never more (and often less) than the number of distance vectors, and they may be easier to compute and store than distance vectors.

**Theorem 2** *Let  $\mathbf{U}$  denote an  $m \times m$  permutation matrix and  $\pi$  the corresponding permutation of the set  $\{1, 2, \dots, m\}$ . The loop permutation  $\mathbf{L} \Rightarrow \mathbf{L}'$  induced by  $\mathbf{U}$  is valid, if and only if for each direction vector  $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_m)$  of  $\mathbf{L}$ , the vector  $(\sigma_{\pi(1)}, \sigma_{\pi(2)}, \dots, \sigma_{\pi(m)})$  is positive.*

*Proof* The “if” Part. Suppose the condition holds. Let  $\mathbf{d} = (d_1, d_2, \dots, d_m)$  denote a distance vector of  $\mathbf{L}$ . Let  $\sigma$  be the direction vector of  $\mathbf{d}$ . Since  $\sigma_i = \text{sgn}(d_i)$ , it follows that  $(\text{sgn}(d_{\pi(1)}), \text{sgn}(d_{\pi(2)}), \dots, \text{sgn}(d_{\pi(m)}))$  is positive. This means  $(d_{\pi(1)}, d_{\pi(2)}, \dots, d_{\pi(m)}) > \mathbf{0}$ , that is,  $\mathbf{d}\mathbf{U} > \mathbf{0}$ . (See (8).) Hence the transformation  $\mathbf{L} \Rightarrow \mathbf{L}'$  is valid by Theorem 1.

The “only if” Part can be proved similarly. ■

**Corollary 1** *If the loop permutation  $\mathbf{L} \Rightarrow \mathbf{L}'$  is valid, then the direction vectors of  $\mathbf{L}'$  are the vectors  $(\sigma_{\pi(1)}, \sigma_{\pi(2)}, \dots, \sigma_{\pi(m)})$ , where  $(\sigma_1, \sigma_2, \dots, \sigma_m)$  is any direction vector of  $\mathbf{L}$ .*

A direction vector  $(\sigma_1, \sigma_2, \dots, \sigma_m) > \mathbf{0}$  prevents the loop permutation defined by a permutation  $\pi$  of  $\{1, 2, \dots, m\}$  if  $(\sigma_{\pi(1)}, \sigma_{\pi(2)}, \dots, \sigma_{\pi(m)}) < \mathbf{0}$ . A very useful permutation is the interchange of two loops  $L_r$  and  $L_{r+1}$  in  $\mathbf{L}$ , where  $1 \leq r < m$ . The only direction vectors that would prevent it have the form  $(0, 0, \dots, 0, 1, -1, *, *, \dots, *)$ , where there are  $(r - 1)$  zeros in the front and “ $*$ ” indicates any member of the set  $\{0, 1, -1\}$ . Algorithm 2.1 in [5] shows how to find all direction vectors that prevent the permutation of a loop nest  $\mathbf{L}$  defined by any given permutation  $\pi$  of  $\{1, 2, \dots, m\}$ .

*Remark 1* Loop permutation is often used to achieve a desirable memory access pattern. A few comments are made here about inner and outer loop parallelization using permutation (see [5]). Let  $1 \leq r \leq m$ . Note that the loop  $L_r$  carries a dependence if and only if there is a direction vector of the form  $(0, 0, \dots, 0, 1, *, \dots, *)$  with  $(r - 1)$  leading zeros. Consider these special cases:

1. Column  $r$  of the direction matrix  $\Delta$  has only 0’s. By a suitable loop permutation,  $L_r$  can be moved inward or outward to any position, and it can run in parallel there. (When a loop moves, its limits may change.)
2. Column  $r$  of  $\Delta$  has only 0’s and 1’s. Then  $L_r$  can move outward to any position.
3. Column  $r$  of  $\Delta$  has only 1’s. Then  $L_r$  can move outward to any position. If it is made the outermost loop, then no other loop in  $\mathbf{L}'$  will carry a dependence and each inner loop can run in parallel.
4.  $L_r$  carries no dependence. Then  $L_r$  can move inward to any position where it can run in parallel. (Even when  $L_r$  carries a dependence, it may still be possible to move it inward to a position where it runs in parallel.)
5. If there is no  $-1$  in the direction matrix  $\Delta$ , then every permutation of the nest  $\mathbf{L}$  is valid.

### Loop Skewing

Let  $a$  and  $b$  denote two integers such that  $1 \leq a < b \leq m$ . Let  $\mathbf{U}$  denote the matrix obtained from the  $m \times m$  unit matrix by replacing the 0 on row  $a$  and column  $b$  with an

integer  $z$ . Then  $\mathbf{U}$  is a skewing matrix and unimodular. The transformation  $\mathbf{L} \Rightarrow \mathbf{L}'$  induced by  $\mathbf{U}$  is a *skewing* of the loop  $L_b$  by the loop  $L_a$  with skewing factor  $z$ . (More precisely, it is an *upper* skewing.) The index vector  $\mathbf{K}$  of  $\mathbf{L}'$  is given by

$$\mathbf{K} = \mathbf{IU} = (I_1, \dots, I_a, \dots, I_{b-1}, I_b + zI_a, I_{b+1}, \dots, I_m).$$

**Theorem 3** Let  $1 \leq a < b \leq m$ . A skewing of the loop  $L_b$  in the nest  $\mathbf{L}$  by the loop  $L_a$  is always valid.

*Proof* Let  $\mathbf{d} = (d_1, d_2, \dots, d_m)$  denote any distance vector of  $\mathbf{L}$ . Then

$$\mathbf{d}\mathbf{U} = (d_1, d_2, \dots, d_{b-1}, d_b + zd_a, d_{b+1}, \dots, d_m).$$

Since  $\mathbf{d} > \mathbf{0}$ , it follows that  $(d_1, d_2, \dots, d_{b-1}) \geq \mathbf{0}$ . If  $(d_1, d_2, \dots, d_{b-1}) > \mathbf{0}$ , then  $\mathbf{d}\mathbf{U} > \mathbf{0}$ . Suppose  $(d_1, d_2, \dots, d_{b-1}) = \mathbf{0}$ . Then  $d_a = 0$  since  $1 \leq a < b$ . Hence,  $\mathbf{d}\mathbf{U} = (0, 0, \dots, 0, d_b, d_{b+1}, \dots, d_m) = \mathbf{d} > \mathbf{0}$ . Now apply Theorem 1. ■

**Corollary 1** After a skewing of a loop  $L_b$  by a loop  $L_a$  with a skewing factor  $z$ , the distance vectors of the transformed nest  $\mathbf{L}'$  are vectors of the form  $(d_1, d_2, \dots, d_{b-1}, d_b + zd_a, d_{b+1}, \dots, d_m)$ , where  $(d_1, d_2, \dots, d_m)$  is any distance vector of  $\mathbf{L}$ .

**Remark 2** Let  $\mathbf{d} = (d_1, d_2, \dots, d_m)$  be a distance vector of  $\mathbf{L}$ . Since  $\mathbf{d}$  is positive, it must be of the form  $\mathbf{d} = (0, \dots, 0, d_a, d_{a+1}, \dots, d_m)$  where  $1 \leq a \leq m$  and  $d_a > 0$ . Suppose there is a  $b$  such that  $a < b \leq m$  and  $d_b < 0$ . Then skewing of the loop  $L_b$  by the loop  $L_a$  with the positive skewing factor  $[-d_b/d_a]$  will change  $\mathbf{d}$  into a distance vector  $(d_1, d_2, \dots, d_{b-1}, d'_b, d_{b+1}, \dots, d_m)$ , where  $d'_b \geq 0$ . For a given distance vector  $\mathbf{d}$  with a leading element  $d_a$ , it is possible to skew all such loops  $L_b$  in one step via a single unimodular matrix that is the product of a number of skewing matrices.

By systematically considering all distance vectors of the above type and applying loop skewing repeatedly, it is thus possible to find a loop nest  $\mathbf{L}'$  equivalent to  $\mathbf{L}$  such that the distance matrix  $\mathbf{D}'$  of  $\mathbf{L}'$  has all nonnegative entries. Furthermore, by a slight extension, one can make column  $m$  of  $\mathbf{D}'$  a vector of positive integers. Then column  $m$  of the direction matrix  $\Delta'$  of  $\mathbf{L}'$  will be a column of 1's. By Remark 1.3, the innermost loop  $L'_m$  of  $\mathbf{L}'$  can be made the outermost loop to create an equivalent program where all the  $(m - 1)$  inner loops can run in parallel. See the example below.

**Example 4** Consider a loop nest  $\mathbf{L} = (L_1, L_2, L_3)$  with the distance matrix

$$\mathbf{D} = \begin{pmatrix} 0 & 2 & -4 \\ 2 & -3 & 5 \\ 0 & 0 & 2 \end{pmatrix}.$$

By applying loop skewing repeatedly and a loop permutation at the end,  $\mathbf{D}$  can be transformed into the form described in Remark 2:

$$\begin{aligned} & \begin{pmatrix} 0 & 2 & -4 \\ 2 & -3 & 5 \\ 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 2 & 0 \\ 2 & -3 & -1 \\ 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} 1 & 2 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 2 & 0 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 2 & 2 \\ 2 & 1 & 2 \\ 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 2 & 0 & 2 \\ 2 & 2 & 1 \\ 2 & 0 & 0 \end{pmatrix} \end{aligned}$$

The index vector of the loop nest transforms as follows:

$$\begin{aligned} (I_1, I_2, I_3) &\Rightarrow (I_1, I_2, I_3 + 2I_2) \Rightarrow (I_1, I_2 + 2I_1, I_3 + 2I_2 + I_1) \\ &\Rightarrow (I_1, I_2 + 2I_1, I_3 + 3I_2 + 3I_1) \Rightarrow (I_3 + 3I_2 + 3I_1, I_1, I_2 + 2I_1) \\ &= (K_1, K_2, K_3), \end{aligned}$$

where  $(K_1, K_2, K_3)$  is the index vector of the final loop nest. Thus,  $(I_1, I_2, I_3) = (K_2, -2K_2 + K_3, K_1 + 3K_2 - 3K_3)$ . Limits for  $K_1, K_2, K_3$  can be computed from the limits for  $I_1, I_2, I_3$  by Fourier's method in the usual way. The two inner loops of the final nest do not carry a dependence, and hence they can run in parallel.

## Related Entries

- [Code Generation](#)
- [Parallelization, Automatic](#)
- [Loop Nest Parallelization](#)
- [Parallelism Detection in Nested Loops, Optimal](#)

## Bibliographic Notes and Further Reading

Research on unimodular transformations goes back to Leslie Lamport's paper [9] in 1974, although he did not

use this particular term. This essay is based on the theory of unimodular transformations as developed in the author's books on loop transformations [4, 5]. The general theory grew out of the theory of unimodular transformations of double loops described in [3]. (Watch for some notational differences between those references and the current essay.) The paper by Michael Wolf and Monica Lam [13] covers many aspects of unimodular transformations in detail. See also Michael Dowling [7], Erik H. D'Hollander [6], and François Irigoin and Rémi Triolet [8].

Loop interchange has been studied extensively by Michael Wolfe [15, 16], and by Randy Allen and Ken Kennedy [1, 2]. Implementation of the wavefront method by loop skewing and interchange was discussed by Wolfe in [14]. The same technique is described in a general setting in [13].

Wei Li and Keshav Pingali [10] discuss a loop transformation framework based on the class of nonsingular matrices (that includes unimodular matrices).

## Bibliography

1. Allen R, Kennedy K (Oct 1987) Automatic translation of FORTRAN programs to vector form. ACM Trans Program Lang Syst 9(4):491–542
2. Allen R, Kennedy K (Oct 2001) Optimizing compilers for modern architectures. Morgan Kaufmann, San Francisco
3. Banerjee U (1990) Unimodular transformations of double loops. In: Proceedings of the third workshop on languages and compilers for parallel computing, Irvine, 1–3 Aug 1990. Available as Nicolau A, Gelernter D, Gross T, Padua D (eds) (1991) Advances in languages and compilers for parallel computing. The MIT Press, Cambridge, pp 192–219
4. Banerjee U (1993) Loop transformations for restructuring compilers: the foundations. Kluwer Academic, Norwell
5. Banerjee U (1994) Loop transformations for restructuring compilers: loop parallelization. Kluwer Academic, Norwell
6. D'Hollander EH (July 1992) Partitioning and labeling of loops by unimodular transformations. IEEE Trans Parallel Distrib Syst 3(4):465–476
7. Dowling ML (Dec 1990) Optimal code parallelization using unimodular transformations. Parallel Comput 16(2–3):157–171
8. Irigoin F, Triolet R (1989) Dependence approximation and global parallel code generation for nested loops. Cosnard M et al. (eds) Parallel distrib algorithms. Elsevier (North-Holland), New York, pp 297–308
9. Lamport L (Feb 1974) The parallel execution of DO loops. Commun ACM 17(2):83–93
10. Li W, Pingali K (Apr 1994) A singular loop transformation framework based on non-singular matrices. Int J Parallel Program 22(2):183–205

11. Schrijver A (Oct 1987) Theory of linear and integer programming. Wiley, New York
12. Williams HP (Nov 1986) Fourier's method of linear programming and its dual. The American Mathematical Monthly 93(9):681–695
13. Wolf ME, Lam MS (Oct 1991) A loop transformation theory and an algorithm to maximize parallelism. IEEE Trans Parallel Distrib Syst 2(4):452–471
14. Wolfe M (Aug 1986) Loop skewing: the wavefront method revisited. Int J Parallel Program 15(4):279–293
15. Wolfe M (1986) Advanced loop interchanging. In: Proceedings of the 1986 international conference on parallel processing, St. Charles, 19–22 Aug 1986, pp 536–543. IEEE Computer Society Press, Los Angeles
16. Wolfe MJ (1989) Optimizing supercompilers for supercomputers. The MIT Press, Cambridge

## Universal VLSI Circuits

### ► Universality in VLSI Computation

## Universality in VLSI Computation

GIANFRANCO BILARDI<sup>1</sup>, GEPPINO PUCCI<sup>2</sup>

<sup>1</sup>University of Padova, Padova, Italy

<sup>2</sup>Università di Padova, Padova, Italy

## Synonyms

Area-universal networks; Universal VLSI circuits

## Definition

A VLSI circuit  $U(A)$  is said to be *area-universal* if it can be configured to emulate every VLSI circuit of a given area  $A$ . If  $U(A)$  has area  $A_U$ , then it has *blowup*  $\alpha = A_U/A$ . If any circuit with area-time bounds  $(A, T)$  is emulated by  $U(A)$  in time  $T_U \leq \sigma T$ , then  $U(A)$  has *slowdown*  $\sigma$ . Clearly, smaller blowup and smaller slowdown reflect a better quality of a universal circuit. An analogous formulation enables the study of area-universal general-purpose routing. The broad goal of research on area-universality is to characterize blowup/slowdown trade-offs, that is, what is the minimum blowup achievable for any given slowdown.

## Discussion

### Models of VLSI Computation

The VLSI model of computation (see Article [1] of this Encyclopedia) was formulated in the late seventies to capture the resource trade-offs of computation by large-scale integrated circuits. In this model, a computational engine is an interconnection of basic Boolean gates and storage elements (flip-flops), with a specified space layout defining the geometric placement of these building blocks (*nodes*) and of their interconnections (*wires*). Although the details of the model were developed with integrated circuit technology in mind, the spirit of the model is quite general and captures constraints that apply to any reasonable physical computing device. These key physical constraints can be stated as follows.

*P1 – Principle of information density.* There is a lower bound to the amount of space required to store a bit.

*P2 – Principle of bandwidth density.* There is an upper bound to the amount of information that can be transmitted through a unit of surface in a unit of time.

*P3 – Principle of computation delay.* There is a lower bound to the time necessary to compute a Boolean function of a given constant number of inputs.

*P4 – Principle of communication delay.* There is an upper bound to the speed at which information can travel.

The bounds stated by the above principles are assumed to be fixed positive constants. Any given technology typically comes to within constant factors of these bounds and technological progress aims, by and large, to an improvement of these factors.

Most of the research on VLSI computing has been based on Thompson's model [2], which does not actually include principle P4, as it assumes that information can be transmitted in constant time along a wire, irrespective of its length. The rationale was that, in the VLSI systems of the time, transmission delays did not add substantially to computation delays. An alternate VLSI model, including principle P4, was proposed by Chazelle and Monier [3]. A detailed study of how the predicted development of silicon technology would have impacted the choice between the two models was carried out by Bilardi, Pracchi, and Preparata in [4], indicating (correctly, in retrospective)

that the Thompson model would prove to be a reasonable approximation for a decade or more, at least with reference to single-chip systems. Furthermore, even when not applicable to the entire computing system, Thompson's model still leads to valuable insights on the design of sufficiently small regions of the system. In this entry, VLSI universality will be discussed in the regime where communication delays are negligible. In the complementary regime, governed by principle P4, computing systems do exhibit different behaviors, with significant impact on machine organization and algorithm design, as discussed by Bilardi and Preparata in [5].

### The Question of Universality

Many studies have utilized the VLSI model of computation to explore area-time trade-offs in special-purpose computing. Given a target problem, the objective is to determine, for any achievable computation time  $T$ , the minimum area  $A$  such that there is a VLSI circuit that can be laid out in area  $A$  and can solve the problem in time  $T$ . The area-time trade-off has been determined for several key problems. For example, the cumulative product of  $N$  elements drawn from a semigroup of fixed size can be computed in area  $A = \Theta(N/T)$ , where  $\Omega(\log N) \leq T \leq O(N)$ . For the addition of two  $N$ -bit numbers, the area becomes  $A = \Theta((N/T)\log(N/T))$ , where  $\Omega(\log N) \leq T \leq O(N)$ . Finally, the integer multiplication of two  $N$ -bit numbers, the sorting of  $n$  integer keys of  $2\log n$  bits each (hence  $N = 2n\log n$  input bits overall), and the  $n$ -point Discrete Fourier Transform in the ring of the integers modulo  $m$  (hence  $N = n\log m$  input bits overall), are all problems which feature optimal circuits of area  $A = \Theta(N^2/T^2)$ , where  $\Omega(\log N) \leq T \leq O(\sqrt{N})$ . The designs achieving the optimal area-time performance are typically based on different topologies for different problems, or even for different computation times concerning the same problem. Thus, a variety of interconnection topologies have been exploited for various operations, including *meshes* of various dimensions, the *tree*, the *shuffle-exchange*, the *cube connected cycles* and its *pleated* version, the *mesh-of-trees*, and several others. This state of affairs seems to lead to an unwelcome conclusion: if the optimal execution of different computations requires different architectures, then no architecture is universally good,

and general-purpose computing is inherently suboptimal. Is this conclusion really warranted? The theory of area-time universality in VLSI is a way to address this fundamental question.

To be specific and quantitative, the question for the case of circuits can be formulated as follows (the adaptation to routers will be discussed at the end of the next section). A VLSI circuit  $U(A)$  is *area-universal* if it can be programmed to emulate every circuit of a given area  $A$ . If  $U(A)$  has area  $A_U$ , then its *blowup* is  $\alpha = A_U/A$ . If any circuit with area-time bounds  $(A, T)$  is emulated by  $U_A$  in time  $T_U \leq \sigma T$ , then  $U(A)$  has *slowdown*  $\sigma$ . Clearly, smaller blowup and smaller slowdown reflect a better quality of a universal circuit. The general goal is to characterize the blowup–slowdown trade-off, that is, what is the minimum blowup achievable for any given slowdown.

In the VLSI model, the basic processing elements are Boolean gates with a fixed number, say  $q$ , of inputs and outputs. If the unit of length is chosen to reflect the feature size of the technology, then the area of an elementary gate is a small, integer constant. It is straightforward to design a  $q$ -universal gate of constant area  $a_q$ , programmable to compute any Boolean function of  $q$  inputs and outputs, in constant time  $\tau_q$ . A universal circuit  $U(A)$  can then be conceived as a set of  $A$  universal gates connected by a routing network programmable to simulate the interconnection of any specific circuit to be emulated. Since the universal Boolean gates contribute a constant factor both to the blowup and to the slowdown, the pivotal issue is the area-time performance of the routing network. Although a general permutation network could easily be adapted for the purpose, it would result in an inefficient solution. In fact, a data exchange that can be realized in unit time by a set of wires laid out in area  $A$  is considerably more constrained than a general permutation. To be area-time efficient, a router must take advantage of such constraints, which are formulated quantitatively in the next section.

## Bandwidth and Area

A key insight of the theory of VLSI layout is that the area of a graph is closely related to how well such graph can be embedded into a binary tree. This insight is systematically and beautifully developed by Bhatt and Leighton

in [6]. Below, the results relevant to the present discussion are reviewed and cast in the unifying terminology of graph embedding.

In an embedding, each vertex  $v \in V$  of a *guest* graph  $G = (V, E)$  is mapped onto a node  $\phi(v) \in W$  of a *host* graph  $H = (W, D)$ , and each edge  $(a, b) \in E$  of  $G$  is mapped onto a path  $\psi(a, b)$  joining node  $\phi(a)$  to node  $\phi(b)$  in tree  $H$ . The *load*  $\ell(w)$  of a host node  $w \in W$  is the number of guest vertices mapped onto  $w$ . The *dilation*  $d(a, b)$  of a guest edge  $(a, b) \in E$  is the length (number of edges) of the corresponding host path  $\psi(a, b)$ . The *congestion*  $c(u, w)$  of a host edge  $(u, w) \in D$  is the number of guest edges whose corresponding path includes edge  $(u, w)$ . Good embeddings are characterized by small values of load, dilation, and congestion.

A two-dimensional VLSI layout of a graph  $G$ , for simplicity assumed throughout of degree 4, can be defined as an embedding of  $G$  into a rectangular grid, subject to three constraints: (1) the load of each grid node is at most one; (2) the congestion of each grid edge is at most one; and (3) the grid path associated with a guest edge  $(a, b)$  can only traverse grid nodes of zero load, except for  $\phi(a)$  and  $\phi(b)$ . It turns out that a good layout of graph  $G$  leads rather straightforwardly to a good embedding of  $G$  in a binary tree; it is also the case that a good layout can be obtained from a good tree embedding, although the process is more subtle. Intuitively, searching for a good tree embedding is easier than searching for a good grid layout, due to the removal of constraints (1), (2), and (3) above and to the fact that a tree embedding is uniquely determined by vertex placement, if simple paths (which are unique for any choice of their endpoints) are used for hosting edges. The price to pay for the simplification is that, even if the given tree embedding is optimal, the area of the resulting layout may be suboptimal by up to two logarithmic factors.

A tree embedding can be obtained from a layout as follows. By simple transformations that increase the area by at most a constant factor, the layout is made to fit a  $\sqrt{A} \times \sqrt{A}$  square grid, where  $\sqrt{A}$  is an integer power of two. Next, each grid vertex is labeled with a pair of integer coordinates  $(i, j)$ , with  $i, j = 0, \dots, \sqrt{A}-1$ . The grid vertices are labeled so that vertex  $(0, 0)$  is the north-west corner of the grid, thus  $i$  increases eastward and  $j$  increases southward. Let  $H = (W, D)$  be a complete, ordered binary tree of  $A$  leaves, numbered  $0, \dots,$

$A - 1$ , from left to right. The tree embedding is based on the *shuffle-major* one-to-one correspondence that associates grid vertex  $(i, j)$  with leaf  $k$  of tree  $H$ , where the binary representation of  $k$  is obtained from those of  $i \equiv (i_{(\log A)/2-1} \dots i_0)$  and  $j \equiv (j_{(\log A)/2-1} \dots j_0)$  as  $k = \text{shm}(i, j) \equiv (j_{(\log A)/2-1} i_{(\log A)/2-1} \dots j_1 i_1 j_0 i_0)$ . Then, each node of the tree is naturally associated with the region of the grid matched to its descendant leaves. This association can be understood as a recursive decomposition, with the root corresponding to the entire layout, the left and the right children of the root, respectively, corresponding to the west and to the east halves of the layout, the four grandchildren of the root corresponding, from left to right, to the north-west, south-west, north-east, and south-east quadrants, and so on. With the preceding setup, a layout of a graph  $G = (V, E)$  naturally induces an embedding of  $G$  in  $H$  where (1) a vertex  $v$  of  $G$  placed at vertex  $(i, j)$  of the grid is mapped onto leaf  $\phi(v) = \text{shm}(i, j)$  of  $H$  and (2) an edge  $(a, b)$  is mapped onto the unique simple tree path from  $\phi(a)$  to  $\phi(b)$ .

In the resulting embedding, the load is one for the leaves and zero for the internal nodes. Furthermore, edges have dilation at most  $2 \log A$ . Finally, the congestion  $C_q$  of a tree edge joining a node  $w$  at level  $q+1$  to its parent  $u$  (at level  $q$ ) is at most  $3\sqrt{2}\sqrt{A/2^q}$  at even levels ( $q = 0(\text{root}), 2, \dots, \log A - 2$ ) and is at most  $4\sqrt{A/2^q}$ , at odd levels ( $q = 1, 3, \dots, \log A - 1$ ), since  $C_q$  cannot exceed the perimeter of the layout region corresponding to node  $w$ , as the layout of all the edges of  $G$  congesting tree edge  $(w, u)$  must cross the boundary of said region. A graph  $G$  is said to have *tree-area*  $\tilde{A}$  if  $\tilde{A}$  is the minimum value such that  $G$  admits a tree embedding with the properties just derived for graphs of area  $\tilde{A}$ . From a tree-embedding of area  $\tilde{A}$ , a grid layout of area  $A = O(\tilde{A} \log^2 \tilde{A})$  can be constructed in polynomial time [6]. The procedure is subtle and exploits some powerful combinatorial lemmas. A variant of the procedure also yields a more sophisticated embedding of  $G$  in  $H$  satisfying the following properties. The load of any node at level  $q$  is at most  $\lambda\sqrt{A/2^q}$ ; the congestion of any edge joining a node  $w$  at level  $q + 1$  to its parent is at most  $\gamma\sqrt{A/2^q}$ ; and the dilation of any edge is at most 4, where  $\lambda$  and  $\gamma$  are suitable constants. The developments of [6] revolve around the concept of minimum  $\sqrt{2}$ -*bifurcator* of a graph  $G$ , denoted by  $F$  and related to the tree area as  $F = \Theta(\sqrt{\tilde{A}})$ .

Most machines meant for general-purpose computing consist of a set of processing elements that can exchange messages via a routing network. In this context, it becomes interesting to compare machines with the same type of processing elements, but with different routers. Analogously to the question raised in the previous subsection for circuits, one can ask how well a given general-purpose router  $R(A)$  can deliver messages, compared to *any* router  $G$  of area  $A$ . To this end, it is useful to define the *load factor*  $\lambda$  of a given set of messages, whose sources and destinations (called *terminals*) are placed in a square region of area  $A$ , as the maximum over all the edges of the tree  $H$  (defined above) of the ratio between the number of messages with source and destination on opposite sides of an edge, and the congestion value  $C_q$  associated with that edge. It has to be observed that  $\lambda$  is a lower bound to the routing time of a message set for any network that can be laid out in a square region of area  $A$ . Therefore, the quality of a universal VLSI router is conveniently captured by its blowup and by its routing time, expressed as a function of  $\lambda$  and  $A$ .

## Universal Fat-Tree Architectures

Most efficient area-universal networks proposed in the literature exhibit a treelike structure, with the bandwidth of child-to-parent channels growing from the leaves toward the root. Informally, these networks are called *fat-trees*. Usually, the leaves of a fat-tree act as processing elements (in universal circuits) or as terminals (in universal routers) and the subnetworks located at the internal nodes perform a mere switching role. However, in a few notable cases, internal nodes also play an active role.

Typically, the bandwidth between a node and its parent doubles every other level, going from constant at the  $N$  leaves to  $\sqrt{N}$  at the root. All variants of fat-trees considered in the literature have layout area  $O(N \log^2 N)$ . In [7], Bilardi and Bay define the class of *channel-sufficient* fat-trees, as those where the maximum number of edge-disjoint paths connecting any two sets of leaves comes within a constant factor of the largest value compatible with the node-to-parent bandwidths. Graphs in this class, which contains all fat-trees proposed in the literature, are shown to require area  $\Omega(N \log^2 N)$ . When targeting universality for area  $A$ , if

a fat-tree with  $N = \Theta(A)$  leaves is chosen, then the area becomes  $\Theta(A \log^2 A)$ , resulting in a blowup  $\alpha = \Theta(\log^2 A)$ .

The first efficient universal fat-tree was proposed by Leiserson in [8]. Its internal nodes are realized as partial concentrators of constant depth, whence the name of *Concentrator Fat-Tree* (CFT). On a CFT, a message set of load factor  $\lambda \leq 1$  can be routed in time  $O(\log A)$ , hence, as an area-universal circuit, the CFT exhibits slowdown  $\sigma = O(\log A)$ . The authors of [8] also devise an off-line decomposition of an arbitrary message set of load factor  $\lambda$  into  $O(\lambda \log A)$  message sets of load factor smaller than 1. Thus, as an area-universal router, the CFT exhibits a routing time of  $O(\lambda \log^2 A)$ .

Routing time on the CFT has been subsequently improved by Greenberg and Leiserson in [9] and by Leighton et al. in [10], with randomized, online algorithms. Concentrators are powerful routing components, but setting their switches is achieved by computing bipartite graph matchings, for which area-time efficient, deterministic algorithms are not known. For this reason, two new fat-trees that do not make use of partial concentrators, namely, the *pruned butterfly* and the *sorting fat-tree*, have been introduced by Bay and Bilardi in [11] for online, deterministic routing. Finally, a variant of the CFT, called *Fat-Pyramid*, was proposed by Greenberg in [12] to achieve good blowup and slowdown in a VLSI model where the cost to transmit a bit over a wire is a (quite) general function of the wire length, ranging between constant and linear time.

In the context of universal circuits, the *Meshed CFT* has been considered, both by Leighton et al. in [10] and by Bay and Bilardi in [13]. The Meshed CFT consists of  $O(A/\log^2 A)$  meshes of size  $O(\log A \times \log A)$ , connected to the leaves of a CFT with root bandwidth  $O(\sqrt{A}/\log A)$ ; the total area is  $O(A)$ . In [10], the routing required to simulate a circuit makes use of standard permutation routing techniques on the mesh combined with a sophisticated randomized technique on the CFT. When evaluating the area requirements of the resulting circuit in the bit model, the mesh nodes require  $\Theta(\log \log A)$  bits apiece to encode switching decisions, pushing the total area to  $O(A \log \log A)$ . The  $O(\log \log A)$  blowup is avoided in [13] by a specialized routing approach that makes crucial use of the fact that the permutations to be routed in the meshes are not arbitrary, but arise from a layout. In both designs,

in spite of the reduced root bandwidth ( $O(\sqrt{A}/\log A)$ ) against  $O(\sqrt{A})$  of the original CFT), the routing time remains  $O(\log A)$ , thanks to a careful pipelining of  $O(\log A)$  message waves along the tree. In [13], it is also shown that the entire transformation of a given circuit layout into the meshed CFT configuration needed to simulate it can be computed in (optimal) sequential time  $O(A)$ . Whether a universal circuit of area  $O(A)$  can achieve sublogarithmic slowdown remains an open question. As a corollary of a more general result, Bilardi et al. in [14] have shown that any universal circuit which employs an embedding of the guest circuit for its simulation must incur a slowdown of  $\Omega(\sqrt{\log A}/\log \log A)$ . It is instead possible to achieve sublogarithmic and even constant slowdown with a substantial blowup, as established by Bhatt, Bilardi, and Pucci in [15]. Their construction makes crucial use of the constant-dilation tree-embedding discussed in Chapter 3 and of a redundant-computation technique. A slowdown of  $O(\log \log A)$  was also achieved by Kaklamani, Krizanc, and Rao in [16], with a multi-fat-tree network targeted to the simulation of planar circuits, and a with butterfly network, targeted to the simulation of circuits with a number of nodes suitably sublinear in the area. The butterfly approach suffers a larger blowup than the one achieved in [15] for the same slowdown, even for the restricted class of circuits that it can handle; however, the techniques of [16] are different from those employed in [15] and could have other significant applications.

A summary of the main characteristics of the universal designs discussed above is displayed in Table 1. For the sake of uniformity and comparison, in the table as well as in the preceding discussion, all results have been translated to a common framework. In particular, layouts are taken to be two dimensional, with area and time evaluated in Thompson's model. Also, the family of simulated routers (respectively, circuits) is assumed to be all those admitting layouts within area  $A$ . All routing times, except that of [10], apply to messages of  $b = O(\log A)$  bits, although they remain unchanged if  $b = O(1)$ . Features of a design that do not fit into the outlined common framework are explicitly reported and marked with symbol “ $\ominus$ ” if they represent a restriction, and with symbol “ $\oplus$ ” if they represent an enhancement.

Finally, it is interesting to observe that a universal router naturally yields a universal circuit, by first

**Universality in VLSI Computation.** **Table 1** Each table entry refers to a specific universal (router or circuit) design. The following features are displayed: (a) the topology; (b) the mode of operation (only for routers), that is, whether or not the universal router requires preprocessing to be configured for the simulation (off-line vs. online design), and whether or not the routing algorithm makes use of random bits (randomized vs. deterministic design); (c) the area blowup; (d) the routing time for universal routers (as a function of the load factor  $\lambda$  of the message set to be routed), or, alternatively, the slowdown of the simulation for universal circuits; and (e) the corresponding scientific reference

| Universal routers                                                                                        |         |                  |                                                           |      |
|----------------------------------------------------------------------------------------------------------|---------|------------------|-----------------------------------------------------------|------|
| Topology                                                                                                 | Mode    | Blowup           | Routing time                                              | Ref  |
| Concentrator Fat-Tree (CFT)                                                                              | off/det | $O(\log^2 A)$    | $O(\lambda \log^2 A)$                                     | [8]  |
| CFT                                                                                                      | on/rand | $O(\log^2 A)$    | $O(\lambda \log A + \log^2 A \log \log A)$                | [9]  |
| CFT ( $\ominus$ :word model)                                                                             | on/rand | $O(\log^2 A)$    | $O(\lambda + \log A)$                                     | [10] |
| Pruned Butterfly                                                                                         | on/det  | $O(\log^2 A)$    | $O(\lambda \log^2 A)$                                     | [11] |
| Sorting FT<br>( $\ominus$ : constant-degree message sets)                                                | on/det  | $O(\log^2 A)$    | $O(\lambda \log A + \log^2 A)$                            | [11] |
| Fat Pyramid<br>( $\ominus$ : $O(A/\log A)$ terminals)                                                    | off/det | $O(1)$           | $O(\lambda + \log A)$<br>( $\oplus$ :general delay model) | [12] |
| Universal circuits                                                                                       |         |                  |                                                           |      |
| Topology                                                                                                 |         | Blowup           | Slowdown                                                  | Ref  |
| CFT                                                                                                      |         | $O(\log^2 A)$    | $O(\log A)$                                               | [8]  |
| Mesched CFT                                                                                              |         | $O(\log \log A)$ | $O(\log A)$                                               | [10] |
| Mesched CFT                                                                                              |         | $O(1)$           | $O(\log A)$                                               | [13] |
| Multi-Computation FT (parametric design)<br>( $\oplus$ : simulates circuits with $\sqrt{A}$ -bifurcator) |         | $O(A^\epsilon)$  | $O(1/\epsilon)$                                           | [15] |
|                                                                                                          |         |                  | $4 \log \log A / \log A \leq \epsilon \leq 1$             |      |
| Multi-FT<br>( $\ominus$ : planar guest circuits only)                                                    |         | $O(\log^2 A)$    | $O(\log \log A)$                                          | [16] |
| Butterfly (parametric design)<br>( $\ominus$ : $O(\sqrt{A^{1+\epsilon}} \log A)$ nodes)                  |         | $O(A^\epsilon)$  | $O(1/\epsilon + \log \log A)$                             | [16] |
|                                                                                                          |         |                  | $0 < \epsilon < 1, \epsilon$ constant                     |      |

matching the terminals of the host router with the computing elements of the guest circuit according to the tree embedding induced by the guest's layout, and then hardwiring the routing decisions relative to the message set induced by the guest's proximity structure. The slowdown of the resulting universal circuit is obtained by setting  $\lambda = 1$  in the formula for the routing time. The blowup could increase, due to the area needed to store the hardwired routing decisions, although the increase is often negligible. The resulting performance metrics tend to be inferior to those of explicitly designed universal circuits. On the positive side, the class of simulated circuits typically includes all those of tree-area  $A$  (or, equivalently, with bifurcator  $F = O(\sqrt{A})$ ), which is a larger family than all circuits with a layout of area  $A$ . A

similar observation applies to the planar circuit simulator of [16], which is able to simulate all planar graphs with  $O(A)$  nodes, some of which are known to have area  $\omega(A)$ .

## Conclusions

Research on VLSI universality has demonstrated the viability of universal circuits and routers featuring area and time performances which are only a few logarithmic factors away from those exhibited by specialized devices of a stipulated area budget. This line of work has provided significant insight on several important technological issues. Regarding circuits, universal designs provide theoretical grounding to the compelling practical evidence that Field-Programmable

Gate Arrays (FPGAs) are indeed a viable and cost-effective option for the design of highly configurable, general-purpose integrated circuits and may offer considerable savings at the cost of a modest degradation in performance for a vast number of applications. On the other hand, the inevitable slowdowns incurred by such devices could be undesirable for specific, mission-critical computations, for which the development of specialized circuitry is most appropriate. The legacy of area-universal routing is instead related to the emergence of a number of successful commercial fat-treelike interconnection networks for parallel machines including the Thinking Machines CM-5, the Quadrics CS2, and the IBM RS/6000 SP.

## Related Entries

- [Models of Computation, Theoretical](#)
- [VLSI Computation](#)

## Bibliographic Notes and Further Reading

The following bibliography lists most relevant references on area-universality. For a more comprehensive introduction on research on the VLSI model of computation, the reader is referred to Jeffrey D. Ullman's book: *Computational Aspects of VLSI*, Computer Science Press, Rockville MD, 1984.

## Bibliography

1. Preparata FP. VLSI computation. In: Encyclopedia of parallel computing. Springer
2. Thompson CD (1980) A complexity theory for VLSI. Ph.D. thesis, Tech. Rep. CMUCS- 80-140, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh
3. Chazelle B, Monier L (1981) A model of computation for VLSI with related complexity results. In: Proceedings of the 13th ACM symposium on theory of computing, Milwaukee, pp 318–325
4. Bilardi G, Pracchi M, Preparata FP (1982) A critique of network speed in VLSI models of computation. IEEE J Solid-St Circ SC-17(4):696–702
5. Bilardi G, Preparata FP (1995) Horizons of parallel computation. J Parallel Distr Com 27(2):172–182
6. Bhatt SN, Leighton FT (1984) A framework for solving VLSI graph layout problems. J Comput Syst Sci 28:300–343
7. Bilardi G, Bay PE (1994) An area lower bound for a class of fat-trees. In: Proceedings of the second European symposium on algorithms, Utrecht, pp 413–423

8. Leiserson CE (1985) Fat-trees: universal networks for hardware-efficient supercomputing. IEEE T Comput C-34(10):892–900
9. Greenberg RI, Leiserson CE (1989) Randomized routing on fat-trees. In: Micali S (ed) Randomness and computation. JAI Press, Greenwich, pp 345–374
10. Leighton FT, Maggs BM, Ranade AG, Rao S (1994) Randomized routing and sorting on fixed-connection networks. J Algorithm 17(1):157–205
11. Bay PE, Bilardi G (1995) Deterministic on-line routing on area-universal networks. J ACM 42(3):614–640
12. Greenberg RI (1994) The fat-pyramid and universal parallel computation independent of wire delay. IEEE T Comput 43(12): 1358–1364
13. Bay PE, Bilardi G (1993) An area-universal VLSI circuit. In: Proceedings of the 1993 symposium on integrated systems, Seattle, pp 53–67
14. Bilardi G, Chaudhuri S, Dubhashi DP, Mehlhorn K (1994) A lower bound for area-universal graphs. Inform Process Lett 51(2): 101–105
15. Bhatt SN, Bilardi G, Pucci G (2008) Area-time tradeoffs for universal VLSI circuits. Theor Comput Sci 408(2–3):143–150
16. Kaklamani C, Krizanc D, Rao S (1993) Universal emulations with sublogarithmic slowdown. In: Proceedings of the 34th IEEE symposium on foundations of computer science, Palo Alto, pp 341–350

## UPC

WILLIAM CARLSON<sup>1</sup>, PHILLIP MERKEY<sup>2</sup>

<sup>1</sup>Institute for Defense Analyses, Bowie, MD, USA

<sup>2</sup>Michigan Technological University, Houghton, MI, USA

## Synonyms

- [Unified parallel C](#)

## Definition

UPC, Unified Parallel C, is a parallel programming language which is derived from the C language. It supports the Partitioned Global Address Space (PGAS) programming model.

## Discussion

### Introduction

UPC is a strict superset of the C programming language in the sense that any legal C program is a legal UPC program. It extends the C memory and execution model, and implements the Partitioned Global Address Space (PGAS) programming model. UPC adds several

new type-qualifiers to describe the sharing and consistency models of objects (e.g., variables, structures, and arrays). It also adds some new keywords to control the synchronization of the parallel threads (e.g., activities) within the UPC runtime environment. The driving goal behind UPC is to maintain the C aesthetic in a parallel programming language and model: users have a high degree of affinity with the machine they are programming; there are minimal runtime checks and restrictions; and programs are portable to a wide variety of parallel platforms.

As it is entirely possible to write in many programming styles within the C programming model and language, UPC makes no demands on programmers to conform to a particular parallel programming style. For example, UPC programs have been written in the styles of dynamic work queues; message queues; fine- and coarse-grained shared memory; fine- and coarse-grained distributed memory, lock- and barrier-based synchronized; synchronization-free “Monte Carlo”; and “embarrassing parallel.” UPC compilers strive to render each of these styles as efficiently as can be, but cannot prevent users from writing programs in a style that does not match a particular system’s performance profile. For example, it is possible to write a fine-grained, lock-based shared-memory program in UPC and run it on a system with no native support for such features. Performance of such attempts will be predictably horrid. Conversely, running such a program on a well-equipped system may be delightful.

UPC is implemented by a variety of compilers and runtime systems. These include both open-source-based compilers and runtimes as well as proprietary ones. Together these implementations allow users to run UPC on almost every significant parallel system in service today. It is important to note that this includes systems which have native support for shared memory and those whose physical interconnection is based on message passing paradigms. In addition, UPC support is planned for all future products of the main vendors of HPC systems. On all of these systems, UPC offers competitive performance to other parallel programming models, as well as the productivity offered by the C programming model and a shared-memory programming paradigm. Links to a variety of these implementations are provided in the Compiler section of the Bibliography.

## Parallelism Model

UPC uses the SPMD (Single Program Multiple Data) execution model and the PGAS (Partitioned Global Address Space) model of memory. In UPC, these models are so intertwined that it is difficult to discuss them independently. This article will first describe the execution model with the understanding that the threads operate within the context of the yet to be defined PGAS memory model.

Each instance of execution is called a *thread*. A UPC programmer typically thinks of a thread as a sequential program which shares some data with some number of other threads. The two keywords `MYTHREAD` and `THREADS` are expressions with a value of type `int` that provide a unique thread index for each thread and the total number of threads, respectively. UPC threads are statically declared, heavy-weight threads whose execution span the life of the whole program. There is no mechanism within UPC to spawn or kill threads. Where and how the threads are executed in hardware is determined by the system, not the language. UPC programs are usually executed via a system-dependent “parallel run” command, sometimes called `upcrun`. Arguments to such a command control the number of threads that will execute and can pass parameters to the runtime system to best suit a particular system.

Instruction-driven synchronization is provided by barriers and locks. The `upc_barrier` statement provides a “split-phase” barrier: `upc_notify` followed by `upc_wait`. These functions are *collective* functions in the sense that they must be called by all threads. The `upc_notify`/`upc_wait` pair implements a barrier because no thread can return from the call to `upc_wait` until all threads have called the `upc_notify`. The split-phase barrier requires that no globally sensitive code, including other collectives, be executed between the `upc_notify` and `upc_wait`. It provides a mechanism to reduce barrier overhead by allowing local or non-globally sensitive work to be placed between the two phases. An extreme instance of this is the bulk-synchronous model in which all “computation” is placed between the notify and wait while all data interaction (“communication”) is placed between the wait and the following notify.

Locks in UPC are provided by the functions `upc_lock`, `upc_lock_attempt`, and `upc_unlock` along with library functions to allocate locks.

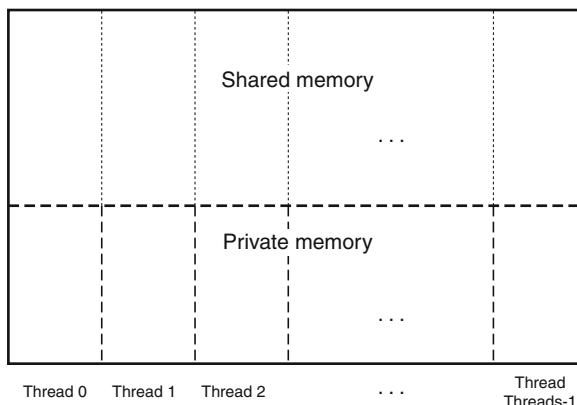
These operate on objects of type `upc_lock_t`, an opaque type for a shared object which takes on one of two states, *locked* or *unlocked*. Since locks can be allocated, a programmer can use them in a number of ways. For example, one could make an association between a lock and a section of code to provide instruction-driven synchronization like a critical section. If one allocated enough locks to associate individual locks with shared objects or subsets of object, one could use them for data-driven synchronization like atomic or transactional memory operations. Again, UPC defines the semantics, but makes no assumptions about the performance characteristics.

## Memory Model

UPC pairs the SPMD execution model with a compatible PGAS memory model. The notion of affinity in UPC captures the locality between processors and memory that exists on most parallel architectures.

[Figure 1](#) shows that memory in UPC is segmented into private and shared. For each thread, *private* memory has precisely the same semantics as in C programs. Private memory for one thread cannot be addressed by any other thread. This is the extreme case of memory locality. The ability to use private memory is important because it allows the explicit description of thread level concurrency and the associated scaling of processor to memory bandwidth; the importation of normal C functions on a thread by thread bases; and the explicit expression of the lack synchronization required among different threads acting on their own private objects.

*Shared* memory is declared via the keyword type-qualifier `shared` or is allocated by a library function



UPC. **Fig. 1** UPC Memory model

defined to create a shared object. UPC logically partitions shared memory into equivalence classes based on threads. This relationship is referred to as *affinity*. The affinity of shared objects is defined by cyclically assigning blocks of objects to threads. The default block size of one corresponds to the standard cyclic distribution. The other extreme is the standard block distribution that has only one block per thread.

The fact that shared objects are assigned affinity in a known pattern allows programmers and compilers to exploit the performance characteristics of private memory for some uses of the shared objects. The `upc_forall` loop construct does precisely this. The command has the form:

```
upc_forall(init;condition;
 increment;affinity)
```

This is the familiar `for` loop except that the bodies of the loop are only executed by the threads that match the affinity field. If the affinity field is an integer, the thread which is congruent to that integer, modulo `THREADS`, is the thread that executes the loop body. If the affinity field is a shared address, then the thread with affinity to the object at that address is the thread that executes the loop. It is important to note that this construct is not a “fork-join” based parallel loop, nor does it impose synchronization before, during, or after the loop. It does not express parallelism, it expresses locality. In addition to being a useful idiom for programmers that can be optimized by compilers, it is a suggestive abstraction that ties the execution model to a PGAS memory model for a two-level memory hierarchy. Generalizations of `upc_forall` may be useful for deeper memory hierarchies.

The memory consistency model for UPC also facilitates performance by eliminating unrequested synchronization. Every memory reference for a shared object is either *strict* or *relaxed*. One can set the default behavior for all objects in a program by including either the header file `<upc_strict.h>` or `<upc_relaxed.h>` at the beginning of the program. One can override the default behavior for a particular object with use of the reference type-qualifier `strict` or `relaxed`. Finally, one can control the behavior a particular use of an object with a compiler pragma.

*Strict* references are sequentially consistent and *relaxed* references provide a weak consistency. The key

is the interaction of the two. The strict references lay down a sequentially consistent grid of references that provides fenced regions in which the ordering of relaxed references can be somewhat arbitrary. That is, strict accesses always appear (to all threads) to have executed in program order with respect to other strict accesses, and in a given execution all threads observe the effects of strict accesses in a manner consistent with a single, global total order over the strict operations. Any sequence of purely relaxed shared accesses issued by a given thread may appear to be reordered relative to program order, and different threads need not agree upon the order in which such accesses appeared to have taken place. The only exception to the previous statement is that two relaxed accesses issued by a given thread to the same memory location where at least one is a write will always appear to all threads to have executed in program order. When a thread's program order dictates a set of relaxed operations followed by a strict operation, all threads will observe the effects of the prior relaxed operations made by the issuing thread (in some order) before observing the strict operation. Similarly, when a thread's program order dictates a strict access followed by a set of relaxed accesses, the strict access will be observed by all threads before any of the subsequent relaxed accesses by the issuing thread. Consequently, code blocks containing only relax operations are open to all serial compiler optimization techniques and strict operations can be used to synchronize the execution of different threads by preventing the apparent reordering of relaxed operations contained within the grid of strict operation.

### An Example Program

The following UPC code implements a fairly standard algorithm to illustrate a Monte Carlo technique for calculating the mathematical constant Pi. It does so by determining what percentage of random values in a unit rectangle fall within a unit quarter-circle and multiplying by four. It is a parallel algorithm in that any number of parallel actors can make contributions to the accumulations of “tries” and “hits.”

```
#include <upc.h>
static shared int tries;
static shared int hits;
double pi(int count) {
 int i;
 for (i=0; i<count; i++) {
```

```
 tries++;
 double x = drand48();
 double y = drand48();
 if (x*x+y*y <= 1.0)
 hits++;
 }
 double a = (double) hits/ tries;
 return (4.0*a);
}
```

The only changes from a standard C program required to make this a parallel program are the addition of the `shared` keyword to the two accumulators. In this very simple version of the code, there is no synchronization required. Any thread calling the function will use the values of `tries` and `hits` that all threads have completed by the point it completes its loop. Of course this will result in threads returning different values of `Pi`, but it is important to note that the algorithm itself is nondeterministic, so the writer of this code decided that added synchronization was unnecessary. Note that this function could be called any number of times by any group of threads, each time a “better” answer would be returned (assuming of course that the implementation of `drand48()` is a good random number generator!). If the writer wanted to ensure that all threads returned the same value, a `upc_barrier` statement could be added before the `return` statement, but this would then require that all threads call `pi()` collectively.

This simple version may not scale very well to large numbers of threads on some systems as they will “fight” over the shared variables and each access to these is a remote access, which are generally more expensive than local accesses, sometimes dramatically so on large systems. A second version, given in the following code, would be “faster” on many systems.

```
#include <upc.h>
static shared int tries [THREADS];
static shared int hits [THREADS];
double pi(int count) {
 int i;
 for (i=0; i<count; i++) {
 tries [MYTHREAD]++;
 double x = drand48();
 double y = drand48();
 if (x*x+y*y <= 1.0)
 hits [MYTHREAD]++;
 }
 int t_tries=0, t_hits=0;
 for (i=0; i<THREADS; i++) {
```

```

 t_tries += tries[i];
 t_hits += hits[i];
}
double a = (double) t_hits/ t_tries;
return (4.0*a);
}

```

Here the program will likely run faster and scale better as any reasonable implementation of UPC will know that (due to affinity) all accesses in the first loop are effectively local. It maintains the characteristic of a loosely synchronized program. If the writer wished a more strictly synchronized program, than version two, a barrier could be inserted before the second loop. In fact, the writer could use the collective function `upc_all_reduce()` to avoid writing that second loop. Using the provided collective function could also increase performance, as it may be implemented with a more clever algorithm than the simple loop above. But the writer of this code did not want to “over-synchronize.”

A savvy reader of this article may notice that the second program also eliminates race conditions that the first code contained relative to the parallel increments of the shared variables. It is a philosophical question whether race conditions are “good” or “bad.” Understanding them, using them or avoiding them is a programming choice. It could be argued that for this algorithm, one would obtain better results by not preventing the race. Following the general C philosophy, UPC takes no position on whether race conditions are “good” or “bad,” but it does provide mechanisms to manage them. If the programmer wanted to avoid the nondeterministic effects of the race condition, she could surround the shared accesses with a pair of `upc_lock()`/`upc_unlock()` routines. This would likely result in an even less scalable program. If available, she could also use atomic operations. Atomic increment operations are available on some machines and are, in fact, faster than usual read from memory, update, and write back cycle. Hence, the compiler or the runtime system may choose to use the atomic update for the sake of performance and incidentally avoid the race condition.

## Applications

Like C, UPC is a high-level language that is designed to give programmers control of the machine without imposing one particular paradigm. The following sketches what a program might look like if UPC were used to implement a few of the most popular parallelization strategies.

Consider the master/slave model of parallelism. If one uses a `upc_lock` to protect a shared work queue, all the threads could execute as slave tasks. Each thread in turn: obtains the lock, takes the next assignment and updates the work queue, then releases the lock and executes the slave task. The process repeats until the work queue is empty. There are minor details in starting the process and terminating the process, but the shared work queue allows one to use a master/slave model without the need of a master thread.

Again, the programmer can chose the appropriate amount of synchronization. If the size of the task is large compared to modifying the work queue, then conflicts at the critical section are probably low and the lock overhead is probably tolerable. If the application has a large Amdahl fraction because of the conflicts at the critical section, one could remove that overhead at the expense of the redundant work caused by not using a lock.

A number of applications are commonly identified as applications that use blocking point-to-point communication to exchange objects and synchronize threads. These include applications like Fast Fourier Transforms, most codes that use domain decomposition, non-continent sorts, and dense linear algebra. These have well-known distributed-memory implementations. Since UPC has a well-defined data layout scheme, forming partnerships between threads (the equivalent of picking send/receive pairs) and the use of explicit messages can be avoided by using standard (albeit sometimes messy) pointer arithmetic and simple assignment statements on shared variables. The second version of the Pi program above is a trivial example of a UPC version of such a program. The first loop use of affinity on a shared array is effectively using private variables on each thread. The difference between message passing and shared memory is illustrated in the second loop. UPC programs can “receive” the values “on” another thread with a simple read statement without

interrupting its work flow. This one-sided communication does not require synchronization, so synchronization is a choice.

Often these applications work “in phases” or with an outer loop. The outer loop commonly executes a compute phase and then a communication phase. The synchronization at this level would be instruction driven. Depending on the application this can be an exact fit for the bulk-synchronous programming model or in other cases the communication can essentially disappear. If threads only write to objects for which they have affinity, there can be no race conditions and computation phase can be asynchronous. If the communication phase must wait until the computation is completed, a split-phase barrier as described above would yield a bulk-synchronous implementation. In addition, if the algorithm enjoys sufficient reuse of off-affinity references one could use `upc_memcpy` to make private copies of shared objects to improve performance in the next computation phase. In many cases the communication phase is unnecessary as the communication will be accomplished by interweaving off-affinity references with the computation. The relaxed consistency model then allows one to overlap (via prefetching or caching) these off-affinity references with the ongoing computation. On some systems this can essentially hide the communication costs.

UPC does not impose a synchronization model, a preferred granularity, or parallelization technique. Therefore, the UPC programmer is free to match applications, algorithm choices, and implementation to a given architecture in order to maximize performance.

## UPC History

UPC is the direct descendant of three C language extensions developed in the 1990s: Split-C [4], PCP [1], and AC [2]. Each of these languages had implementations on several platforms of the day and was the result of several years of experimentation. In 1996, the principal contributors to these languages met at UC-Berkeley and designed UPC. Its initial implementation was completed in 1997 for the Cray T3-E system by Carlson and Draper. The first published specification was [3]. The UPC consortium was formed in May 2000 and is open to all interested parties. The consortium controls the

specification which stands at version 1.2, published in May 2005.

## Related Entries

- ▶ [Coarray Fortran](#)
- ▶ [PGAS \(Partitioned Global Address Space\) Languages](#)

## Biographical Notes and Further Reading

### Textbook

Tarek El-Ghazawi, William Carlson, Thomas Sterling, Katherine Yelick, UPC: Distributed Shared-Memory Programming, John Wiley & Sons, Hoboken, New Jersey, 2005.

### Web Documentation

UPC Documentation,  
<http://upc.gwu.edu/documentation.html>  
 UPC Wiki, <https://upc-wiki.lbl.gov/index.php>

### Compilers

IBM XL UPC Compilers,  
<http://www.alphaworks.ibm.com/tech/upccompiler>  
 HP UPC, <http://h30097.www3.hp.com/upc/>  
 Intrepid Technology, Inc, <http://www.intrepid.com/>  
 GCC UPC, <http://www.gccupc.org/>

### Research Efforts

UPC@Berkeley, <http://upc.lbl.gov>  
 UPC@Florida, <http://www.hcs.ufl.edu/upc>  
 UPC@GWU, <http://upc.gwu.edu>  
 UPC@MTU, <http://upc.mtu.edu>

## Bibliography

1. Brooks E, Warren K (1995) Development and evaluation of an efficient parallel programming methodology, spanning uniprocessor, symmetric shared-memory multi-processor, and distributed-memory massively parallel architectures, poster session at Supercomputing '95, San Diego, CA, 3–8 December 1995
2. Carlson WW, Draper JM (1995) Distributed data access in AC. Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), Santa Barbara, CA, 19–21 July 1995, pp 39–47
3. Carlson WW, Draper JM, Culler DE, Yelick K, Brooks E, Warren K (1999) Introduction to UPC and language specification, CCS-TR-99-157. IDA/CCS, Bowie

4. Culler DE, Dusseau A, Goldstein SC, Krishnamurthy A, Lumetta S, von Eicken T, Yellick K (1993) Parallel programming in split-C. In: Proceedings of Supercomputing '93, Portland, OR, 15–19 November 1993, pp 262–273

---

## Use-Def Chains

► [Dependence Abstractions](#)

# V

## Vampir

HOLGER BRUNST, ANDREAS KNÜPFER  
Technische Universität Dresden, Dresden, Germany

### Synonyms

Vampir 7; VampirServer; VampirTrace; Vampir NG; VNG

### Definition

In the scope of HPC, the name Vampir denotes a software framework that addresses the performance monitoring, visualization, and analysis of concurrent software programs by means of a technique referred to as event tracing. Originally, VAMPIR was an acronym for “Visualization and Analysis of MPI Resources.” Today, the framework no longer addresses MPI programs only. Support for many different programming paradigms and performance data sources has been included over time.

The framework consists of a monitor and a visualization component, which are sometimes independently referred to as VampirTrace and Vampir. The monitor component attaches to a running software program and records timed status information about the invocation of program subroutines, the usage of communication interfaces, and the utilization of system resources.

### Discussion

#### Monitoring

The monitoring component (VampirTrace) is used for recording the event traces during a program run. This process involves two steps, the instrumentation step and the runtime recording step [2, 3].

The former modifies the target executable by inserting measurement points that later on allow the detection of runtime events. The latter step handles the runtime events as they occur. This includes collecting

events and attributes of events as well as buffering and storing the event data stream.

#### Supported Paradigms and Languages

VampirTrace currently supports event tracing for the programming languages C, C++, Fortran, and Java. It works for sequential programs as well as for parallel programs using MPI, OpenMP, POSIX Threads, GPUs, the Cell Broadband Engine, or hybrid combinations of them.

#### Instrumentation

Instrumentation is the modification of the target program in order to detect predefined runtime events. It is inserting monitoring probes on different levels and in different ways:

**Source code:** On the source code level either manually or automatically with the help of source-to-source translation tools. This may take place during the software development as permanent part of the code or as a preprocessing step during compilation.

**Compiler:** By compilers with specific command line switches and interfaces. This is supported by most relevant compilers today, including the GNU compiler collection, the OpenUH compilers, and the commercial compilers from Intel, Pathscale, PGI, SUN, IBM, and NEC.

**Re-linking:** Re-linking with a wrapper library that includes instrumentation. Typically, the wrapper library will refer to the original library for the provided functionality.

**Binary:** By binary rewriting of a ready executable either in a file or a memory image.

In VampirTrace, different instrumentation techniques can be combined for different types of events. For convenience, the complex internal instrumentation details are hidden in compiler wrappers. The compiler wrappers can be used like regular compiler commands. They refer to underlying compilers but perform all

instrumentation activities and/or add instrumentation command line options.

## Runtime Recording

The runtime recording library receives events from the instrumentation layer. It is responsible for collecting the events together with related attributes. Then it stores them as event records in a memory buffer which is written to a file eventually.

Below, the most important types of events are listed. They are recorded with a high-resolution time stamp and a location identifier specifying the process or thread. Furthermore, there are additional type-specific attributes.

**Enter/leave:** Entering to or returning from a subroutine call. This is used for instrumented subroutines in the user code as well as for MPI calls, POSIX I/O calls, or LIBC calls.

**Send/receive:** Sending or receiving of point-to-point messages both for the blocking and the non-blocking versions. This is modeled according to the MPI standard but can be used for alternative message-passing models.

**Begin/end collective communication:** Begin and end of an MPI collective communication operation. They are present at every rank participating in an MPI collective operation.

**Begin/end of I/O operation:** Begin and end of an I/O operation.

**Counter sample:** Provides the current value of a performance counter.

**Timers and Timer Synchronization** Precise timing plays an important role for event tracing. Therefore, VampirTrace supports a wide range of generic and platform specific high-precision timers that have a resolution of up to a single CPU clock tick.

VampirTrace provides its own timer synchronization mechanism because most high precision timers in parallel environments are asynchronous. During trace collection, only local timer values are recorded. In addition, synchronization information is collected, which allow the conversion from local time stamps to globally synchronized time stamps in a post-processing step.

**Performance Counters** VampirTrace supports a number of sources for performance counters. Firstly, it can read

hardware performance counters via the PAPI library, SUN Solaris CPC counters, and NEC SX counters. Secondly, it allows to record memory allocation statistics, I/O throughput statistics, and arbitrary counter values provided by the program.

**Profiling** As an alternative mode, VampirTrace supports profiling, which delivers a concise statistic summary instead of a detailed trace file.

## Open Trace Format

The standard trace file format of VampirTrace is the Open Trace Format (OTF). It is developed and maintained by ZIH, Technische Universität Dresden in cooperation with the University of Oregon and the Lawrence Livermore National Lab.

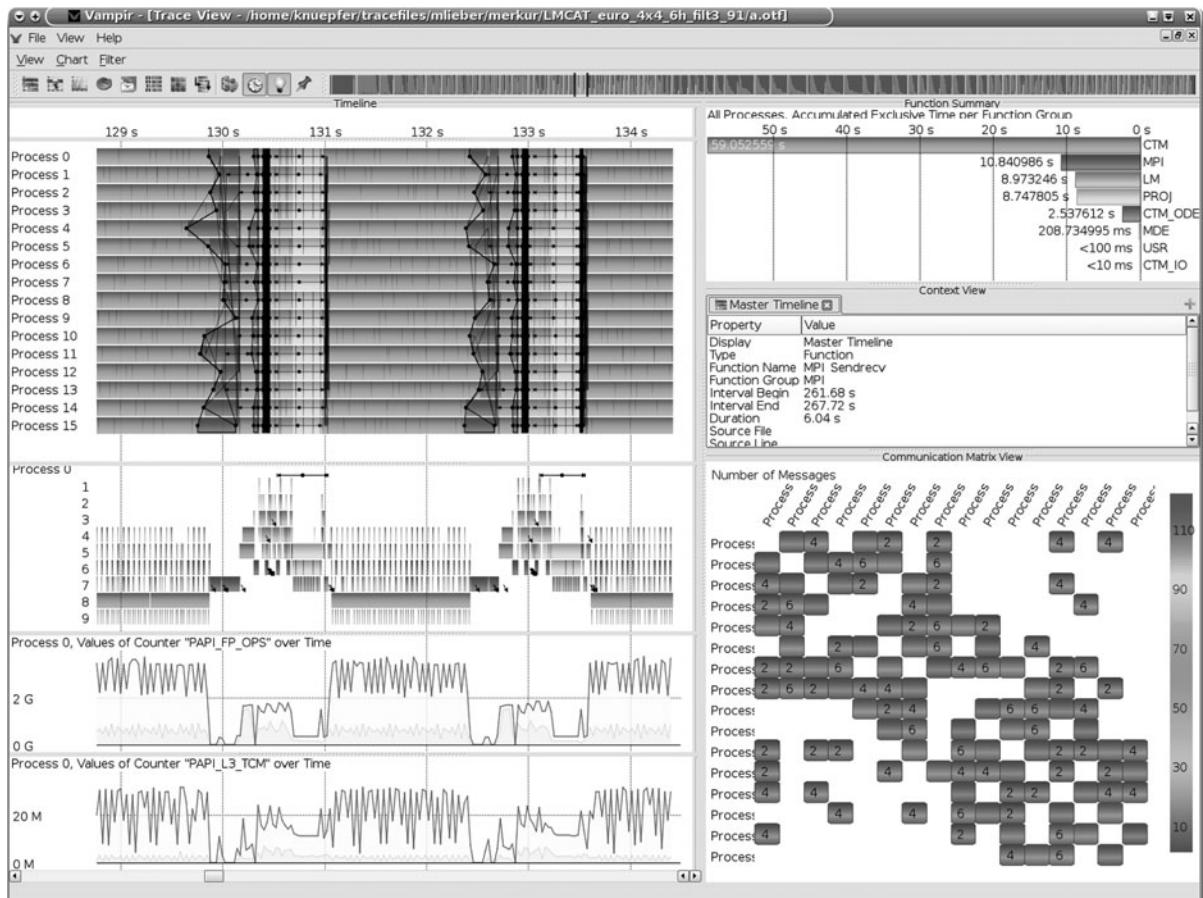
## Visualization

The visualization component of Vampir displays the runtime behavior of parallel programs. It visualizes event traces gathered by the monitoring component VampirTrace. The Vampir tool translates trace file data into a variety of graphical representations that give developers detailed insights into performance issues of their parallel program. Vampir allows to quickly browse large performance data sets by means of interactive zooming and scrolling. The detection and explanation of unexpected or incorrect performance behavior is straightforward due to the exact rendering of the program flow. Vampir supports two general display types for the illustration of trace data: timeline and chart views [1, 4].

## Timelines

A timeline view shows detailed event-based information for arbitrary time intervals. It graphically presents the chain of events of monitored processes on a horizontal time axis. Detailed information about subroutine invocation, communication, synchronization, and hardware counter events is provided. Vampir currently features three different timeline types: master timeline, process timeline, and counter-data timeline. Figure 1 shows the three different timeline types on the left-hand side from top to bottom.

**Master Timeline** The master timeline consists of a collection of rows which represent individual processes.



**Vampir. Fig. 1** Vampir GUI with the most important displays: the small *Navigation Timeline* (top), the *Master Timeline* (upper left), the *Process Timeline* (middle left), two *Counter Timelines* (bottom left), the *Function Summary* (upper right), the *Context View* below (middle left), and the *Communication Matrix View* (bottom right)

All timelines are equipped with a horizontal timescale. Likewise, timelines provide the names of the depicted processes on the left. Color-coding is used to identify specific program actions. In this example, “dark” sections identify MPI communication and synchronization. This color-coding is customizable and depends on the user and the contents of the recorded trace file.

**Process Timeline** The process timeline resembles the master timeline but has a slight difference. In this case, the different vertical levels represent the different stack levels of subroutine calls. The main routine begins at the topmost level, a respective subroutine call is depicted a level below, and so forth. If a subroutine is completed, the graphical representation of the calling routine continues one level above.

**Counter Data Timeline** The counter timeline shows recorded performance counter values of the processes with a scale on the left-hand side. The display gives the minimum, average, and maximum values if the individual fluctuations are too tiny to distinguish.

**Zooming** All timeline displays allow zooming in time to look at details within the huge amounts of data. The zoom intervals of all timelines are aligned such that zooming in one of the displays will update all others to the same time interval.

## Charts

In addition to the timeline displays, there are chart displays that provide summarized performance metrics computed from the corresponding event data. Again,

the chart displays are connected to the zoom interval of the timeline views. The summary information always relates to the current zoom interval, i.e., the statistics are constraint to the current contents of the timeline views.

**Function Summary** The function summary gives a profile (statistics) about subroutine calls, either as number of calls or inclusive/exclusive timing for individual subroutines or for classes of subroutines. It provides different graphical representations.

**Communication Matrix View** The communication matrix provides statistics about point-to-point messages between sender and receiver processes in a two-dimensional matrix with a color legend. It can report message counts, data volumes, timing, and speed. This display can also be zoomed to a subset of senders/receivers or coarsened to groups of processes.

**Call Tree** The call tree display presents caller and callee relations between subroutines in the current zoom interval.

**Context View** Finally, the context view shows individual detail information for a selected event (by mouse click), e.g., subroutine calls or messages.

## Editions

The Vampir performance data browser is available for all major platforms, including Unix, Microsoft Windows, and Apple Mac OS X-based systems. Three different product editions are available. “Vampir-Light,” “Vampir-Standard,” and “Vampir-Professional” address the needs of small, midsize, and large institutions. These three editions mainly differ in terms of the supported platform size and pricing. In addition to the above editions, a feature and time-restricted copy of Vampir is available for students and evaluation purposes.

## History

In 1992, the development of the Vampir performance visualization tool was started by Wolfgang E. Nagel and Alfred Arnold at the Center for Applied Mathematics (ZAM) of Research Center Jülich in Germany. In these early times, it was still called *PARvis*. When the software became a commercial product in 1996, its name

was changed to the acronym VAMPIR (visualization and analysis of MPI resources) in order to underline its focus at that time, which was the visualization of MPI parallel programs. The German company *Pallas GmbH* became the commercial distributor for the software in 1996. In 1998, the design responsibilities for the Vampir software tool moved from Research Center Jülich to the Center for High Performance Computing of Technische Universität Dresden due to the relocation of its founder Wolfgang E. Nagel. Between the years 2004 and 2005, the software was temporarily distributed by the Intel Corporation under the label *Intel Trace Analyzer*. Since 2005, the original software is available again under its most prominent name *Vampir* and can be obtained from Technische Universität Dresden and GWT-TUD GmbH which are colocated in Dresden.

## Future Directions

Motivated by the changing requirements and needs of parallel software developers, the Vampir tool suite is under constant development. The tapping and processing of new performance data sources, namely, hardware accelerators and energy meters is an active field of research. A customizable event data mining engine and the comparison of successive trace runs are under investigation. Furthermore, the overall scalability of the tool suite will be improved in the near future by means of compression techniques that are based on pattern detection algorithms, which allow the elimination of redundant information [5].

## Credits

We would like to acknowledge Vampir’s original inventors, Prof. Wolfgang E. Nagel and Alfred Arnold. Furthermore, we would like to acknowledge the financial support from the Bundesministerium für Bildung und Forschung, the European Union, and other government sponsors. Finally, we would like to acknowledge Forschungszentrum Jülich, Indiana University, and Oak Ridge National Laboratory, in particular Bernd Mohr, Craig Stewart and Rainer Keller.

## Licenses and Other Software

VampirTrace and OTF come under a BSD Open Source license. They are also included as default components in the Open MPI distribution from version 1.3, in the Sun

Cluster Tools from version 8.2, and in the Open Speed Shop package from version 1.9.

## Related Entries

- [Performance Analysis Tools](#)

## Bibliographic Notes and Further Reading

1. For more information about product releases of Vampir, please visit <http://www.vampir.eu>
2. For more information about VampirTrace, please visit <http://www.tu-dresden.de/zih/vampirtrace>
3. For more information about the Open Trace Format (OTF), please visit <http://www.tu-dresden.de/zih/otf>
4. For more information about Vampir related research, please visit <http://www.tu-dresden.de/zih/ptools>

## Bibliography

1. Nagel WE, Arnold A, Weber M, Hoppe H-Chr, Solchenbach K (1996) VAMPIR: visualization and analysis of MPI resources. *Supercomput J* 1(12):69–80. SARA, Amsterdam
2. Müller M, Knüpfer A, Jurenz M, Lieber M, Brunst H, Mix H, Nagel WE (2007) Developing scalable applications with Vampir, VampirServer and VampirTrace. In: Parallel computing: architectures, algorithms and applications, *Adv Parallel Comput* 15:637–644. IOS Press, Amsterdam. ISBN 978-1-58603-796-3
3. Knüpfer A, Brunst H, Doleschal J, Jurenz M, Mickler LH, Müller M, Nagel WE (2008) The Vampir performance analysis tool-set. In: Tools for high performance computing. Springer, Berlin, pp 139–155. ISBN 978-3-540-68561-6
4. Brunst H (2008) Integrative concepts for scalable distributed performance analysis and visualization of parallel programs. Dissertation, Technische Universität Dresden. ISBN 978-3-8322-6990-6
5. Knüpfer A (2009) Advanced memory data structures for scalable event trace analysis. Dissertation, Technische Universität Dresden, Suedwestdeutscher Verlag fuer Hochschulschriften. ISBN 978-3-838-10943-5

## Vampir 7

- [Vampir](#)

## Vampir NG

- [Vampir](#)

## VampirServer

- [Vampir](#)

## VampirTrace

- [Vampir](#)

## Vector Extensions, Instruction-Set Architecture (ISA)

VALENTINA SALAPURA

IBM Research, Yorktown Heights, NY, USA

## Synonyms

[Data-parallel execution extensions](#); [Media extensions](#); [Multimedia extensions](#); [SIMD \(Single Instruction, Multiple Data\) Machines](#); [SIMD extensions](#); [SIMD ISA](#)

## Definition

Instruction-Set Architecture (ISA) vector extensions extend instruction-set architectures with instructions which operate on multiple data packed into vectors in parallel.

## Discussion

Instruction-set architecture vector extensions or vector instructions are SIMD instructions. SIMD stands for “Single Instruction stream, Multiple Data stream,” and it is a technique used to exploit data level parallelism.

SIMD architectures apply operations to a fixed number of bits in a single logical computation. These bits are generally spatially local (i.e., a set of successive bits) and can represent a variable number of distinct data elements. The number of elements in a SIMD vector is a direct relationship of the total SIMD register size and the number of bits per data element – one can generally fit 16 8-bit (byte) elements in a 128-bit SIMD register, providing for a 16-way parallel operation on bytes, but only 4 32-bit data elements can fit in the same register, providing a maximal 4-way parallel operation on 32-bit (word) elements.

The maximal speedup available by applying a SIMD architecture to a problem is bounded by both the maximal number of elements in the SIMD vector register and the computational efficiency of the SIMD algorithm. Computational efficiency is a measure of the effective utilization of the elements available within the SIMD register. If a SIMD register can contain 4 elements, but only 2 of the elements contain data that are used in the computational algorithm, then the computational efficiency of that SIMD algorithm is 50% (i.e., 2 out of 4). Combining the maximum parallelism with the computational efficiency gives a rough measure of the overall speedup.

## History

There are two distinct types of architectures captured by the classification “vector architecture” which is used today to describe instructions which operate on multiple data values simultaneously. The term was initially coined when traditional vector processors were in common use. Vector-processing facilities operated on multiple data items stored in a vector register containing a large number of vector elements ordered sequentially in the vector register. A single instruction (e.g., a vector add) would indicate that the successive elements of each source operand register would be added together and placed into a third vector register. This process necessarily specified a set of multiple data items (in this case tuples) on which the same instruction operation (i.e., add) was to be performed. An example of such vector processors is the Cray X-MP from the early 1980s.

Modern vector architectures also operate on vectors of data, but do so in a more time-parallel fashion. Vector length is typically held to be quite short, usually dependent on the size of the vector elements, and generally not exceeding 16 elements. These SIMD architectures generally fix the total capacity of a vector register (in terms of bits) to a well-defined constant value (typically 128 bits) and allow a varying number of elements per vector based on the bit-size representation of the element data type (e.g., 16 8-bit byte elements, or 4 32-bit word elements). This architecture facility is referred to as “SIMD architecture,” and qualify with “short/parallel vector SIMD” when necessary.

The first architecture to include short parallel vector SIMD instructions was the Intel i860 [1]. These instructions were targeted at 3D graphics acceleration. Vectors

had a data width of 64 bits and were stored in the floating-point registers. Processing was performed on vectors of eight 8-bit elements, four 16-bit elements, or two 24-bit or 32-bit elements, by a dedicated graphics processing unit.

Subsequently, several other architectures were extended with a SIMD facility to support graphics processing: as an example, the HP Precision Architecture’s MAX facility was targeted at speeding up MPEG decompression, achieving the first real-time software-based decoding of MPEG video streams. Unlike the separate i860 facility, the HP MAX facility shared the data path of the main integer pipeline and stored four 8-bit elements of data in the integer (general purpose) register file. Based on this choice, key operations such as vector byte add could be achieved simply by breaking the carry chain between the bytes of the normal 32-bit integer data path, resulting in only minor incremental cost to support this “subword” SIMD facility [2].

Architects chose a different route for the IBM Power Architecture Vector Multimedia Extension facility. The Power Architecture vector facility (also known as AltiVec) added a dedicated SIMD vector register file with 32 128-bit registers, and a rich set of operation primitives and data types ranging from 4-bit pixels, 8-bit, 16-bit, 32-bit integers, to 32-bit (IEEE 754 single-precision) floating point. Execution commonly occurs in four execution pipelines, one dedicated to simple integer operations (add/subtract/logical), one for complex integer operations (multiply, multiply-add), one for floating-point operation, and one for permute (data formatting) operations [3].

Starting with Power 7, a new vector-scalar extension (VSX) integrates the floating point and VMX register files to offer a single 64-entry register file for scalar and vector computation while retaining binary compatibility with legacy applications.

Several application-domain specific SIMD facilities are provided for the Power Architecture, including GameCube’s processing of two single-precision floating-point data elements in a single 64-bit double-precision floating-point register, Motorola’s SPE, and BlueGene’s “Double-Hummer” 2-wide double-precision floating-point elements in paired floating-point registers.

The Intel x86/AMD64 architecture has several distinct SIMD facilities. The MMX facility supports integer

data formats in eight 64-bit registers, which “overlay” the traditional FPU register stack. The 3DNow! extensions add single precision floating-point operations. A third SIMD facility (SSE) includes an eight-entry 128-bit register file and also provides scalar floating-point operations to serve as both a new floating-point facility and a SIMD architecture.

The Cell Synergistic Processor Architecture is a new architecture built from the ground up around short parallel vector SIMD processing. A single 128-entry 128-bit register file stores both scalar and vector data, and SIMD data paths provide the computation capabilities for both scalar and SIMD processing. The SPU supports a repertoire of 8-bit, 16-bit, and 32-bit integer operations, and 32-bit and 64-bit floating-point operations [4].

## Area Efficiency

As processor designs have become increasingly complex, ISA vector extensions have become more attractive by delivering increased performance for a range of workloads. Unlike traditional ILP extraction in microprocessors which require to fetch, decode, issue, and complete more instructions to increase compute performance, vector extensions increase the compute capability *per instruction*. Thus, while ILP performance techniques require scaling up of all parts of the processor, such as instruction fetch bandwidth, instruction issue bandwidth, global completion tables or reorder buffers, the amount of register file and data cache ports and, hence, register file size, as well as execution units, vector instructions can increase performance by replicating execution units operating in lockstep and ensure adequate data delivery by widening register files and implementing wide data cache and register file ports. In contrast, instruction processing capabilities, such as the number of instructions fetched, decoded, renamed, issued, and completed remains independent of the number of operations completed by a single instruction.

In addition to the attractive area efficiency, vector extensions are comparatively easier to verify than many other performance techniques, as such designs increase the dataflow content of a design by replicating execution units. The dataflow of these execution units can in turn be verified in a modular fashion at the individual dataflow level.

## Power and Energy Efficiency

Vector extensions have also become increasingly attractive in the power-constrained design space in which modern microprocessor designs must be implemented. From a power- and energy-efficiency perspective, reduced area consumption directly translates into a reduction in power and energy use.

Beyond the power and energy benefits associated with area reduction, the performance increase offered by exploiting data parallelism with vector extension can also be translated into using more efficient operating points.

By exploiting the increased performance potential offered by vector extensions to operate microprocessors in a more power-efficient regime, power and energy efficiency can be dramatically improved. Because performance  $perf = ops/cycle * f$ , any increase in ops/cycle can be offset by a commensurate reduction in operating frequency. Following the power equation

$$P = CV^2f$$

(here  $P$  is power,  $C$  capacitance,  $V$  voltage, and  $f$  operating frequency), and the observation

$$f \sim V$$

any increase in processing performance can be used as a cubic reduction in power due to the  $V^2 f$  factor. On the other hand, the capacitance  $C$  will only rise modestly as the number of execution units is increased.

Salapura et al. study energy efficiency of the BlueGene system and contains a discussion of power efficiency of SIMD architectures based on power/performance characterization of BlueGene workloads [5].

## Software Enablement

Programmers can exploit vector capabilities provided by vector ISA extensions in several different ways.

### Compiler Supported Exploitation

Compiler support is the single most important enabler, and yet also the biggest obstacle, to ensure the reach of vector architectures beyond the traditional graphics and high-performance compute kernels.

In this approach, application is written without any vector architecture in mind. The compiler detects

parts of the code with data parallelism and generates the code which takes advantage of SIMD capabilities for the target architecture. As advantage, data parallelism in applications can be exploited to achieve higher performance, the application is portable, and the usage of vector unit and vector instructions is transparent to the programmer. This approach requires compiler support, developed libraries, and middleware for the target vector architecture to exploit the vector capabilities.

### Usage of Intrinsics

To overcome the limitations of compilers commonly in use today, SIMD instructions are often generated directly using intrinsics, a form of inline assembly. This approach requires the programmer to map the algorithm to the sequence of assembly instructions, but it allows the compiler to perform register allocation and instruction scheduling. When using compiler intrinsics, the programmer gives directions to the compiler on which variables to perform the specified vector operation for the target vector architecture.

As a result, the code is optimized for the particular vector architecture, and it exploits vector capabilities efficiently and achieves high performance. As a disadvantage, the application developer has to be familiar with the vector instructions of the target architecture and to be able to use them efficiently. The resulting code is also typically not portable to any other vector architecture.

To port the code to some other vector architecture, compiler intrinsic for the vector instructions specified in the original code have to be replaced to match the new target vector architecture. Alternatively, the vector intrinsics from the first architecture can be mapped to the equivalent vector operation or sequence of vector operations of the second target architecture. This task can be facilitated by using translation libraries.

To date, widespread adoption of SIMD-based computing has been hampered by low programmer productivity due to the need for low-level programming to exploit the SIMD ISA.

### Limitations

While exploiting vector architectures with intrinsics allows more efficient exploitation, the speedup attainable with vector architectures is often limited. Some

of these limitations causing a program not to achieve performance increase when using vector processing are listed here.

### Sequential Code

The applications with data level parallelism gain the largest performance improvement when using vector unit. Not all applications can benefit from vector processing. For example, in a widely cited study performed at the University of California at Berkeley [6] finite-state machine (FSM)-based algorithms associated with text processing were identified as particularly challenging to leverage the parallelism that modern processors have made available. In these algorithms, a loop iterates over a pointer-based linked list. The current element has to be read in order to determine the memory location of the next memory element.

### Data Formatting

Layout of data in memory impacts performance of vector architectures. If data are laid out in the memory in the consecutive memory locations, data can be loaded into vector registers. Otherwise, data need to be formatted until they are in a format suitable for vector processing. This manipulation introduces overhead which diminishes or eliminates performance increase of vector processing. Applications that operate on non-contiguous memory location will not experience the same speedup due to SIMD.

The overhead due to formatting requires additional formatting instructions and increases register pressure in order to keep values in the register file as they are being formatted. Reduced number of registers can in turn reduce the effectiveness of compiler optimizations.

As of today, no processor provides hardware support for loading from or storing to non-contiguous memory locations by using some form of SIMD gather/scatter instructions.

Another data formatting factor which impacts performance of vector architectures is data alignment. An aligned reference means that the desired data reside at an address that is a multiple of the vector register size. Contiguous data are unaligned when they are not physically located at addresses that are multiple of the SIMD data width. For example, vector loads on Power 6 or on Cell PPU/SPU can only load data that start at

addresses that start at addresses which are aligned at 16-byte boundaries.

The compiler attempts to position data at aligned addresses, but the data address cannot be always determined at compile time, for example, when address calculation uses a result from previous operations. To ensure that data are aligned, an alignment sequence can be used. This sequence loads all necessary data, and some additional data, and then formats it accordingly to use only data dictated by the application. This introduces overhead and, in turn, reduces the performance advantage of using vector instructions. In the cases there is a large number of misaligned streams, performance speedup will be minimal due to increased instruction bandwidth and register pressure [7].

Recent vector architectures provide hardware support for loading contiguous memory locations that are not aligned at multiple of the vector length of the vector unit. Frequently, such loads are inefficient and have significantly higher latency compared to vector loads of aligned data. Inefficient hardware implementation may simply shift the software overhead to the hardware.

## Exemplary ISA Vector Extensions

### AltiVec and VMX

AltiVec is a floating-point and integer SIMD instruction-set architecture designed by the IBM, Motorola (the Motorola semiconductor division is now operating independently as Freescale Semiconductor), and Apple between 1996 and 1998.

AltiVec is implemented in a number of PowerPC processors. The first implementation of AltiVec was in Motorola's G4 PowerPC. IBM implements AltiVec in a number of IBM processors, in IBM's PowerPC 970 (also called G5 by Apple) and Power 6 processors. Since Freescale owns a trademark for AltiVec, the system is also referred to as IBM Vector Multimedia Extension VMX by IBM, and Velocity Engine by Apple. AltiVec was the first SIMD instruction set to gain widespread acceptance by programmers. Apple was the primary customer to adopt AltiVec in PowerPC-based Mac PCs.

AltiVec/VMX architects a dedicated set of vector register file with 32 128-bit registers that can represent 16 8-bit signed or unsigned characters, eight 16-bit signed or unsigned shorts, four 32-bit integers, or four 32-bit floating-point single precision variables. AltiVec

supports a special RGB “pixel” data type and provides cache-control instructions intended to minimize cache pollution when working on streams of data.

Most AltiVec instructions take three register operands, and there is a small number of four operand instructions. The four operand instructions include merged floating-point and integer multiply-add instructions, which is very important in achieving high floating-point performance, and a vector permute instruction. Another very useful four operand instruction is the permute instruction. It allows to take any byte of either of two input vectors, as specified in the third vector, and place it in the resulting vector. This allows for complex data manipulations in a single instruction and is very useful in a number of applications, ranging from image processing to cryptography.

AltiVec is a standard part of the Power ISA v.2.03 [8] and later specifications.

VMX128 is a modified version of VMX for the Xenon processor (used in Microsoft's XBox 360). It architects 128 registers and adds support for sum-across operations, where all the vector elements in a single vector register are added together to generate a single sum. These are application-specific operations for accelerating 3D graphics and game physics.

The Blue Gene/L and BlueGene/P supercomputers use a double-precision floating-point SIMD unit. This unit uses 128-bit vector registers to contain two double-precision floating-point values, and the scalar and vector registers share the same register file.

### MMX

MMX is a SIMD instruction set designed by Intel and implemented in the Pentium processor in 1997.

MMX used 64-bit vector data, which were stored in the floating-point registers. Processing was performed on vectors of eight 8-bit elements, four 16-bit elements, or two 32-bit elements by a dedicated graphics processing unit. The mapping of MMX registers onto floating-point registers made difficult to use floating-point and SIMD data in the same application, and required mode switching. MMX provided only integer operations [9].

AMD designed an extension to the x86 instruction set with the 3DNow! instruction set. 3DNow! added floating-point support to MMX SIMD instructions. This first implementation of 3DNow! instruction set was in the AMD's K6-2 processor in 1998.

## SSE

Streaming SIMD Extensions (SSE) is a SIMD instruction set extension of the x86 architecture. SSE was designed by Intel and introduced in 1999. The first SSE implementations were in Intel's Pentium III processors and later in AMD's Athlon XP and Duron processors.

SSE uses a separate set of vector registers and general purpose registers. It has eight 128-bit registers that can represent sixteen 8-bit bytes or characters, eight 16-bit short integers, four 32-bit integers, or four 32-bit floating-point single precision variables, or two 64-bit integers or two 64-bit double-precision floating-point numbers. SSE includes integer and floating-point instructions. Unlike MMX instruction set, SSE does not reuse the floating-point registers but implements separate registers for SIMD. This allows interleaving of SSE and scalar floating-point operations without having to switch between the two modes.

Since the introduction of SSE, SSE has gone through several revisions (SSE, SSE2, SSE3, SSE4). SSE2 was introduced in Pentium 4 processors in 2002. SSE2 adds new instructions for double-precision (64-bit) floating point. SSE3 was introduced in new Pentium 4 chips in 2004. SSE3 adds the capability to work horizontally in a register such as instructions to add and subtract the multiple values stored within a single register. SSE4 was introduced in the Intel Core processor in 2006.

## Cell SPE

The Cell Synergistic Processor Architecture (SPE) instruction set is designed by IBM, Sony, and Toshiba. It is implemented in IBM's Cell BE processor for Sony's PlayStation 3 in 2006 and in PowerXCell 8i processor by IBM in 2008.

The Cell SPE architecture implements a single 128-entry 128-bit register file for storing both scalar and vector data and supports both scalar and SIMD processing. The SPE can perform four single precision, or two double-precision floating-point operations, sixteen 8-bit integer, eight 16-bit integer, or four 32-bit integer operations.

Many architectures separate scalar registers and execution units from SIMD registers and execution units. Some applications (for example, as text processing) require frequent data movement between the two sets of registers. The movement of data between vector and general purpose registers is achieved by storing data

from one set of registers and loading it in the another registers. This type of transfer incurs a significant delay often wiping out any performance gains that might be gotten from exploiting SIMD data parallelism.

Cell SPE architects a unified register file for storing both short vector and scalar values, and it reuses its execution units for both scalar and vector processing. Scalar values are stored in the leftmost slot of the vector register, the so-called "preferred slot." The unified register file makes data sharing between scalar and SIMD vector operations straightforward and incurs no delay.

## VSX

VSX (Vector-Scalar Extension) is a new SIMD instruction set designed by IBM. It is first implemented in IBM's Power 7 processor in 2009 and described in Power ISA v2.06 [10].

VSX implements 64 SIMD registers and includes instructions for double-precision floating point, decimal floating point, and vector execution. VSX implements a unified register file, where floating-point registers and VMX/AltiVec vector registers are mapped to the VSX vector registers. VSX SIMD floating-point operations use 64 128-bit vector registers (32 of them overlayed on top of the VMX registers, and 32 overlaid on top of the scalar floating-point registers) [11].

VSX implements the concept of "preferred slot," as introduced in Cell's SPE, where the leftmost element of a register can be used as both scalar and vector element. The data from the merged vector-scalar floating-point registers are used for scalar and vector floating-point processing, as well as for vector integer processing. Furthermore, VSX introduces double-precision floating-point vector operations to increase parallelism in double-precision floating-point processing.

## Bibliography

1. Kohn L, Margulis N (1989) Introducing the Intel i860 64-bit microprocessor. *IEEE Micro* 9(4):15–30
2. Lee R (1997) Effectiveness of the MAX-2 multimedia extensions for PA-RISC 2.0 processors. *HotChips IX*, Palo Alto, 24–26 August 1997
3. Diefendorff K, Dubey P, Hochsprung R, Scales H (2000) AltiVec extension to PowerPC accelerates media processing. *IEEE Micro* 20(2):85–95

4. Gschwind M, Hofstee P, Flachs B, Hopkins M, Watanabe Y, Yamazaki T (2006) Synergistic processing in cell's multicore architecture. *IEEE Micro* 26(2):10–24
5. Salapura V, Walkup R, Gara A (2006) Exploiting workload parallelism for performance and power optimization in BlueGene. *IEEE Micro* 26(5):67–81
6. Asanovic K, Bodik R, Catanzaro BC, Gebis JJ, Husbands P, Keutzer K, Patterson DA, Plishker WL, Shalf J, Williams SW, Yelick KA (2006) The landscape of parallel computing research: a view from Berkeley. Technical Report, UCB/EECS-2006-183, EECS Department, University of California, Berkeley
7. Eichenberger A, Wu P, O'Brien K (2004) Vectorization for SIMD architectures with alignment constraints. In: Programming language design and implementation. ACM, New York
8. Power ISA v.2.03, Power.org, 2006-08-29. [http://www.power.org/resources/downloads/PowerISA\\_203.Public.pdf](http://www.power.org/resources/downloads/PowerISA_203.Public.pdf)
9. Peleg A, Weiser U (1996) MMX technology extension to the Intel architecture. *IEEE Micro* 16(4):42–50
10. Power ISA Version 2.06. Power.org, 2009-02-10. [http://www.power.org/resources/downloads/PowerISA\\_V2.06\\_PUBLIC.pdf](http://www.power.org/resources/downloads/PowerISA_V2.06_PUBLIC.pdf)
11. Gschwind M, Olsson B (2009) Multi-addressable register file. Patent Application US-20090198966, Aug 2009

## VLIW Processors

JOSEPH A. FISHER<sup>1</sup>, PAOLO FARABOSCHI<sup>2</sup>, CLIFF YOUNG<sup>3</sup>

<sup>1</sup>Miami Beach, FL, USA

<sup>2</sup>Hewlett Packard, Sant Cugat del Valles, Spain

<sup>3</sup>D. E. Shaw Research, New York, NY, USA

## Definition

VLIW (*Very Long Instruction Word*) is a CPU architectural style that offers large amounts of irregular instruction-level parallelism (ILP) by overlapping the execution of multiple machine-level operations within a single flow of control. In a VLIW, the instruction-level parallelism is visible in the machine-level program and must be exposed and arranged before programs run; this complex job is done using sophisticated compiler technology, with little, if any, help from the programmer. A classic organization of a VLIW instruction consists of many individual operations bundled together into a long instruction word, with one such word issued each processor cycle. VLIW processors are used extensively in high-performance embedded applications, and have found some success as high-performance servers.

## Discussion

VLIW architectures offer large amounts of instruction-level parallelism by arranging a parallel execution pattern in advance of the running of the program. The parallelism is carried out among multiple machine-level operations, typically RISC-style, within a single flow of control. In a VLIW, a single program counter is used to determine the instruction stream, and instructions (each of which might itself include multiple operations) are fetched and dispatched one at a time with no rearrangements of their order done in the hardware. The execution arrangement is orchestrated in advance of the running of the program. The hardware simply fetches, decodes, and issues the wide instructions in an in-order fashion, preserving the order that the compiler had previously established.

Unlike vector architectures which require code with a regular form of parallelism where the same operation is applied to parallel data, VLIW parallelism does not rely upon such regular patterns being identified and presented to special execution units. The identification and then choreography of “irregular” parallelism can

## Vectorization

- FORGE
- Parallelization, Automatic
- Parafrase

## Verification of Parallel Shared-Memory Programs, Owicky-Gries Method of Axiomatic

- Owicky-Gries Method of Axiomatic Verification

## View from Berkeley

- Green Flash: Climate Machine (LBNL)

## Virtual Shared Memory

- Software Distributed Shared Memory

be extremely complex, and since VLIWs must look like ordinary processors to the user, sophisticated compiler technology is required for a VLIW to be effective. This effect is so strong that VLIW architectures are often co-designed with a matching compiler technology.

The “very long” instructions are what inspired the name VLIW, which provides a good mnemonic device for the key architectural characteristics. There is, however, no requirement that a VLIW actually have “long instructions.” Rather, the key defining characteristic is that the single-stream ILP is exposed in the program, and thus planned in advance, rather than organized on-the-fly while the program runs. Operations might instead be statically scheduled to be issued rapid-fire, while several previous operations are still in flight. Such an architecture style, sometimes called *superpipelining*, has the same key characteristics as one with actual long instructions, differing only in implementation details, and can be considered as a form of VLIW where statically scheduled parallel instructions overlap over time.

VLIWs are often juxtaposed with Superscalar architectures, since both have the goal of rearranging the instruction stream to effect a speedup via the parallel execution of simple operations on multiple functional units. The two design styles differ fundamentally in when and by what mechanism the rearrangement is done: whether beforehand at compile time (VLIW), or via specialized hardware operating each cycle while the program runs (Superscalar).

Real implementations have interpreted the “Very” in VLIW in rather different ways, depending on the target domain, the available circuit, and compiler technology. For example, in high performance computing, the Multiflow Trace architecture (a 1984–1990 mini-supercomputer) could issue up to 28 operations per cycle. In the embedded space, the vast majority of VLIW architectures to date issue between 4 and 8 operations per cycle.

Two techniques that combine architectural support with compiler transformations, speculative execution and predicated execution, offer complementary methods to increase instruction-level parallelism in ILP processors. In *speculative execution*, the compiler moves operations above conditional branches that dominate the original location of the operation. Because the branch is conditional, this code motion is a gamble: if the compiler guesses right, the program runs faster; if

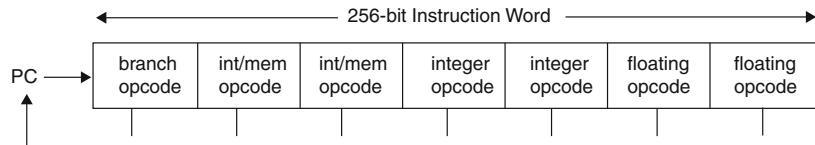
not, the program will have wasted execution resources. Care must be taken to suppress side effects (e.g., memory stores or exceptions) that would change the semantics of the program if the speculated operation is incorrectly executed. In *predicated execution*, the instruction-set architecture of the machine supports additional input registers to operations that conditionally enable or disable (viz., predicate) the operation. Compiler optimizations can then choose to predicate operations from different control paths and fuse those paths into a single flow of control. Predication can reduce branch-related pipeline penalties and enable further scalar optimizations, but it comes at the cost of a more complicated implementation and potentially increased instruction fetch bandwidth when the operations from all fused paths must be fetched. These two techniques are covered by other articles in this encyclopedia, and will not be discussed in more detail in this article.

## VLIW Implementation

A classic VLIW implementation might have instruction encodings that look like the following, from the Multiflow Trace 7/300 (1987–1990) ([Fig. 1](#)).

The Trace 7/300 issues seven operations per clock cycle: one branch, two integer operations, two memory operations (either of which can optionally be an integer operation instead), and two floating operations. A machine-level program is a sequence of such instructions, as shown below, with one instruction issued per clock cycle, unless the CPU stalls ([Fig. 2](#)).

Classic VLIW execution unit hardware usually consists of several execution functional units, several register banks and paths to memory arranged in a crossbar for unrestricted access to all in any cycle, as shown below. Implementation realities will often make such rich connection not scalable beyond a certain number of fully connected units, because of the non-linear relationship between the number of execution units and the area and complexity of the register file and the bypass (forwarding) logic. Sometimes functional units can be arranged to only be connected to a subset of the register banks – for example, two identical integer ALUs might each address a different register bank. In this case, the partitioned-register architecture is referred to as a *clustered VLIW* – successfully dealing with clustered register banks is a compiler research topic of its own.



VLIW Processors. Fig. 1 A possible encoding of a 256-bit wide VLIW instruction composed of 7 operations

|                | branch opcode | int/mem opcode | int/mem opcode | integer opcode | integer opcode | floating opcode | floating opcode |
|----------------|---------------|----------------|----------------|----------------|----------------|-----------------|-----------------|
| Instruction 01 |               |                |                |                |                |                 |                 |
| Instruction 02 |               |                |                |                |                |                 |                 |
| Instruction 03 |               |                |                |                |                |                 |                 |
| Instruction 04 |               |                |                |                |                |                 |                 |
| Instruction 05 |               |                |                |                |                |                 |                 |
| Instruction 06 |               |                |                |                |                |                 |                 |
| Instruction 07 |               |                |                |                |                |                 |                 |
| Instruction 08 |               |                |                |                |                |                 |                 |
| Instruction 09 |               |                |                |                |                |                 |                 |
| Instruction 10 |               |                |                |                |                |                 |                 |
| Instruction 11 |               |                |                |                |                |                 |                 |

VLIW Processors. Fig. 2 A machine-level program with one instruction issued per clock cycle, unless the CPU stalls

The two figures below show the block diagrams of (unclustered and clustered) VLIW architectures (Fig. 3).

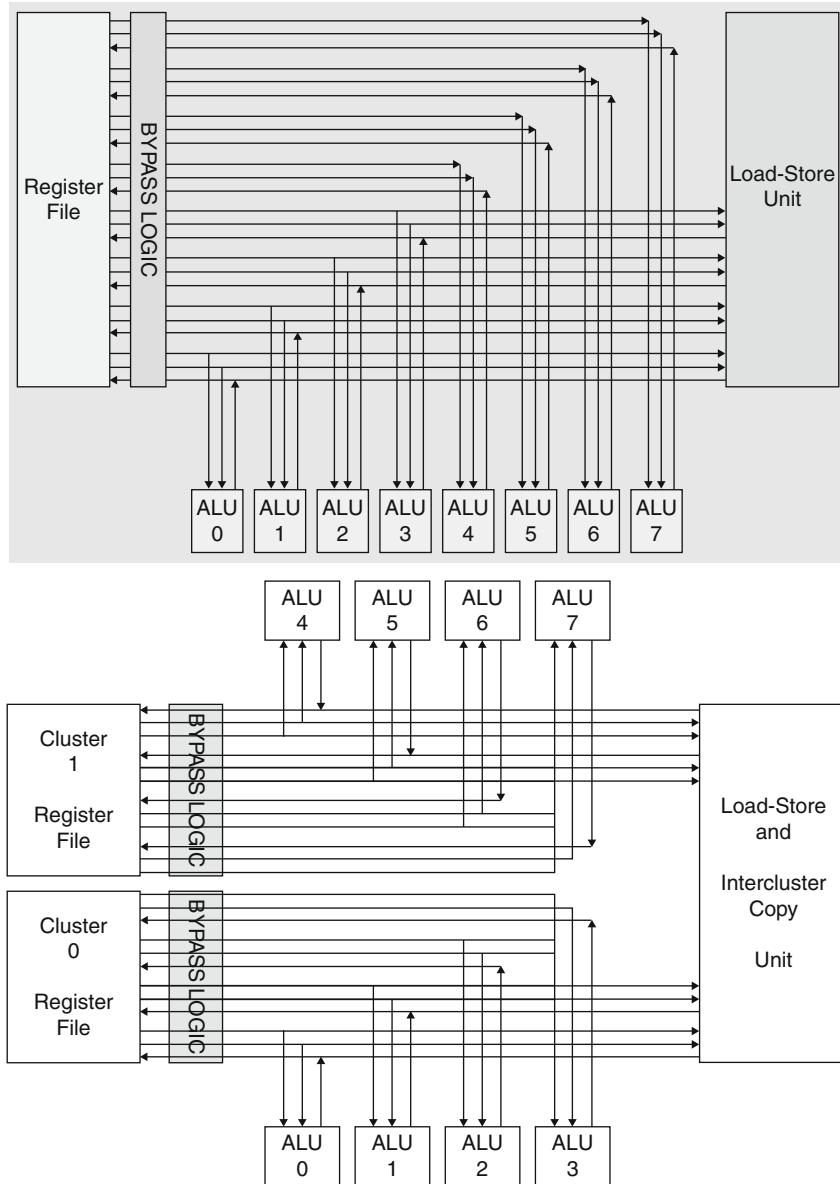
### Benefits of VLIW

The VLIW design style exists to provide effective high performance, with little if any programmer intervention, and without great hardware cost. A VLIW usually requires less hardware than other ILP approaches (such as superscalar or dataflow) to high performance, since

the choreography of parallelism is done in advance and there is no need for hardware to rearrange the instruction execution sequence at run time.

The absence of rearrangement hardware, running every cycle, has broad implications for performance, cost, and power:

- A faster clock is possible (since there is less to do each cycle).
- There is less hardware, thus a lower silicon cost.



VLIW Processors. Fig. 3 Block diagrams of (unclustered and clustered) VLIW architecture

- Less power is used.
- Far more ILP is practical, since the additional hardware needed to rearrange computations tends to grow exponentially with the amount of ILP available.

A result of this is that ILP in the range of 4–8 operations issued per cycle is common in VLIWs. The Multiflow Trace 28/300 offered 28 operations issued per cycle, and

for some heavily computational code was able to sustain a large fraction of complete utilization.

These advantages exist primarily in comparison to superscalar architectures, the closest equivalent form of parallelism. When compared to other forms of parallelism, both VLIW and superscalar architectures have the advantage that the computing paradigm does not change, that is, parallel processes and regular structures do not have to be identified in the code. To the programmer, both are much like “normal” scalar processors.

## Object-Code Compatibility

The greatest shortcoming in VLIW's use as "ordinary" computers lies in its defining characteristic. Old programs will not run correctly on new machines where either the set of hardware resources or the number of cycles to perform an operation (i.e., its *latency*) has changed vs. what the compiler assumed when it generated the original object code. The exact computation to be carried out, and the machine on which to execute it, is visible in a VLIW binary program. Thus, a new implementation of a VLIW architecture is not likely to be object-code compatible with an old one, which represents a major flaw in VLIW's use as a general-purpose computer. One consequence of this shortcoming is that VLIWs have found far greater success in the embedded computing world, where object-code compatibility is far less of a factor than on general purpose computers such as desktops or servers.

If an architecture is not changed dramatically, there are several techniques that can help solve this problem, individually or in combination:

- If the new architecture is in some sense a superset of the old one, then the system can be designed to allow the old programs to run on a limited portion of the hardware, corresponding to the old architecture. Eventually, the code may be recompiled to get greater performance on the new architecture. This upward compatibility was used by Multiflow, where the system that issued 14 operations at a time could run code compiled for the system that issued 7. It is also used, in a different manner, in the Intel IA-64 where the compiler generates code assuming a superset of the resources are available, and each implementation may choose to only execute some operations in parallel.
- If the incompatibility results largely from changed latencies, then programs can carry information about their assumptions, and mode switches built into the CPU can enforce the correct timings (delaying any instruction issue when a component operation cannot be guaranteed to have finished).
- The system can statically change old programs into new ones that follow the new architectural assumptions at the time the program is loaded.
- The system can dynamically change old programs into new ones that follow the new architectural

assumptions while the program is running. This only makes sense if the translations can be cached and reused repeatedly. For example, the x86-compatible Transmeta processors (2000–2005) used a similar dynamic binary translation technology to generate VLIW instructions from a stream of x86 instructions on the fly.

## Other Perceived Disadvantages of VLIW

### Dynamic Code Behavior

Often compilers cannot predict the behavior of a program, most importantly in branch direction and in memory latency. A VLIW, having had its execution laid out in advance, can't adjust to these dynamic changes.

- To ameliorate the problem of unpredictable branch directions, a compiler uses static branch prediction to assume that the more likely branch will be taken, and inserts code to correct the program behavior when this is wrong. The development of these techniques (Trace Scheduling was the first) greatly increased the potential for ILP in ordinary programs. Short, hard-to-predict branches can also be eliminated through various forms of predication (also known as conditional, or guarded, execution).
- To ameliorate the problem of unpredictable memory latency, VLIWs often include decoupled register fetch mechanisms, and other dynamic features.

The designers of VLIW architectures have been careful to regard the VLIW design style not as a dogma, but rather as a design principle, to guide decisions but not mandate them. When dynamic features have been desirable, designers have felt free to use them. For example, the use of register scoreboarding reduces code size by removing the need for no-op operations. Dynamic branch prediction reduces the misprediction penalty by prefetching instructions among the most likely path. Stall-on-use memory techniques help ameliorate the L1 cache miss penalty when the compiler can schedule the consumer of a memory operation later than the shortest latency without impacting performance.

### Code Size

A VLIW can offer a great deal of parallelism, but most sections of code will only use a portion of what is offered. Code outside the dense inner loops will often

use only a very small part of what is offered. While performance improvements in such sections of code are very valuable, and VLIWs tend to do comparatively well there, most instructions still contain only a few operations to be carried out. That leaves many fields of the long instruction word blank, or filled with no-ops. Left untreated, this would cause programs to occupy an undesirably large amount of memory.

Every modern VLIW, from the 1980s forward, has dealt with this problem in some way. Sophisticated code compression techniques have been developed that bring a VLIW's code size down to the size of uncompressed RISC programs. Compression appears at every level of the instruction memory hierarchy, though typically the greatest degree of decompression is done as the instruction cache is filled.

### **Compiler Sophistication and Compile Time**

The job of arranging a VLIW's parallelism, which falls to the compiler, is difficult to engineer and time-consuming to run. While this is true, it is a fallacy to think of these effects as an undesirable quality of the VLIW design style, especially when that is compared to superscalars. The actual scheduling phase of a VLIW compiler, while complex, is fairly well understood and need not take a lot of compile time.

The hard part of the compiling process is doing the code rearrangements to expose the parallelism in the first place, including the branch prediction mentioned above. All of this is a function of the available instruction-level parallelism, not of the style of architecture that exploits it. The reason VLIW architectures require "heroic compilers" is that they offer so much ILP. If it were practical to build a superscalar architecture with that much ILP, it would require an equivalently heroic compiler.

### **Dealing with Exceptions**

Any system that does as much rearranging of a program as a VLIW does must deal with preserving the original program semantics in the face of exceptions. Even worse, rearranging programs by speculating the execution of an instruction before a controlling branch may cause exception operations to execute, when they would not have executed at all in the original program's sequential order. A vast body of technology, too much

to give justice to here, has grown to address this problem, including the concepts of "dismissible loads" and "delayed exceptions."

## **Important VLIW Systems**

### **Startup Computer Companies in the 1980s**

The first commercial VLIW systems were small supercomputers:

- The Trace 7/200, and its successors, introduced in 1987 by the startup Multiflow Computer
- The Cydra-5, introduced in 1987 by the startup Cydrome

About 120 Multiflow Traces were sold, most into mechanical, chemical, and electronic simulation environments. Despite Multiflow's initial success, neither company survived beyond 1990.

### **First VLIW Microprocessor**

After 9 years of design, the Philips Life-I (later renamed the TriMedia) was introduced in 1996, and was the first commercially successful VLIW Microprocessor. Eventually, the division of Philips that produced the TriMedia became the private company NXP Semiconductors, which still designs and builds VLIW microprocessors, the latest being the PNX1005, used in digital TVs and other digital audio and video products, as well as in cellular handsets and other applications.

### **Other Modern VLIW Microprocessors**

Today's most important VLIW microprocessors, and their main uses, include:

- The STMicroelectronics ST231 (used extensively in digital video and printing and scanning)
- The Texas Instruments C6x family (used extensively in cellular basestations and smartphones, and in-home entertainment)
- The Fujitsu FR-V (used in digital cameras)

Although companies do not generally release volumes for individual parts, ST has reported that over 70 million ST231 cores were shipped by April, 2009, and Texas Instruments report that over 80% of all cellular base stations contain VLIW processors. VLIW processors have been used successfully in products such as hearing aids, graphics boards, and network communications devices.

There are also numerous "Fabless" (Intellectual-Property-Based) VLIW Designs commercially available

from CEVA, Silicon Hive, Tensilica, and others. These processors are generally found as the high-performance core or cores in a larger system-on-a-chip.

### VLIW in Processors for General Purpose Use

During the decade of the 1990s, two prominent efforts were made to use VLIW in desktops, notebooks, and servers.

- Transmeta used a VLIW processor as the underlying engine to run an emulator of the Intel x86. However, Intel and AMD's strength in the marketplace, and the complexity of setting up a high-volume manufacturing operation was too much of a barrier for the excellent cost/performance characteristics of this product to overcome.
- Intel's Itanium processor family borrows many characteristics from VLIWs, while adding many new complex features of its own. Intel refers to the design style of that family as “EPIC” (explicitly parallel instruction computing). The Itanium borrows many design artifacts from the Cydrome, and founders of both Cydrome and Multiflow were involved in the initial design at HP Labs. While the Itanium is the fourth most popular server architecture today, it has not met industry projections for its volume, as its lack of compatibility and its ineffective early implementations have greatly limited its growth.

### VLIW was Motivated by Compiler Techniques

The Trace Scheduling compiler algorithm was put forward by Joseph Fisher [1] at the Courant Institute of New York University in 1979. Trace scheduling was designed to solve the “horizontal microcode compaction problem” by eliminating the restriction that overlapping operations must originate from the same basic block of straight-line code. No practical algorithm to do so had been put forward before that. The basic idea in trace scheduling has two steps. First, select code originating in a large region (typically many basic blocks). Second, compact operations from the entire region as if it were one big basic block (which often contains a lot of ILP), possibly adding extra operations to maintain the semantics of the program, despite the violence done to the flow of control.

Given that Trace Scheduling made far more ILP available, Fisher predicted that general-purpose systems

like today's VLIWs made sense, coining the term Instruction-level Parallelism for the kind of parallel processing [2]. While at Yale University, Fisher then defined and named the VLIW design style in 1981, publishing his thoughts in 1983 [3]. By January of 1987, Multiflow Computer, a company started by Fisher and members of his Yale University group, delivered high-performance commercial VLIWs to customers.

During that same period, software pipelining began to mature. Software pipelining is a compiler technique (originally, it was sometimes done by hand) to speed up inner loops. It does this by rearranging the loop so that operations from multiple iterations are computed within the same revised inner loop. Code is placed before (prologue) and after (epilogue) the new loop to preserve the program semantics, by starting the first iteration(s) and completing the last one(s).

Software pipelining was done by hand on early computers containing some ILP, and was carried out via simple pattern matching by the Fortran compiler for the CDC-6600 as early as the 1960s. By the early 1980s, the Floating Point Systems Fortran compiler could software pipeline a loop consisting of a single basic block [4, 5]. In 1981, Bob Rau [6] described a more scientific version of software pipelining, using the term “Modulo Scheduling.” His work helped lead to an early interest in VLIWs, and Rau went on to start Cydrome, which, along with Multiflow, pioneered VLIWs.

Modulo Scheduling researchers all favored “rotating registers,” as found in the FPS systems. Rotating registers change their addresses to match the code iterations in modulo scheduled loop. These have been shown to not be necessary, and whether they are beneficial is still controversial. They are found in the Itanium and (in a more restricted form) in the TI C6x.

### Precursors of VLIWs

During the 1960s and 1970s, three different architectural trends led the way toward VLIWs.

1. *High-performance general-purpose scientific computers.* These were the supercomputers of their day. The best examples are the CDC-6600 and the IBM 360/91, which was preceded by the IBM research processor Stretch. They derived ILP by performing several arithmetic operations simultaneously, in a manner arranged by the hardware (like today's

superscalars). Although the hardware could identify independent operations and schedule their issuance, practitioners quickly realized that a compiler could place operations close enough to each other in the machine-level program that the hardware, with its limited visibility, could overlap them more effectively. Because these systems typically overlapped only 2 or 3 operations, the compiler's rearrangement only needed to span short sections of straight-line code in the source program.

These systems were general-purpose ILP CPUs. Primitive software pipelining and basic block instruction scheduling both originated in this environment. Researchers working on these systems, convinced by flawed experiments that available ILP was limited to a factor of 2 or 3 times speedup, never pushed the superscalar aspects of their technology farther.

2. *Attached signal processors, or “array processors.”* In the 1970s, and perhaps earlier, engineers and scientists built hardware that resembled VLIW, called attached signal processors, or “array processors.” The most famous of these were the Floating Point Systems AP-120b (introduced in 1975) and FPS-164 (introduced in 1980), though there were many others, offered by Numerix, Mercury, CDC and Texas Instruments, among others. Although they evoke today’s VLIWs, they were typically built to compute a particular inner loop very effectively, and were thus very idiosyncratic, and their parallelism was almost always specified by hand, rather than via compiling. Circuits are still built in this style, and are called DSPs (Digital Signal Processors).
3. *Horizontal microcode.* Microcode is an RISC-like level of code, sometimes found hard wired in a CPU to emulate the true, more complex, machine-level architecture. Since the emulator is a single program, running all the time, it was straightforward to build “horizontal microcode,” in which several of the pieces of hardware run in parallel. This again resembles the hardware structure of VLIW.

In the latter two domains, compiling was regarded as desirable, but only as an afterthought. It was never very effective, but research into compiling in those domains led to the compiler breakthroughs that enabled VLIW.

## Related Entries

- [Cydra 5](#)
- [Dependence Analysis](#)
- [Instruction-Level Parallelism](#)
- [Modulo Scheduling and Loop Pipelining](#)
- [Multiflow Computer](#)
- [Speculation](#)
- [Trace Scheduling](#)

## Bibliography

1. Fisher JA (1979) The optimization of horizontal microcode within and beyond basic blocks: an application of processor scheduling with resources. Ph.D. dissertation, Technical Report COO-3077-161. Courant Mathematics and Computing Laboratory, New York University, New York
2. Fisher JA (1981) Trace scheduling: a technique for global microcode compaction. IEEE T Comput 30(7):478–490
3. Fisher JA (1983) Very long instruction word architectures and the ELI-512. In: Proceedings of the 10th annual international symposium on computer architecture, Stockholm, Sweden, 13–17 June 1983, pp 140–150
4. Charlesworth A (1981) An approach to scientific array processing: the architectural design of the AP-120b/FPS-164 family. IEEE Comput 14(3):18–27
5. Touzeau RF (1984) A FORTRAN compiler for the FPS-164 scientific computer. In: Proceedings of ACM SIGPLAN’ 84 symposium on compiler construction, Montreal, pp 48–57
6. Rau BR, Glaeser CD (198) Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In: Proceedings of the 14th annual microprogramming workshop, IEEE Press, Piscataway, pp 183–197
7. Fisher JA, Faraboschi P, Young C (2005) Embedded computing – a VLIW approach to architecture, compiler, and tools. Morgan Kaufmann, San Francisco

## VLSI Algorithms

- [VLSI Computation](#)

## VLSI Computation

FRANCO P. PREPARATA  
Brown University, Providence, RI, USA

## Synonyms

[Application-specific integrated circuits](#); [VLSI algorithms](#)

## Definition

VLSI Computation (computation within the Very-Large-Scale-Integrated technology) concerns the analysis of the computations realized by large integrated networks, whereby the traditional distinction between networks and computations disappears (each network is “dedicated” to the execution of a particular algorithm). Specific algorithms realized by such circuits are evaluated in terms of their efficient use of the integrated technology.

## Discussion

### Historical Background

Switching networks, the basic constituents of digital systems, were traditionally realized by interconnecting discrete electrical components, such as transistors, resistors, capacitors, etc., by means of conducting wires.

Transistors, the fundamental switching elements, were initially realized by appropriately modifying (“doping”) different portions of the surface of a piece of semiconductor material. A key innovation occurred in the late 1950s of the past century, when it was realized that conducting wires could also be implemented on the same surface, either by deposition of metal or by modification of the semiconductor. The consequence was that several distinct transistors could be fabricated on the *same* piece of material, along with appropriate interconnecting wires. This discovery ushered in the era of integrated circuits, initially as interconnections of a few gates, called *micrologics*. The technology progressed rapidly by reducing the dimensions of the individual transistor and therefore increasing the number of transistors realizable within a fixed area of material. In those days (1965) *Moore’s law* was formulated, the empirical observation that the number of transistors that could be placed on an individual integrated circuit would double approximately every two years.

In a few years it was possible to bring to market extremely simple processors, entirely realized on a single plate of semiconductor (the *chip*) and acting on operands of very few bits (4 or 8). The progression, however, was extremely rapid: by the late 1970s, the computer science and engineering community became fully aware of a mature technology of large-scale integrated circuits, and for the theoretical component of this community a new model of computation emerged:

the discipline of VLSI computation was therefore established within the scope of computer science research and became the theoretical underpinning of massive parallel computation. Although today parallel computation research pursues different directions, VLSI computation remains a rich theoretical acquisition.

### A Model of VLSI Computation

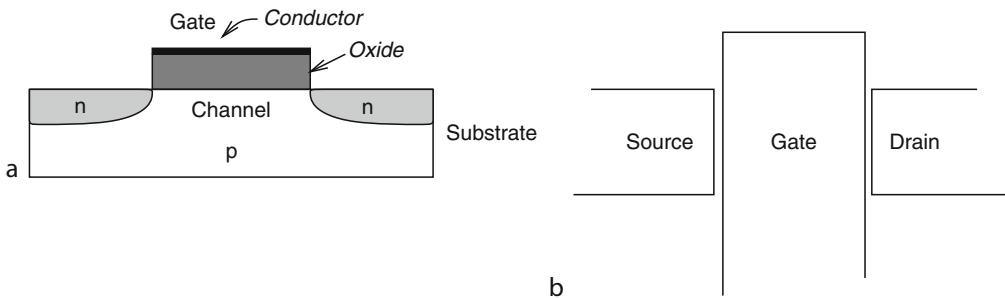
A paramount theoretical issue is the formulation of the computation model to be adopted. The computation model defines the framework within which performance is to be evaluated; it defines the nature of the usable resources and their respective costs. As is typical of all models, we expect the VLSI model to be reasonably simple to facilitate mathematical treatment, but sufficiently faithful to the physical reality to provide an adequate level of confidence to predictions based on the model. This goal calls here for a review, albeit extremely sketchy, of the technology.

The fundamental element of a widely used, but by no means exclusive, VLSI technology is the Metal-Oxide-Semiconductor (MOS) transistor (other technologies can be treated analogously). Its geometry is illustrated in Fig. 1. In Fig. 1a a thin silicon wafer is selectively doped, by diffusion of appropriate atoms, into a p-type *substrate* and two n-type electrodes called *source* and *drain*, separated by a gap called *channel*. On top of this gap a third electrode, called *gate*, is created by depositing an (insulating) oxide layer and a conducting layer on top of it. A voltage applied to the gate creates a conducting channel between source and drain. The control exerted by the gate is the hallmark of the transistor.

Revealing is the top view of the device (Fig. 1b). Here the transistor may be viewed simply as a (horizontal) strip comprising source and drain “crossed” by a (vertical) strip containing the gate. Similarly, a wire joining different electrodes appears as a strip of conducting material on the surface of the chip. The conclusion is that a VLSI circuit is described by the geometry of an interconnection of strips, representing devices and wires.

We can now elucidate the abstraction process leading from technology to model. The model assumptions can be subdivided into three groups:

1. Network layout on the semiconductor chip.



VLSI Computation. Fig. 1 Geometry of the MOS transistor

The physical layout of a VLSI circuit must be as compact as possible. However, manufacturing must guarantee the integrity of the circuit, that is, the layout must comply with *design rules* for transistors and wires (to avoid accidental interruption a wire cannot be too narrow, to avoid shorts two wires cannot be laid out too close to each other). Therefore, the design rules for the layout strips mainly concern widths and separations, and are expressed in terms of a technology-independent parameter  $\lambda$  called *feature size*, typically expressing the sum of the width of a wire and its separation from adjacent layout items (The decrease of  $\lambda$  is a measure of technological progress: in fact  $\lambda$  has decreased by nearly two orders of magnitude since the late 1970s). Another important parameter is the number  $v$  of usable layers, a parameter that over the years has grown appreciably over its original value (of just 2). However, VLSI technology is essentially two-dimensional. The fundamental reason, beside ease of manufacturing, is heat dissipation: as  $\lambda$  decreases, higher clock speed and shorter transistor switching times are attainable. This results in higher production of heat, which must be dissipated in a medium in contact with the chip surface.

The area of an integrated circuit is obviously the sum of the areas of its transistors, its wires, and its input/output pads. We shall later see the dominant role of wire area in fast circuits.

## 2. Timing of signal transmission from device to device.

The elementary action involved in timing considerations is the switching of a transistor and the propagation of its output signal to another transistor. In a simplified view, this action is the cascade of switching time and propagation time. The latter, due

to the finiteness of the speed of light, clearly grows with the length of the wire. However, in compact circuits switching time dominates propagation time, and this circumstantial evidence was used originally to axiomatize that the time of an elementary action is constant (independent of the wire layout length).

## 3. Input/output protocols.

To be honest with respect to the use of area, each input is assumed to be read just *once* (denoted *time-determinate*) and in just *one pad* (denoted *place-determinate*), so that it remains stored within the chip. In addition, each input and output is uniquely identified by its time of use and by its place of occurrence. This provision excludes trivial algorithms, where one could claim sorting just by the order in which randomly permuted inputs are read out as outputs.

We can therefore summarize the model discussed above as follows:

- VLSI networks are synchronous boolean machines with the following features, presented here as the axioms of the model:
  - Axiom 1. Wires have minimum width  $\lambda$ , transistors have minimum area  $C_T \lambda^2$ , I/O pads have minimum area  $C_P \lambda^2$  ( $C_T, C_P$  are technological parameters).
  - Axiom 2. The time of an elementary action is constant (synchronicity).
  - Axiom 3. Inputs are read once and in one place. Input/outputs are time- and place-determinate.

For its simplicity, this VLSI model belongs to the best computer science tradition of machine models. Its salient features dispense with details and aim to

capture the essentials of the technology. As we shall discuss later, its soft spot is the timing assumption (Point 2 above).

### Area-Time Trade-offs

The primary concern of the design of algorithms is to establish bounds on the resources used by a computational task. In the context of VLSI the crucial resources are computation time (number of time units) and circuit size (area). Here, as in any other area of algorithmic research, we try to find intrinsic relationships between resources (in the form of lower bounds), and then design algorithms that possibly achieve these bounds.

The relation between time and area is based on the evaluation of the flow of information required for the correct execution of a computation. Specifically, we wish to identify sections (lines) in the chip through which a given flow of information is required by the computation. Two types of sections can be identified:

1. Input/output boundary
2. Internal sections

The boundary trivially defines its required flow, as information proportional to the problem size, conventionally  $n$ . More subtle is the analysis of the internal flow.

Assume, for simplicity, that the input bits are stored uniformly within the two-dimensional chip domain. A *balanced bisection* is a curve that divides the domain, and correspondingly the input set, into two parts of about the same size. Assume also that a computational argument shows that  $I$  bits must be exchanged through the bisection to correctly complete the computation. Then, if  $d$  is the length of the bisection, the number  $w$  of wires crossing the bisection satisfies  $w \leq vd/\lambda$ ; in clocked operation, at least  $I/w$  clock cycles (time units) are required. All we need is to relate length  $d$  to the chip area  $A$ . Simple arguments show that  $A \geq d^2/2$ , so that, denoting  $T$  the number of time units, we conclude

$$T \geq \frac{I\lambda}{v\sqrt{2A}} \quad \text{or} \quad AT^2 \geq cI^2$$

for some technological constant  $c$ . This important relationship exhibits a trade-off between area and time: it gives formal substance to the intuitive expectation that, for a given problem, faster VLSI circuits have larger size.

As we shall see in the next section, several important problems are covered by this analysis.

It is worth mentioning that, according to the preceding analysis, as we increase  $T$  the area  $A$  correspondingly decreases. There are problems, like the sorting of very long keys, where the area may become too small to store the data, that is, it becomes *saturated*. When saturated information is the mechanism limiting performance, optimal circuits have an area growing with  $1/T$  rather than  $1/T^2$ .

Finally, the outlined bisection-width analysis applies to a class of computations (a large and significant one) where the internal information flow depends upon the entire input set. However, there are other significant problems that are not covered by this rubric. These are the computations where subsets of the output variables depend upon subsets of the input variables, so that, for any such pair of (input,output) subsets, no variable can be output before all variables of the corresponding input set have been stored in the network. Moreover, the boolean circuit computing the output variables has depth at least logarithmic in the size of its input set. In other words, each input must remain stored for some time within the chip, that is, the flow of information is being slowed down. This behavior, dubbed *computational friction*, is characterized as the following area-time trade-off:

$$\frac{AT}{\log A} \geq c''I.$$

### VLSI Architectures and Algorithms

The preceding section outlines, in a very succinct form, the intrinsic limitations placed by the VLSI technology on computations realized by networks. A crucial feature of VLSI computation is the potential of being able to select, for any given application, the targeted computation time and to attempt to design a network realizing it. So, at one end of the spectrum we have sequential circuits, and the other the fastest parallelizations.

The objective of this section is to consider classes of important computations, to describe their VLSI implementations, and to compare their performance with the lower-bounds developed in the previous section.

Preliminarily, we must complete the definition of VLSI machine model with the notions of “architecture” and “algorithm.”

## 1. Architecture

- An *architecture* is a graph  $G = (V, E)$ , whose nodes  $V$  are modules and whose edges  $E$  are (bundles of) wires.
- The *diameter* of an architecture is the maximum of the lengths of the shortest path between any two of its modules. (A large diameter is  $\Theta(\sqrt{A})$ , a small diameter is  $\Theta(\log A)$ .)
- A *module* is the elementary component of an architecture. It could be as simple as a transistor or as complex as a microprocessor. The structure of the module is assumed independent of the size of the network.
- Operation is *synchronous*, that is, clocked.

## 2. Algorithm

- $X = \{x_1, \dots, x_n\}$  are *variables* and  $K = \{k_1, \dots, k_m\}$  are *constants*.
- A *parallel step*  $\sigma$  is a collection of assignments

$$\sigma = \{x_i \leftarrow f_j(X \cup K) \mid i = 1 \dots n, f_j \in \mathcal{F}\}$$

where  $\mathcal{F}$  is a set of functions for which hardware is duly provided in the network. The semantics is that all assignments in set  $\sigma$  are simultaneously executed.

- A *parallel computation* is a time sequence

$$\sigma_1 \sigma_2 \dots \sigma_p$$

- A *parallel algorithm* is a procedure executed as a parallel computation.

A most significant achievement of research in VLSI computation has been the identification of a few fundamental interconnections (networks) and of their natural association with classes of computations. Most interesting architectures results from the composition of such interconnections. Even more revealing is the identification of classes of computations matched to each of such architectures. Such classes are unified not by superficial similarity of applications, but by the communication patterns, or data exchanges, implied by their algorithms. Such fundamental interconnections are discussed next.

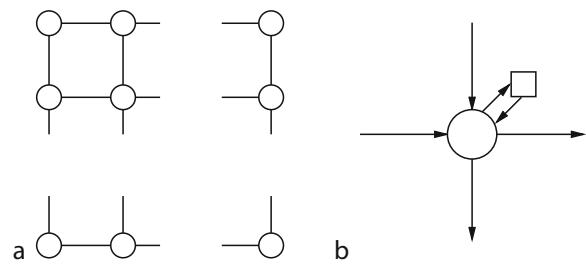
## Meshes

Meshes are the simplest architectures. Although higher-dimensional constructions are conceivable, dimensions 1 (array), 2 (grids or meshes, for short), and 3 (three-dimensional grids) have practical value. The array is a chain of modules, where each module communicates,

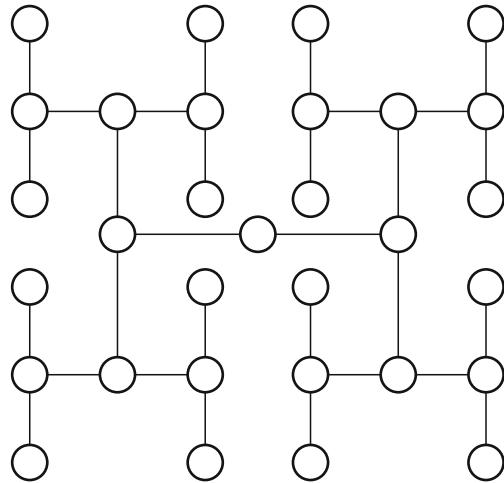
typically, with its adjacent modules. The mesh is a network whose modules are conventionally placed at the vertices of a regular tiling (triangular, square, etc.) of the plane. The most natural mesh, and the one discussed hereafter, is the rectangular  $n \times m$  planar grid, whose modules are placed at points of integer coordinates and each communicates with its four adjacent modules (see Fig. 2a).

A typical use of meshes is as *systolic networks*. The qualifier “systolic” suggests that data is fed to the network in waves to be processed in pipeline fashion. A useful illustration is provided by matrix multiplication, that is, the computation  $C = AB$  ( $A, B$ , and  $C$  are square matrices of dimension  $n$ ). The network is an  $n \times n$  square mesh, whose simple modules (inner-product modules, see Fig. 2b) keep stored one operand  $z$  (initialized to 0), receive inputs  $x$  and  $y$  from their upstream neighbors, perform the update  $z \leftarrow z + x \cdot y$ , and supply the updated  $z$  to their downstream neighbors. Inputs enter the network as waves from the left ( $A$ ) and from the top ( $B$ ): these waves, however, are not the conventional rows and columns, but cross-diagonals, that is, vectors whose components have identical sums of their two indices. The product appears stored in the mesh at the completion of the systolic flow. The network has  $O(n^2)$  area and the computation is completed in  $n$  steps, so that  $AT^2 = O(n^4)$ . Since the corresponding area-time lower-bound is  $\Omega(n^4)$ , this systolic network provides an optimal implementation. Note that the diameter of this network is  $O(\sqrt{N})$ , where  $N$  is the number of its modules.

Systolic implementations have been proposed for a variety of other problems, such as matrix-vector multiplication, open convolution, integer multiplication, solution of lower-triangular linear systems, etc. The



**VLSI Computation. Fig. 2** (a) A two-dimensional mesh;  
(b) inner-product module



**VLSI Computation. Fig. 3** The H-tree layout of a 16-leaf tree network

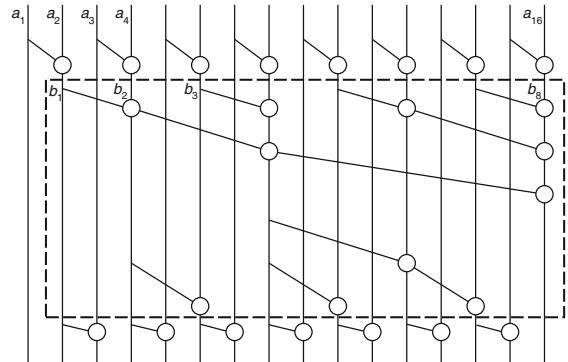
resulting algorithms are, in general, simple and elegant; however, none of these appears to be area-time optimal.

We shall return to meshes as simulators of other more complex architectures.

### Tree-Based Networks

The basic interconnection is represented by a rooted full binary tree. The standard planar layout of a tree with  $n = 2^j$  leaves for  $j$  even has the shape of a square of side-length  $L_j$  and is denoted H-tree by its typical appearance (see Fig. 3). Denoting  $L_0$  the side-length of a module, we have the simple recurrence  $L_j = 2L_{j-2} + L_0$ , yielding  $L_j = L_0 2^{j/2}$ , that is  $A = O(n)$ ; as is intuitive, a bound of the same order is achievable by deploying larger modules as we proceed from the leaves to the root. To be noted is that a tree network has small (logarithmic) diameter.

An important application of tree networks is the implementation of semigroup prefix computations, frequently referred to as “scans.” The operations are, typically, AND, OR, MAX, MIN, ADD, etc., generically denoted here as “ $*$ ”. We are given a sequence  $a_1, a_2, \dots, a_n$  and wish to compute the prefixes  $a_1, a_1 * a_2, a_1 * a_2 * a_3, \dots, a_1 * a_2 * \dots * a_n$  ( $n$  a power of 2, for simplicity). Sequentially, this is trivially doable in time  $O(n)$  with constant area. What is the performance of a fastest (logarithmic time) implementation? Figure 4 supplies the answer: The recursive box is a prefix computer for the sequence  $b_1, b_2, \dots, b_{n/2}$  where



**VLSI Computation. Fig. 4** A circuit for prefix computation

$b_i = a_{2i-1} * a_{2i}$ . We recognize that the explicit part of the diagram consists of the leafward two levels of two binary trees traversed in opposite directions (in addition, the bottom one misses its leftmost edge), so that we conclude that prefixes can be computed by two back-to-back tree networks. Such network can be realized with the previously described H-tree layout, and we reach the conclusion that the network area is  $O(n)$  and the number of steps is  $O(\log n)$ . Lower-bound considerations show that prefix computations are governed by

$$\frac{AT}{\log A} = \Omega(n)$$

Since  $AT/\log A = O(n \log n / \log n) = O(n)$ , the described implementation is area-time optimal. (Note that the sequential implementation is also optimal.)

Prefix computations include binary addition, comparison-exchange, sorting of long keys, etc.

### Hypercubic Networks

A hypercubic network is generated by an iterative construction technique called *dimensional duplication*, where the basic network is a single module  $H_0$  (indexed by the integer 0) and  $H_j$  is obtained by duplicating  $H_{j-1}$  and connecting pairs of homologous modules indexed  $(i, 2^{j-1} + i)$  for  $i = 0, \dots, 2^{j-1} - 1$ . Obviously,  $H_j$  is a  $j$ -dimensional hypercube, and each of its modules is connected to  $j$  other modules, that is, the module type depends on the parameter  $j$ . This feature violates an assumption of the model (modules independent of network size), so that the hypercube cannot be viewed as a VLSI architecture in the narrow sense.

Module  $j$  stores operand  $T[j]$  and is provided with arithmetic capabilities. Hypercubic networks support the *recursive combination* paradigm, executed by the call ALGORITHM( $0, 2^m - 1$ ) of the following procedure:

```

ALGORITHM($l, l + 2^h - 1$)
if ($h > 1$) then
 ALGORITHM($l, l + 2^{h-1} - 1$),
 ALGORITHM($l + 2^{h-1}, l + 2^h - 1$)
 foreach $l \leq j \leq l + 2^{h-1} - 1$
 pardo ($T[j], T[j + 2^{h-1}]$) \leftarrow OPER($T[j], T[j + 2^{h-1}]$)

```

Specialization of the operation OPER yields algorithms for such diverse applications as merging, Fast-Fourier-Transform, convolution, permutations, integer multiplication, etc. The paradigm handles one hypercube dimension at a time (parameter  $h$ ) in a fixed order, and is also referred to as “normal” and “ASCEND-DESCEND.”

Due to its nonstandard modules the hypercube does not lend itself to scalable VLSI layouts, because for very large size  $n$  the outdegree of the modules may be unrealizable. However, there exist effective emulators of the hypercube which have standard VLSI layouts. Significantly, since the computations considered obey the bound  $AT^2 = \Omega(n^2)$ , these emulators are found to be area-time optimal. Two of these emulators, the linear array and the mesh, have large diameters (respectively,  $O(n)$  and  $O(\sqrt{n})$ ); two more emulators, the shuffle-exchange and the cube-connected cycles, have  $O(\log n)$  diameter. The diameter, of course, determines the computation time.

### Composite Architectures

The previous basic architectures have different diameters and are therefore targeted to different computation times. Interestingly enough, these architectures can be combined in a variety of ways, both to solve different classes of problems and to achieve a full range of computation times. The typical composite structure is a network of modules that are themselves complex VLSI architectures.

To illustrate this philosophy we consider an area-time optimal emulator of the hypercube. We begin with

the emulation of normal algorithms by means of a linear array. The key activity to be emulated is

```

foreach ($0 \leq j \leq 2^{h-1} - 1$)
 pardo ($T[j], T[j + 2^{h-1}]$) \leftarrow OPER($T[j], T[j + 2^{h-1}]$)

```

The operands are stored in the array  $T[0, 2^h - 1]$ . The above operation can be executed in the array if operands  $T[j]$  and  $T[j + 2^{h-1}]$  are brought to reside in adjacent modules. This is achieved by the following algorithm executed on the array:

```

for $s = 0$ to $2^{h-1} - 1$
 foreach $0 \leq i \leq s$
 $U \leftarrow 2^{h-1} - 1 - s$
 pardo ($T[U + 2i], T[U + 2i + 1]$)
 \leftarrow EXCHANGE($T[U + 2i], T[U + 2i + 1]$)

```

The overall time required by these rearrangements in processing the  $h$  dimensions is clearly  $O(n)$ , so the array *per se* is not area-time optimal. However, suppose we partition the  $n$  operands into  $n/k$  sets of size  $k$  and assign each set to a distinct linear array to process the lowest  $\log k$  dimensions as outlined above. In order to process the remaining  $\log n - \log k$  dimensions, these arrays are closed by an end-around edge, and are interconnected as a hypercube (where in each array different dimensions are handled by different modules). The resulting modules have outdegree 4 regardless of  $n$ . The resulting composite architecture, called the *cube-connected cycles*, is an emulator of the hypercube. It runs in time  $T = O(k)$  and can be laid out in area  $O(n^2/k^2)$ , so that it is area-time optimal. Since  $k$  can be chosen between  $\Omega(\log n)$  and  $O(\sqrt{n})$ , this is an explicit example of area-time trade-off.

### A Critical Comment

Research on VLSI computation was prompted by a technological revolution and developed in the 1980s under the shadow of a funding policy that promoted massive parallel computing. A sufficiently simple model became established, unleashing a volume of research that elucidated important relationships between computations and parallel architectures.

As mentioned before, the Achilles' heel of the VLSI model is its second axiom concerning timing. It was

soon realized that with reference to the scalability of VLSI architectures, the finiteness of the speed of light and the inability to indefinitely reduce the size of the devices invalidate the timing axiom. It happens that small-diameter networks (notably trees and hypercubes) have maximum wire length growing nearly as the square-root of the network area. In a technology where the physical limits are approached (*limiting technology*) only meshes are truly scalable architectures.

However, it must be mentioned that as this entry is written (2009), the approaching physical limits have revived interest in parallel computations, although along directions different from massive parallel computing.

## Bibliographic Notes and Further Reading

The origins of the theory of VLSI computation can be roughly traced back to the appearance in the late 1970s of a book by Mead and Conway [1], a somewhat simplified presentation of the technology, which enormously stimulated the algorithmic community. At about the same time, C. Thompson [2] presented the VLSI computation model and the area-time approach, and shortly thereafter J.D. Ullman [3] framed the emerging research area in the language of the theoretical community.

The literature on VLSI computation is enormous. A comprehensive reference is the authoritative text by F. T. Leighton [4], which presents a scholarly compendium of the computational/architectural aspect of VLSI at a time (1992) when research interest was

beginning to wane. The interested reader is encouraged to consult this source.

Suffice it to mention here: systolic networks were introduced by H.T. Kung and C.E. Leiserson [5], the shuffle-exchange network was discussed in [2], and the cube-connected-cycles in [6]. The notion of computational friction was proposed in [7]. The implications of the physical limitations are analyzed in [8].

## Bibliography

1. Mead C, Conway L (1980) Introduction to VLSI systems. Addison-Wesley, Reading, MA
2. Thompson C (1980) A complexity theory for VLSI. Ph.D thesis, C.S. Department, Carnegie-Mellon University
3. Ullman JD (1984) Computational aspects of VLSI. Computer Science Press, Rockville, MD
4. Leighton FT (1992) Introduction to parallel algorithms and architectures: arrays-trees-hypercubes, Morgan Kaufmann Publishers, San Mateo, CA
5. Kung HT, Leiserson CE (1978) Systolic arrays for VLSI. Sparse Matrix Proc. 1978–79, SIAM, pp 256–282
6. Preparata FP, Vuillemin J (1981) The cube-connected-cycles: a versatile network for parallel computation. Commun ACM 24(5):300–309
7. Bilardi G, Preparata FP (1986) Area-time lower-bound techniques with applications to sorting. Algorithmica 1(1):65–91
8. Bilardi G, Preparata FP (1995) Horizons of parallel computation. J Parallel Distrib Comput 27:172–182

---

VNG

►Vampir

# W

## Warp and iWarp

JAMES R. REINDERS  
Intel Corporation, Hillsboro, OR, USA

### Definition

The Warp project was a series of increasingly general-purpose programmable systolic array systems and related software. The project was created by Carnegie Mellon University (CMU) and developed in conjunction with industrial partners G.E., Honeywell, and Intel with funding from the U.S. Defense Advanced Research Projects Agency (DARPA). There were three distinct machine designs known as the WW-Warp, PC-Warp, and iWarp. Each successive design was made increasingly general-purpose by increasing memory capacity and relaxing the tight synchronization between processors. A two-cell prototype of WW-Warp was completed in June 1985, the first WW-Warp in February 1986, the first PC-Warp in April 1987, and the first iWarp system in March 1990. iWarp systems were produced and sold by Intel in 1992 and 1993 to universities, government agencies, and industrial research laboratories. Signal and image processing were the target applications for the Warp program, a natural fit for systolic arrays. The Warp program produced pioneering work on software pipelining support in compilers to utilize the LIW capabilities of the cells (processors).

The name Warp referred to warp factors used by the Star Trek TV series to induce very high velocities in anticipation of the new Star Trek episodes, known as "Star Trek: Next Generation," which appeared in the fall of the year that PC-Warp became operational. iWarp added "i" for *integrated* for the VLSI version and was adopted before Intel was selected as recipient of the contract to design and manufacture it. Intel, at the time, used "i" as a prefix on many product names.

### Discussion

#### Introduction

Systolic Arrays, first described in 1978 by H. T. Kung and Charles E. Leiserson, were originally conceived as a systematic approach to utilize rapid advances in electronics, known as Very-Large-Scale-Integrated (VLSI) technology, and coping with inherent difficulties present in designing VLSI systems. The systolic array concept proved to be successful at taking advantage of this evolution and led also to advancements in software models for utilizing hardware parallelism.

At the same time, instruction-level parallelism was attracting considerable interest for utilizing the rapidly expanding capabilities of VLSI. The use of explicitly scheduled instructions in the form of long instruction word (LIW) design was of interest to H. T. Kung and his team.

In 1983, H. T. Kung, at Carnegie Mellon University, started the Warp program with a desire to investigate the many hardware, software, and application aspects of systolic array processing. Over the course of a following decade, the program yielded research results, publications, and advancements in general-purpose systolic hardware design, compiler design, and systolic software algorithms.

At first, Warp was conceived as a limited prototype to be used by the vision group at CMU for low-level vision operations that were simple and highly regular (1D or 2D convolutions, edge detection operators, etc.). As work progressed, opportunities for more complex uses for Warp machines were proposed.

As the Warp design came together, the Strategic Computing Initiative (SCI) was looking for high-performance computing platforms to tackle specific problems, one of which was the development of an Autonomous Land Vehicle (ALV). It was decided to place the CMU prototype Warp machine (two cells) onboard the CMU ALV. The ALV program would need more than one Warp, and H. T. Kung decided to pursue

funding from the SCI program. Through the support of DARPA, and working with industrial partners, a series of three Warp machines were produced over the span of a decade. Since using VLSI was the inspiration for the systolic array concept, it was natural for H. T. Kung to guide the Warp project from an idea, to a wire-wrapped prototype, to a printed circuit card implementation, and finally to a VLSI implementation. In 1986, while the first wire-wrapped prototype of a 10-cell machine was being delivered and the printed circuit version was being designed, Intel was selected to be the industrial partner for the integrated circuit implementation of Warp.

Consistent with the desire to harness leaps in VLSI and heavily influenced by a deep interest in signal and image processing, the use of LIW in the hardware and software was a key area of investigation as well. The Warp program focused specifically on developing a compiler scheduling technique of software pipelining as a key technique to utilize LIW. This contrasted with work by Joseph (Josh) Fisher at Yale University and Multiflow Computer Inc., which focused on developing a compiler scheduling technique of trace scheduling to utilize LIW. Neither effort resulted in hardware that benefited significantly from employing both compiler scheduling techniques together. Later, the Itanium project at HP and Intel included engineers formerly of both the Multiflow and the Warp projects. The Itanium design included hardware designed to be able to utilize both trace scheduling and software pipelining.

Complementing the development of the machines, software research and development was a big part of the Warp program. Advances in compiler techniques for software pipelining, as well as advances in systolic programming and programming models, were made as part of the Warp program.

All Warp and iWarp machines were Systolic Arrays, so they consisted of a set of interconnected cells, each capable of performing at least simple computational operations and communicating only with cells in close proximity. Warp machines were simple linear configurations, while iWarp machines were organized as 2-D arrays with the edges wrapping around to form a torus. In both cases, programming could generally be described in terms of streaming data through the array for processing as it passed through cell. In the simplest example, every cell could be programmed to read a data element, add one to it, and send the result along

to the next cell. Such a program on any of the Warp systems would be written roughly as `sendf(receivef()) + 1.0` and compile to a single instruction that executed in a single cycle, thereby highlighting the tight coupling of computation and communication. A stream of data sent through a machine thus programmed would emerge with every element of the stream incremented by N, where N was the number of cells in the machine. Many systolic algorithms have been developed to take advantage of the parallelism of such a system.

The Warp program focused on practical realizations of systolic array concepts in order to facilitate advanced software research on real-world applications of the machine. This started when the first hand-built two-cell Warp machine, which was designed to solve vision problems, was quickly pressed into service in a real-world challenge in the CMU ALV. This early focus on vision and signal processing, coupled with a desire to solve real problems for researchers outside computer architecture research, lead to four defining characteristics for the Warp program.

## **1. Systolic communications: Efficient fine-grained communications.**

Internode communication is the most fundamental and key issue in the design of a parallel computer because the communication system directly impacts the mapping of an application onto the computer. The exploration of “balanced algorithms” at CMU, led by H. T. Kung starting in the mid-1970s, looked to find algorithms that scale without bounds. Communication with little to no overhead is needed to efficiently deal with the fine-grained parallelism that such algorithms tend to utilize as machine sizes grow. Systolic arrays accomplish this goal by providing a method to directly couple computation to communication. This was true in Warp and iWarp machines. This principle is an important consideration in “balanced” system design to allow high-performance systems to have applicability on a broader set of algorithms.

## **2. Systolic design advantages coupled with general programmability.**

The systolic array approach initially led to very rigidly synchronous hardware designs that were elegant yet proved overly constraining for many programming solutions. Each successive generation of

the Warp program offered increased programmability by relaxing prior rigid synchronization design constraints and increasing cell autonomy by changes including notable increases in cell memory capacity.

### 3. Long instruction word (LIW) designs with optimizing compilers.

The cells of all Warp machines utilized LIW (or VLIW, Very Long Instruction Word) and invested in compiler technology to utilize the LIW capabilities, including extensive research and publishing focused on developing software pipelining techniques.

### 4. Early prototypes for software developers.

While inspired by VLSI opportunities, the Warp project focused on developing early prototype systems using a low-risk approach of combining MSI and LSI parts in a relatively conservative design instead of an immediate leap to VLSI. This led to functional hardware that provided a reliable platform for rich software investigations early in the program. Learnings from the prototypes were then implemented into a VLSI version that in turn could draw on the benefits of several years of software development on the early prototypes.

## The Warp Machines

Warp and iWarp machines were a series of increasingly general-purpose systolic array processors, created by Carnegie Mellon University (CMU), in conjunction with industrial partners G.E. (Syracuse, New York), Honeywell (Minneapolis, Minnesota) and Intel (Hillsboro, Oregon), and funded by the U.S. Defense Advanced Research Projects Agency (DARPA).

There were three distinct machine designs known as the WW-Warp (Wire Wrap Warp), PC-Warp (Printed Circuit Warp), and iWarp (integrated circuit Warp, conveniently also a play on the “i” for Intel).

Each successive generation became increasingly general-purpose by increasing memory capacity and loosening the coupling between processors. Only the original WW-Warp forced a truly lock step sequencing of stages, which severely restricted its programmability but was in a sense the purest “systolic array” design.

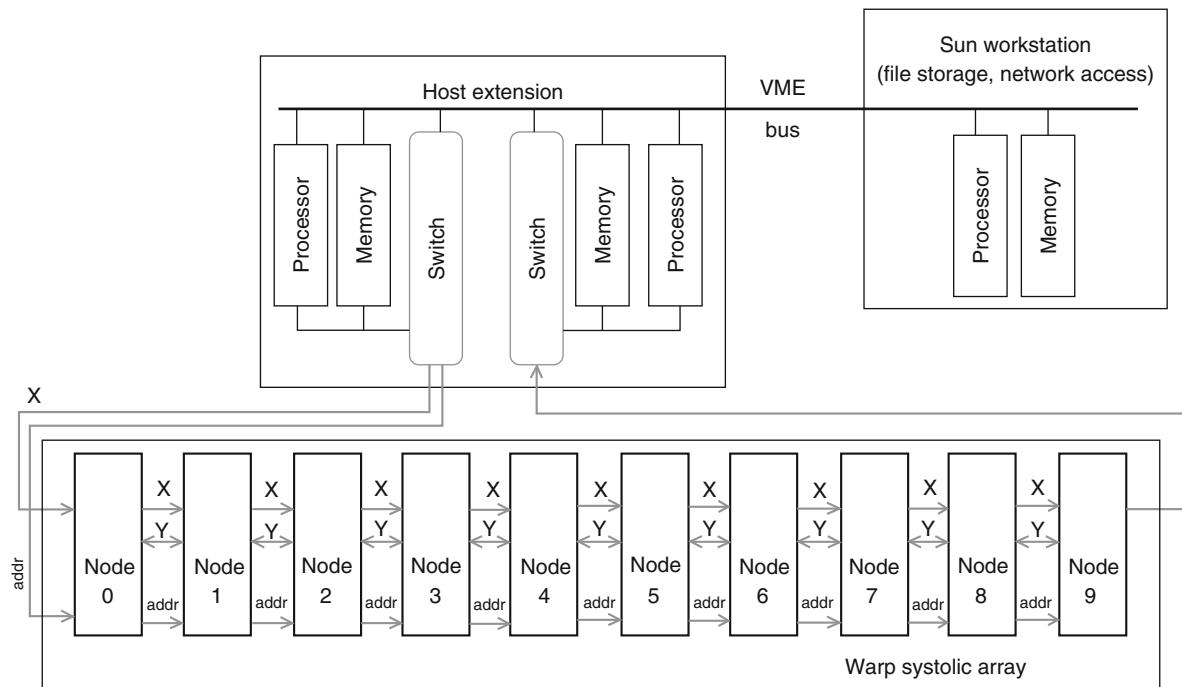
Warp machines were attached to Sun workstations (UNIX based). Software development work for all models of Warp machines were done on Sun workstations (Figs. 1 and 2).

The WW-Warp and PC-Warp machines were systolic array computers with a linear array generally of ten or more cells, each of which is a programmable processor capable of performing ten million single-precision floating-point operations per second (10 MFLOPS). A ten-cell machine had a peak performance of 100 MFLOPS. The iWarp machines doubled this performance, delivering 20 MFLOPS of single precision and supporting a double-precision floating point at half that performance.

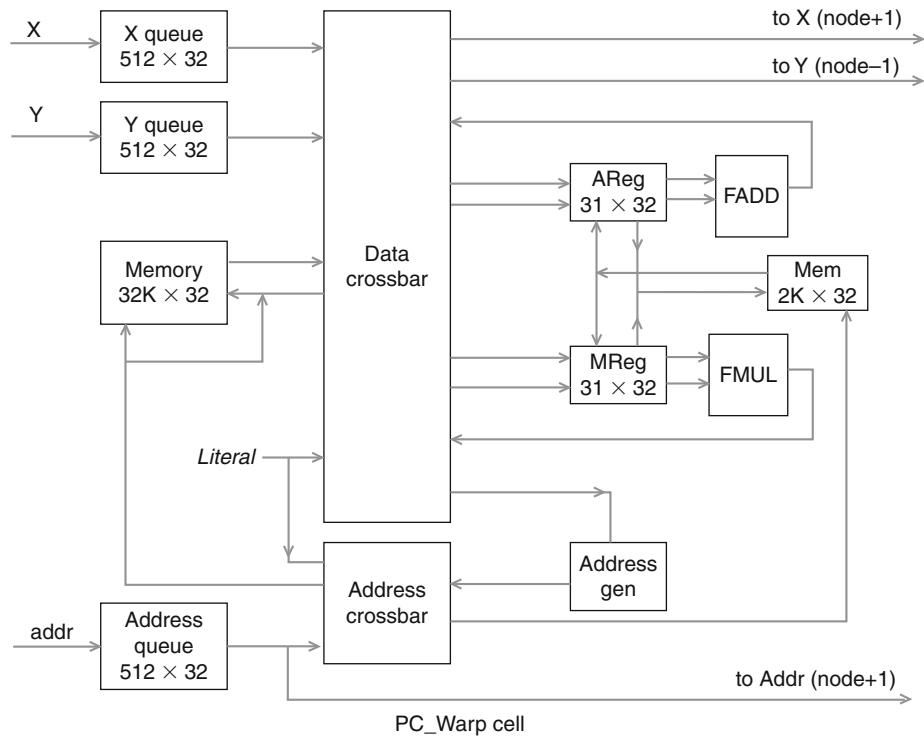
A two-cell prototype of WW-Warp was completed at Carnegie Mellon in June 1985. Two essentially identical WW-Warp machines, each with ten identical boards (nodes) plus an address generator I/O board, were produced in 1986. The system from G.E. was delivered in February 1986; the system from Honeywell was delivered in June 1986. G.E. became the sole industrial partner for the PC-Warp machine construction. Significant redesign work was done to generalize the systolic array before production of the PC-Warp. The first PC-Warp was delivered by G.E. in April 1987. About twenty production machines of the PC-Warp were produced and sold by G.E. during 1987–1988. The PC-Warp machine sold for \$350,000 without the Sun workstation. Each cell (node board) in a PC-Warp consumed about 100 W of power.

In 1986, Intel was selected, as a result of competitive bidding, to be the industrial partner for the integrated circuit implementation of Warp. The first iWarp system, a twelve-node system, became operational in March 1990. Additional steppings of the chip resulted in the final C-Step part that improved the clock rate, beyond the 16 MHz project target, to 20 MHz and worked out most errata. About 39 machines, with more than 1,500 processors combined, were produced and sold by Intel in 1992 and 1993 to universities, government agencies, and industrial research laboratories. The largest system assembled was a 512-processor machine, and one 256-processor machine was made for CMU. Typical production systems were 64-processor systems. The high-speed static memory and the high-performance low-latency communication system proved to be a well-suited target for research efforts and many “proof-of-concept” applications.

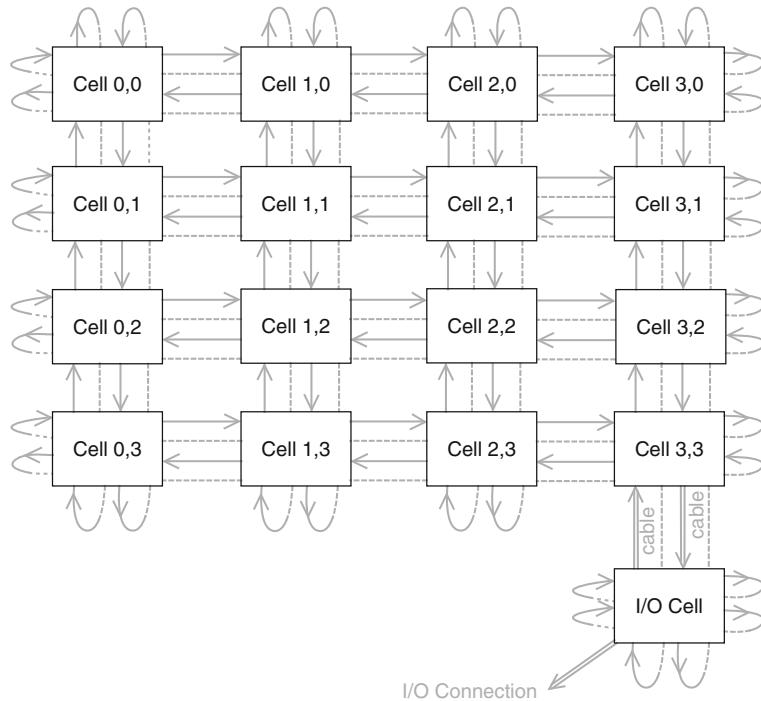
The iWarp machines were based on a full-custom VLSI component integrating a 700,000 transistor LIW microprocessor, a network interface and a switching



**Warp and iWarp. Fig. 1** PC-Warp system architecture



**Warp and iWarp. Fig. 2** PC-Warp cell architecture

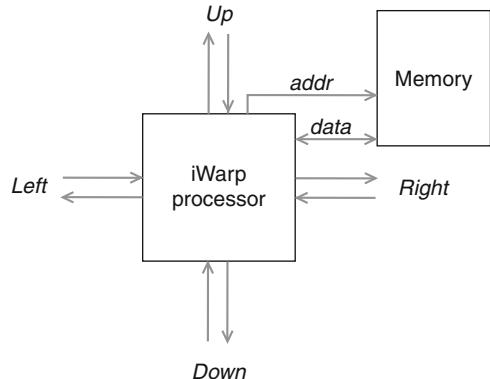


**Warp and iWarp. Fig. 3** iWarp system architecture

node, into one single chip of  $1.2\text{ cm} \times 1.2\text{ cm}$  silicon. The processor dissipated up to 15 W and was packaged in a ceramic pin grid array with 280 pins. Intel marketed the iWarp with the tagline “Building Blocks for GigaFLOPs.”

The iWarp processor was designed specifically for the Warp project, utilized LIW format instructions, and featured tightly integrated communications with the computational processor. The standard iWarp machines configuration arranged iWarp nodes in a  $2m \times 2n$  torus. All iWarp machines included the “back edges” and, therefore, were tori (Figs. 3 and 4).

Programmability increased in each successive generation within the Warp program. In the WW-Warp design, all addresses for data accesses were created on the interface cell, which preceded the first computational cell, and passed along as data on the communication channels. Each WW-Warp cell was technically an independent SIMD machine with local instructions, but the whole design had an underlying assumption that cells would be identically programmed in a very rigid Single-Instruction-Multiple-Data (SIMD) fashion. The lack of ability to locally create data access addresses,



**Warp and iWarp. Fig. 4** iWarp cell architecture

in particular, helped ensure such rigid programming. The SIMD program preceded as a wave through the machine in which cells were skewed in their execution, as opposed to operating in a global lock step, as data and addresses streamed through the machine. The tight coupling of the cells meant that if a cell sent data into a full input queue, the data was lost. This tight coupling made it impossible for the compiler to support a pipeline programming mode in practice.

The very rigid clocking of the WW-Warp was relaxed in PC-Warp by the increasing intracell buffering, introducing intracell flow control, as well as the addition of address generation units on individual cells for local address generation capabilities. With added hardware flow control and a local controller, cells were no longer tightly coupled and could operate independently. Direct coupling of neighboring cells on PC-Warp still had no allowance for data to bypass the computational unit of a cell. Only a programmed operation could move data through a PC-Warp cell because of the direct coupling of communication to the functional units in the cell. iWarp introduced a communication agent on the cell to allow for data to pass through a cell without the need to interact with the computational units.

The design of iWarp was oriented toward preserving the benefits of systolic communication while generalizing the framework to allow for more application diversity. iWarp maintained the systolic feature of connections for low overhead communication, while providing for a broad range of applications by making those connections programmable. In iWarp, physical communication channels were augmented with virtual communications channels that allowed logical communication channels to be established over the same physical hardware to add great flexibility in the programming models.

The iWarp processor was a 32-bit Reduced-Instruction-Set-Computing (RISC) design with a 96-bit LIW instruction that, unlike the Intel i860 processor, had no performance penalty for transitioning between long and short instructions in an instruction stream. The final, C-stepping, of the processor was clocked at 20 MHz. The processor had 20 MIPS integer performance, supported full IEEE floating-point compliance for 32- and 64-bit operations with performance of 20 MFLOPs and ten MFLOPs, respectively. Processors were grouped four on a double-height Eurocard PC board (23 cm × 28 cm). Sixteen of these quad-processor boards fit in a 19-inch rack along with a clock board, and four racks could be placed in a single modified iPSC cabinet (156.5 cm × 53.5 cm × 65 cm deep). Up to four cabinets could be connected together.

WW-Warp and PC-Warp utilized their long instructions to route communication inputs directly to arithmetic units, and outputs of arithmetic units directly to

the communication outputs for a node. In iWarp, some of the register file was mapped to communication so that a communications input from a neighbor was represented as a register read, and a register write sent data to a neighbor. In all these machines, the communications were matched precisely to the computational capabilities of the machine.

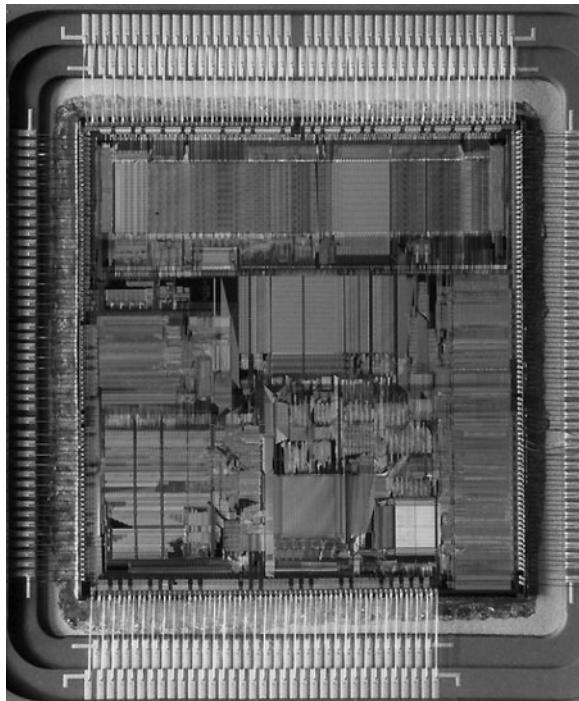
The interconnect capabilities of an iWarp processor were through four 20 MB/s full duplex links. The hardware supported up to 20 logical (virtual) channels that were freely configurable across all four directions and used two pools. Communications were clocked at 40 MHz.

Each successive generation of the Warp program also increased the available memory on each cell. The original WW-Warp design had a very limited 2K program memory but also had little use for large memory modules because the rigidly synchronous design did not encourage use of local storage nor large programs. The relaxation of the rigid connections led to algorithms that benefitted from more local storage so the PC-Warp design had 8X the program store and more memory on each cell. Further relaxing in turn demanded more memory, and the iWarp design initially offered high performance with very limited static memory modules. The creation of a daughter card supporting 16 Mb of dynamic memory greatly increased available memory, for both the application and the resident operating code, and proved very popular. Memory access latency was 100 ns static and 200 ns dynamic. Memory access bandwidth was 160 MB/s. The iWarp processor included eight programmable direct memory access (DMA) agents for data spooling to communication channels.

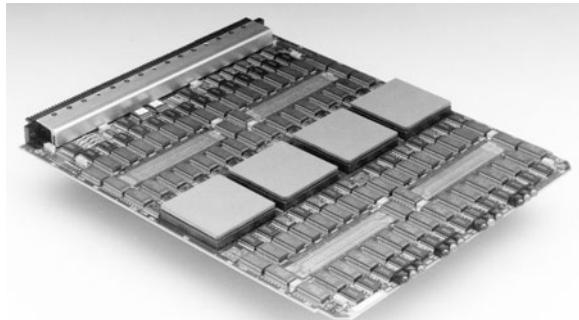
Systems built from iWarp processors could range from four iWarp processors to 1,024 processors in a  $2n \times 2m$  torus. The typical system was 64 cells in an  $8 \times 8$  torus for a total 1.2 GFlop/s peak performance. Measured sustained performance of 64 cells for dense matrix multiply was 1150 MFlop/s, sparse matrix multiply was 400 MFlop/s, and FFT was 700 MFlop/s ([Photos 1–4](#)).

## Compilers

Multiple compilers associated with the Warp program explored exploitation of both cell programming and array level programming.

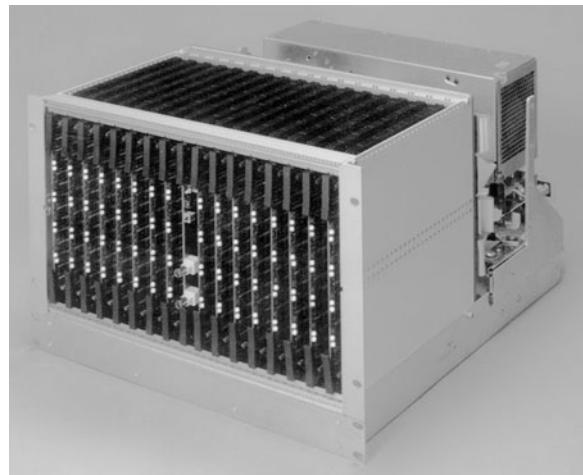


**Warp and iWarp. Photo 1** The iWarp processor, die exposed



**Warp and iWarp. Photo 2** The iWarp quad-cell board (Four processor, memory, communication)

The cells of all Warp machines utilized LIW (or VLIW, very long instruction word). The WW-Warp and PC-Warp used VLIW to directly send commands to every unit on the board in every cycle, using the VLIW 272-bit wide instructions. The iWarp processor utilized a 96-bit LIW instruction with several possible encoding formats to control integer and floating-point operations in parallel. For instruction compaction, iWarp allowed



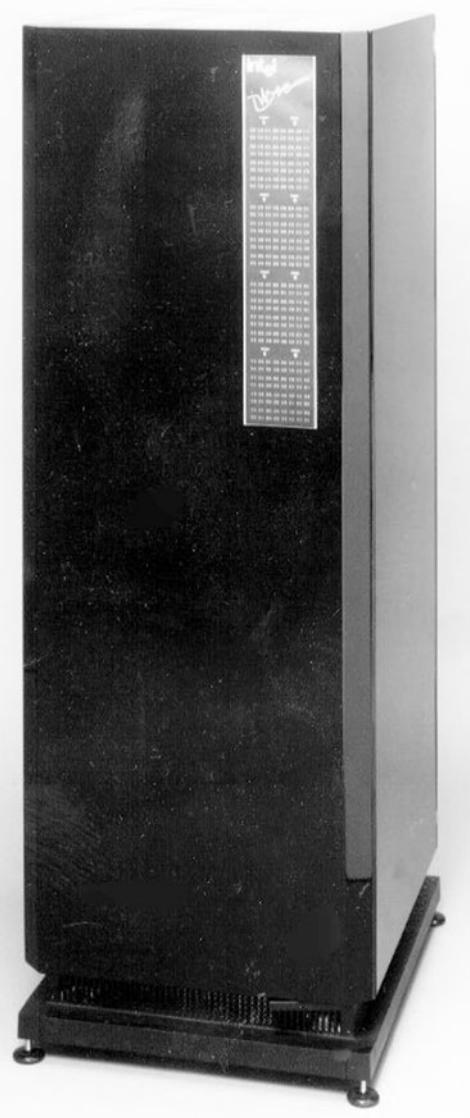
**Warp and iWarp. Photo 3** Sixty-four iWarp cells mounted in a 19-inch rack (1.2 GFLOPs)

32-bit RISC instructions to be freely mixed with the 96-bit LIW instructions. The Warp program invested in compiler technology to utilize the LIW capabilities, including extensive research and publishing on software pipelining techniques.

A research compiler, for a language known as "W2," targeted all three machines and was the only compiler for the WW-Warp and PC-Warp, while it served as an early compiler during development of the iWarp. The production compiler for iWarp was a C and Fortran compiler based on the AT&T pcc2 compiler for UNIX with extensive modifications and extensions for systolic programming and exploitation of LIW via software pipelining.

The W2 compiler was written in Common LISP and ran on Sun Workstations. The W2 language was a restrictive procedural language similar in syntax to Pascal or C, and designed to specify a kernel of computation to be spread across the computation array as well as specify the data input and output sufficiently for the compiler to generate the code to feed the input stream and to recover the output stream.

The initial focus was on producing code for both the interface cells and the computational cells of the WW-Warp machine. The PC-Warp allowed more general programming by providing for local address generation that permitted a more general programming of each cell so the W2 language and compiler evolved to support



**Warp and iWarp. Photo 4** The iWarp system cabinet containing four racks (256 cells)

more general programming. Once operational with creating the proper control code for the machine, the W2 compiler focus took on software pipelining to fully utilize the combination of tightly coupled communication capabilities with the LIW instruction design. Monica Lam's Ph.D. work "A Systolic Array Optimizing Compiler" covers the results in detail, including methods to software pipeline loops to any depth of nesting. The W2 compiler for PC-Warp was taken on by G.E. in 1987,

and CMU focused work on a port of the W2 compiler to iWarp.

It was initially believed that programs for the W2 compiler would typically be "half a page" in length. This was consistent with systolic arrays up to that point. The generalization of PC-Warp saw programs 20X the originally anticipated norm, and automatic program generation systems stretched this even further. Developers of applications for the PC-Warp often requested a more robust compiler for a full and standard language.

The production compiler for iWarp was a C and Fortran compiler based on the AT&T pcc2 compiler for UNIX. HCR Corporation of Toronto, under contract to Intel, provided the initial port of pcc2 for iWarp along with HCR's own global optimizer technology for pcc2. The compilers went on to be extensively modified and extended by Intel and CMU, and were mostly used by developers directly. However, they were also targeted by special frontends, including cfront (for C++), oxygen compiler (for a variant of Fortran77), and the Fx compiler (for a variant of HPF).

Several research compilers also produced code for iWarp, including the retargeted W2 compiler, a code generator for cmcc (a C compiler for research into optimizations at CMU), and a backend for the Stanford SUIF compiler (under the guidance of Prof. Monica Lam).

With the advent of High Performance Fortran (HPF) research, CMU's Fx project was born to explore HPF style investigations on the iWarp platform. Significant energy at CMU in compiler research went into the Fx compiler.

Domain-specific or "application" compilers were an area of interest at CMU. Parallel program generators for iWarp were directed toward specific application domains and included the Apply project (for image processing) and the AL project (for scientific computing).

## Applications

One Warp system was installed in the NavLab, a GM van converted by researchers in the Robotics Institute at Carnegie Mellon University. The NavLab was a laboratory for research in autonomous road following, and a Warp machine (as well as a number of workstations) was installed in the van. The Warp machine turned out to be a good engine for neural net learning. It also performed a variety of other vision and planning

tasks. A Warp machine remained in the NavLab until 1989, when the van caught fire after the air conditioning system leaked liquid onto the computers.

Several NSF grand challenge problems were investigated.

An iWarp machine had successful sea trials on a U.S. Navy submarine.

### **Warp's Influence on the Industry**

The W2 compiler pioneered software pipelining for LIW. The Warp compiler projects provided extensive investigations into software pipelining, including nested software pipelining, and published solutions. These have become an important and popular compiler optimization for most modern computers.

After iWarp, Intel continued work on interconnect chips derived from the iWarp design. The world's first TeraFLOP computer, ASCI Red built for Sandia National Labs by Intel, utilized a variant of the logical channels developed first for iWarp.

iWarp was the first Intel project that utilized workstations for simulations and mask generation, and worked to move Intel's design tools into this environment for the first time. Future generations of Intel chip designs moved away from mainframes following the lead of the iWarp project.

### **Related Entries**

- ▶ [Systolic Arrays](#)
- ▶ [VLIW Processors](#)

### **Bibliographic Notes and Further Reading**

The first papers mentioning Warp that appeared in 1984 focused on machine vision and not the machine. Papers specifically about the Warp machines appeared in 1986 as the first machines were delivered with a pretty complete overview in 1987 [1]. A retrospective on Warp provides insights into the key lessons of Warp [2]. Monica Lam's May 1987 Ph.D. dissertation focused largely on the pioneering work on software pipelining and was later published as a book [3]. A detailed book on iWarp and the experiences of the project from the perspective of two CMU researchers was published in 1998 [8].

### **Bibliography**

1. Annaratone M, Arnould E, Gross T, Kung HT, Lam M, Menzilcioglu O, Webb J (1987) The warp computer: architecture, implementation and performance. *IEEE Trans Comput C-36*(12): 1523–1538
2. Gross T, Lam M (1998) Retrospective: a retrospective on the warp machines. In: ISCA '98: 25 years of the international symposia on computer architecture (selected papers). IEEE, Los Alamitos, pp 42–45
3. Lam MS (1989) A systolic array optimizing compiler. Kluwer Academic, Dordrecht
4. Borkar S, Cohn R, Cox G, Gleason S, Gross T (1988) iWarp: an integrated solution of high-speed parallel computing. In: Proceedings of the 1988 ACM/IEEE conference on supercomputing, 12–17 Nov 1988, Orlando, Florida, pp 330–339
5. Adl-Tabatabai A-R, Gross T, Lueh G-Y, Reinders J (1993) Modeling Instruction-Level Parallelism for Software Pipelining. In: Proceedings of the IFIP WG10.3 working conference on architectures and compilation techniques for fine and medium grain parallelism, Orlando, pp 321–330
6. Borkar S, Cohn R, Cox G, Gross T, Kung HT, Lam M, Levine M, Moore B, Moore W, Peterson C, Susman J, Sutton J, Urbanski J, Webb J (1990) Supporting systolic and memory communication in iWarp. In: Proceedings of the 17th annual international symposium on computer architecture, Seattle, Washington, pp 70–81
7. Intel Corp (1991) iWarp microprocessor (Part Number 318153). Technical information, Order Number 281006, Hillsboro, OR
8. Gross T, O'Hallaron DR (1998) iWarp: anatomy of a parallel computing system. MIT, Cambridge, MA, p 488

### **Wavefront Arrays**

- ▶ [Systolic Arrays](#)

### **Weak Scaling**

- ▶ [Gustafson's Law](#)

### **Whole Program Analysis**

- ▶ [FORGE](#)

**Work-Depth Model**

► [Models of Computation, Theoretical](#)

**Workflow Scheduling**

► [Task Graph Scheduling](#)

# Z

## Z-Level Programming Language

► [ZPL](#)

## ZPL

BRADFORD CHAMBERLAIN  
Cray Inc., Seattle, WA, USA

### Synonyms

[Z-level programming language](#)

### Definition

ZPL is a parallel programming language that was developed at the University of Washington between 1992 and 2005. ZPL was a contemporary of High Performance Fortran (HPF), targeting a similar class of applications by supporting data parallel computations via operations on global-view arrays distributed between a set of distinct processor memories. ZPL distinguished itself from HPF by providing a less ambiguous execution model, support for first-class index sets, and a syntactic performance model that supported the ability to trivially identify and reason about communication.

### Discussion

#### Foundations

ZPL was initially developed by Lawrence Snyder and Calvin Lin at the University of Washington who strove to design a parallel programming language from first principles. To this end, the ZPL effort began by identifying abstract machine and programming models that would serve as the parallel equivalents of the von Neumann machine model and the imperative-procedural programming model used in traditional sequential computing.

For a machine model, ZPL built upon Snyder's CTA or *Candidate Type Architecture*, defined as "a finite set

of sequential computers connected in a fixed, bounded degree graph, with a global controller." [13]. In contrast to theoretical models like the PRAM, the CTA was designed to serve as a more practical machine model for realistic parallel architectures. At the same time, the CTA was intentionally vague about the parameterization of the target architecture's capabilities due to the high degree of variation from one parallel system to another, combined with the fact that the complexity in such models often failed to add value to an end-user. The main point of the CTA was that locality matters: remote accesses are more expensive than local accesses and therefore need to be avoided whenever possible.

In terms of its programming model, ZPL built on Lin et al.'s *Phase Abstractions* programming model [12] which defined three levels of specification for parallel programming, named the X, Y, and Z levels. The phase abstractions define the *X level* as representing Multiple-Instruction Multiple-Data (MIMD) programming in which the programmer is writing code from the point of view of a single sequential processor. The *Y level* deals with logical topology and specifies the virtual communication channels across which the processors communicate and interact with one another. Finally the *Z level* describes the high-level data parallel computations in which all of the processors execute similar instructions to cooperatively implement a single logical operation.

It is this final level that gave ZPL its full – though in practice, rarely used – name: the *Z-level Programming Language*. ZPL was originally designed to be the data parallel component of a broader language named Orca C that spanned the X, Y, and Z levels defined by the phase abstractions. However, as ZPL matured, rather than making it part of a larger sublanguage, it was extended to support increasingly rich modes of parallel computation, ultimately including "X level" per-processor computation in its final years [9].

Theoretical foundations aside, from the programmer's view, traditional ZPL computations provided a conceptual single-threaded model of computation in

which each statement could be thought of as executing to completion before the next began. In reality, the data parallel statements resulted in concurrent execution across all processors in a loosely coupled manner, but the typical programmer could be completely unaware of such details apart from the performance benefits that resulted.

In terms of its implementation, the ZPL compiler generated a single-threaded Single-Program Multiple-Data (SPMD) executable that implemented the computation and communication required to realize the user's data parallel program. The ZPL compiler took a source-to-source compilation approach for the purposes of portability, translating the user's source down to C code that would then be compiled using a platform-specific C compiler. Communication was implemented in terms of a novel API called *the Ironman interface* [4] that separated the semantics of what data needed to be communicated, and when, from architecture-specific capabilities like shared memory, two-sided message passing, one-sided puts and gets, or even distinct communication threads potentially running on coprocessors. The result was a highly portable compiler that could often match the performance of a hand-coded MPI program and even outperform it on architectures that preferred a different style of communication than two-sided message passing [1, 2, 9].

### Primary Concepts: Regions, Arrays, and Directions

Perhaps the key feature of ZPL was its choice to support the notion of a distributed index set using a first-class language concept known as the *region*. Index sets are important for data parallel programming due to their role in defining data structures and iteration spaces. The distribution of these index sets between distinct processors plays a crucial role in dividing parallel work between compute resources while also establishing clear realms of locality. Most traditional languages have failed to provide abstractions for index sets, requiring the programmer to express them using collections of scalar values for each processor and array dimension. In contrast, ZPL's regions support distributed index sets as a high-level language concept. From ZPL's early stages, regions could be named and initialized. Over time they were extended to support assignment and to serve as function arguments while also being made more com-

patible with other types – for instance by supporting the ability to create arrays of regions or region fields within data structures.

As a sample use of regions, the following code declares an  $m \times n$  region and then uses it to declare an array of floating point values:

```
region R = [1..m, 1..n];
var A: [R] float;
```

In addition to declaring arrays, regions also serve the purpose of scoping statements or groups of statements to provide the indices over which array expressions should be evaluated. For example, the following assignments to *A* are prefixed by region scopes that control which elements of arrays *A* and *Index2* are referenced. The initial region scope causes the first assignment to zero out the whole array *A* since it was declared over region *R*; the next restricts the second assignment to *A*'s *j*th row, causing those elements to take on the value of their column index, represented by the built-in virtual array *Index2*:

```
[R] A := 0;
[j, 1..n] A := Index2;
```

Another early ZPL concept was the *direction*, which was essentially a named offset vector. Like regions, directions were initially limited to named constants, but like regions they became increasingly general and first-class as the language progressed.

Region bounds were often defined in terms of *configuration variables* – variables which are given an initial value that can be overridden on the executable's command-line via compiler-generated argument parsing hooks. In addition to explicitly declaring regions in terms of their upper and lower bounds as in the example above, programmers could also create regions by applying a series of *prepositional operators* to existing regions. These operators typically combined the base region with a direction to create a new region. For example, the *of* operator creates regions that describe boundary conditions by using the offset vector to define a region that borders the base region. In contrast, the *by* operator uses its direction to compute a strided subset of its base region.

Using regions, arrays, and directions, a programmer could fairly easily write simple data parallel computations such as the following Jacobi iteration which iteratively replaces an array's elements with the average of its

```

1 program jacobi;

3 config var
4 n : integer = 1000; -- problem size
5 epsilon : float = 0.0001; -- convergence factor

7 region
8 R = [1..n, 1..n]; -- index set for problem
9 BigR = [0..n+1, 0..n+1]; -- same, but with borders

11 direction
12 north = [-1, 0]; -- cardinal directions
13 east = [0, 1];
14 south = [1, 0];
15 west = [0, -1];

17 procedure jacobi();
18 var
19 A, Temp : [BigR] float; -- compute arrays
20 delta : float; -- default region scope
21 [R] begin

23 A := 0.0; -- initialization of array interior
24 [north of R] A := 0.0; -- and boundary conditions
25 [east of R] A := 0.0;
26 [west of R] A := 0.0;
27 [south of R] A := 1.0;

29 repeat -- main loop
30 Temp := (A@north + A@east + A@south + A@west) / 4.0;
31 delta := max<< abs(A-Temp); -- compute convergence
32 A := Temp;
33 until delta < epsilon;

35 writeln(A);
36 end;

```

nearest neighbors. This continues until the largest difference between elements across any two iterations is less than a convergence value, *epsilon*:

## Array Operators and the WYSIWYG Performance Model

In order to support more interesting computations than purely element-wise operations, ZPL supports a number of *array operators* that alter the way an array reference's enclosing region scope provides its indices. For example, the *@-operator*, seen in line 30 of the Jacobi example above translates the region's index set by the given offset before applying it to the array. Thus,  $A@\text{north}$  refers to the index set  $[0..n-1, 1..n]$

rather than the default  $[1..n, 1..n]$  as specified by the enclosing region scope  $[R]$  on line 21. Similarly, the reduction operator ( $<<$ ) used in line 31 collapses values specified by that same region scope  $[R]$  by generating the maximum value computed by the *promotion* of the scalar  $\text{abs}()$  function and subtraction operator across array elements.

Other primary ZPL operators include the flood operator ( $>>$ ) which replicates subarrays of values, the scan operator ( $||$ ) which computes parallel prefix operations, and the remap operator (#) which supports bulk random access to an array's elements using whole-array indexing expressions. In addition, ZPL supports *wrap* and *reflect* operators that support common boundary

conditions on regular grids and a variation on the @-operator to support loop-carried array references [6].

ZPL's semantics require that any two array references within a single statement must be distributed identically, ensuring that like-indexed elements are colocated within a single processor's memory (the remap operator is the one exception to this rule for reasons that we will return to in a few paragraphs). The impact of this rule is that all communication within a ZPL program can be identified and classified by the presence of its array operators.

In particular, if a statement contains no array operators, since all of its array expressions are described by the same region scope and aligned, it will be implemented without communication, resulting in an embarrassingly parallel execution (or perhaps “pleasantly parallel”). In contrast, the application of an array operator implies that the enclosing region scope's index set is transformed in some way for that array reference, requiring communication to bring the array's elements into alignment with the enclosing region scope's index set. For example, since the @-operator implies a translation of the region scope's indices, a translation of array elements in the opposite direction is required to align the indices with those described by the enclosing region scope.

In addition to signaling the need for communication, each array operator also implies a particular style of communication for its array's elements. For example, given ZPL's default block-distributed regions, the @-operator implies nearest neighbor point-to-point communication, while the flood and reduction operators imply broadcasts or reductions of array values across the dimensions of the virtual processor grid for which the operation is being computed.

Because the remap operator can express an arbitrary shuffling of array values, it potentially requires all-to-all communication to bring elements into alignment with the enclosing region scope [10]. This is the reason that an array expression indexed by the remap operator is exempt from the requirement that it be aligned with other arrays in the statement: since remaps can result in arbitrary communication patterns, it hardly matters whether the array argument is aligned with the statement's region scope or not. To this end, remap serves as the operator for bringing distinctly distributed arrays into alignment or for performing operations across arrays of differing rank.

The impact of these semantics was profound, since it permitted ZPL programmers to trivially and visually identify communication within their programs, to classify that communication, and to approximate its cost for a given architecture. As a result, the development team called this property ZPL's *What You See is What You Get* or *WYSIWYG performance model* [3]. Equally important, this syntactic identification of communication also simplified the job of implementing ZPL by making the compiler analysis required to identify and insert communication trivial. For this reason, the WYSIWYG performance model probably served as the biggest differentiator between ZPL and HPF, whose language definition made communication impossible for the programmer to reason about in a portable manner, either syntactically or semantically. Moreover, HPF compilers tended to require complex index analyses to identify common patterns like the vectorizable nearest-neighbor communications that were clearly identified in ZPL via the presence of an @-operator.

## Advanced ZPL

By the late 1990s, the original scope of ZPL had largely been implemented, yet the language remained too restricted to take on more interesting parallel computations such as those represented by the NAS Parallel Benchmarks (NPB) or various scientific applications being undertaken in other departments at the University of Washington. To this end, the ZPL team began working on evolving ZPL, referring to the original feature set as *classic ZPL* and the new extensions as *advanced ZPL*. Over time, the classic versus advanced distinction faded and the name ZPL was ultimately used to describe the full set of features developed within the project.

Several of the features attributed to advanced ZPL stemmed from the evolution of the region from a simple named constant to a first-class language concept, as mentioned previously. In particular, the ability to modify a region's index set and to create arrays of regions supported the ability to describe sparse and hierarchical index sets and arrays [1, 5, 7]. Other changes involved adding the concept of processor-specific values and array dimensions to the language, permitting the user to break out of ZPL's traditional single-threaded data parallel programming model and write computations on a processor-by-processor basis. Another extension promoted the representation of the virtual processor

grid to a first-class concept, permitting arrays to be redistributed by reshaping the set of processors to which they were mapped. This led to a proposed taxonomy of distributions to extend ZPL's traditional support for block-decomposed regions [9].

## Status, Evaluation, and Influence

Several versions of ZPL were released to the public in a binary format during the late 1990s. Thereafter, ZPL was released as open source under the MIT license with the final release occurring in April of 2004. Shortly thereafter, Steve Deitz – the last member of the ZPL team – graduated, at which point the project went on permanent hiatus. At the time of this writing, ZPL remains available for download, but is not actively being developed or supported.

ZPL had a number of limitations that prevented it from ever being broadly adopted. Perhaps chief among these was its support for only a single level of data parallelism (and toward the end, a form of SPMD task parallelism). While this represents a very important common case in parallel computing, real-world applications typically have several different types of parallelism composed in various ways that are richer than ZPL's restricted data parallelism was able to express. Another drawback to ZPL was its use of distinct array types and operations to refer to data parallel computations vs. local, serial array computations. This distinction was crucial to its WYSIWYG performance model but ultimately resulted in a fair amount of frustration and confusion among users, particularly when the same logical computation would have to be written multiple times using distinct concepts to support both distributed and non-distributed arrays. The resulting failure to reuse code was well justified, but frustrating in practice. Finally, ZPL's decision to focus on data parallelism to the exclusion of modern language concepts like object-oriented programming or even modern syntax presented challenges to adoption by a broader or more mainstream programming community.

None of this is to suggest that ZPL was an unsuccessful language. On the contrary, in the (admittedly biased) opinion of this author, it represents the model of a successful academic project in that it picked its target – namely, data parallelism supporting the ability to reason about communication and locality – and focused on it to the exclusion of non-research distractions that were clearly orthogonal and could be dealt

with later, such as the aforementioned lack of object-oriented capabilities or modern syntactic constructs. In doing so, ZPL made a number of significant contributions including the introduction of the region concept, the WYSIWYG performance model, and the identification of useful abstractions for representing processor sets and distributed arrays and index sets.

Perhaps the biggest indicator of ZPL's success was its influence on the next generation of parallel languages, represented most notably by the High Productivity Computing Systems (HPCS) phase III languages – Cray's Chapel and IBM's X10. Both of these languages adopted the notion of a distributed first-class index set (Chapel's *domain* and X10's *region*) while adapting it to deal with some of ZPL's limitations and generalizing its use to support a more general class of parallel computations.

## Related Entries

- [Array Languages](#)
- [Chapel \(Cray Inc. HPCS Language\)](#)
- [HPF \(High Performance Fortran\)](#)
- [PRAM \(Parallel Random Access Machines\)](#)

## Bibliographic Notes and Further Reading

For an introduction to ZPL and the ZPL project that is both more technical and colorful than this one, the reader is referred to Lawrence Snyder's *The Design and Development of ZPL*, published in the History of Programming Languages Conference, 2007 [15]. Another reasonable technical overview would be a workshop paper published toward the end of the ZPL project that served as somewhat of a “greatest hits” overview of the language at the time that the High Performance Computing community was starting to focus increasingly on end-user productivity [2].

For a reference to the language, the most definitive guide is Snyder's *A Programmer's Guide to ZPL* [14]. However, it comes with the important caveat that the language continued to evolve after the book's publication, and as a result many of the advanced ZPL directions were never captured in a comprehensive book or specification. The dissertations by Chamberlain and Deitz fill in some of the gaps by each starting with an overview of ZPL's capabilities in their second chapters [1, 9]. These descriptions were intentionally incomplete, yet provide a good sense of where

the language ended up. These dissertations also provide descriptions of a number of important design and implementation details in ZPL. Chamberlain's thesis describes the WYSIWYG communication model, the Ironman interface, the runtime descriptors used for regions and arrays, and the extensions to support sparse and hierarchical regions. Deitz's dissertation covers support for processor-local data and computation, the first-class representation of processor grids and resulting redistribution machinery, and the proposed distribution taxonomy mentioned previously.

The most important historical foundations of the language come in the form of descriptions of the CTA and phase abstractions [12, 13]. While the dissertations cited above and by other members of the ZPL project are often the best reference for the final work covered in this entry, stand-alone conference papers that may be more approachable due to their size exist for a number of the topics described here including the Ironman interface [4], WYSIWYG performance model [3], wavefront computations [6], hierarchical regions [5], and sparse regions [7].

Though not the focus of this entry, compiler optimizations were also a crucial part of ZPL's ability to achieve good performance, and a number of papers detail optimizations made to improve the performance of communication [8, 10] and computation [11].

## Bibliography

1. Chamberlain BL (2001) The design and implementation of a region-based parallel language. Ph.D. thesis, University of Washington, Nov 2001. <http://www.cs.washington.edu/homes/brad/cv/pubs/degree/thesis.html>
2. Chamberlain BL, Choi S-E, Deitz SJ, Snyder L (2004) The high-level parallel language ZPL improves productivity and performance. In: Proceedings of the IEEE internationals workshop on productivity and performance in high-end computing, Madrid, 2004
3. Chamberlain BL, Choi S-E, Lewis EC, Lin C, Snyder L, Weathersby WD (1998) ZPL's WYSIWYG performance model. In: Proceedings of the IEEE workshop on high-level parallel programming models and supportive environments, Orlando, 1998
4. Chamberlain BL, Choi S-E, Snyder L (1997) A compiler abstraction for machine independent parallel communication generation. In: Proceedings of the workshop on languages and compilers for parallel computing, Minneapolis, 1997
5. Chamberlain BL, Deitz SJ, Snyder L (2000) A comparative study of the NAS MG benchmark across parallel languages and architectures. In: Proceedings of the ACM international conference on supercomputing, Santa Fe, 2000
6. Chamberlain BL, Lewis EC, Snyder L (1999) Array language support for wavefront and pipelined computations. In: Proceedings of the workshop on languages and compilers for parallel computing, La Jolla, 1999
7. Chamberlain BL, Snyder L (2001) Array language support for parallel sparse computation. In: Proceedings of the ACM international conference on supercomputing, Sorrento, 2001
8. Choi S-E, Snyder L (1997) Quantifying the effects of communication optimizations. In: Proceedings of the IEEE international conference on parallel processing, Bloomington, 1997
9. Deitz SJ (2005) High-level programming language abstractions for advanced and dynamic parallel computations. Ph.D thesis, University of Washington, Feb 2005. <http://www.cs.washington.edu/projects/zpl/papers/data/Deitz05Thesis.pdf>
10. Deitz SJ, Chamberlain BL, Choi S-E, Snyder L (2003) The design and implementation of a parallel array operator for the arbitrary remapping of data. In: Proceedings of the ACM conference on principles and practice of parallel programming, San Diego, 2003
11. Deitz SJ, Chamberlain BL, Snyder L (2001) Eliminating redundancies in sum-of-product array computations. In: Proceedings of the ACM international conference on supercomputing, Sorrento, 2001
12. Lin C (1992) The portability of parallel programs across MIMD computers. Ph.D. thesis, University of Washington, Department of Computer Science and Engineering
13. Snyder L (1995) Experimental validation of models of parallel computation. In: Hofmann A, van Leeuwen J (eds), Computer science today. Lecture notes in computer science, vol 1000. Springer, Berlin, pp 78–100
14. Snyder L (1999) Programming guide to ZPL. MIT, Cambridge
15. Snyder L (2007) The design and development of ZPL. In: HOPL III: Proceedings of the third ACM SIGPLAN conference on history of programming languages, San Diego. ACM, New York, pp 8-1–8-37

# List of Entries

- Ab Initio* Molecular Dynamics  
Access Anomaly  
Actors  
Affinity Scheduling  
Ajtai–Komlós–Szemerédi Sorting Network  
AKS Network  
AKS Sorting Network  
Algebraic Multigrid  
Algorithm Engineering  
Algorithmic Skeletons  
All Prefix Sums  
Allen and Kennedy Algorithm  
Allgather  
All-to-All  
All-to-All Broadcast  
Altivec  
AMD Opteron Processor Barcelona  
Amdahl's Argument  
Amdahl's Law  
AMG  
Analytics, Massive-Scale  
Anomaly Detection  
Anton, A Special-Purpose Molecular Simulation Machine  
Application-Specific Integrated Circuits  
Applications and Parallelism  
Architecture Independence  
Area-Universal Networks  
Array Languages  
Array Languages, Compiler Techniques for  
Asynchronous Iterations  
Asynchronous Iterative Algorithms  
Asynchronous Iterative Computations  
ATLAS (Automatically Tuned Linear Algebra Software)  
Atomic Operations  
Automated Empirical Optimization  
Automated Empirical Tuning  
Automated Performance Tuning  
Automated Tuning  
Automatically Tuned Linear Algebra Software (ATLAS)  
Autotuning  
Backpressure  
Bandwidth-Latency Models (BSP, Logp)  
Banerjee's Dependence Test  
Barnes-Hut  
Barriers  
Basic Linear Algebra Subprograms (BLAS)  
Behavioral Equivalences  
Behavioral Relations  
Benchmarks  
Beowulf Clusters  
Beowulf-Class Clusters  
Bernstein's Conditions  
Bioinformatics  
Bisimilarity  
Bisimulation  
Bisimulation Equivalence  
Bitonic Sort  
Bitonic Sorting Network  
Bitonic Sorting, Adaptive  
BLAS (Basic Linear Algebra Subprograms)  
Blocking  
Blue CHiP  
Blue CHiP Project  
Blue Gene/L  
Blue Gene/P  
Blue Gene/Q  
Branch Predictors  
Brent's Law  
Brent's Theorem  
Broadcast  
BSP  
BSP (Bulk Synchronous Parallelism)  
Bulk Synchronous Parallelism (BSP)  
Bus: Shared Channel  
Buses and Crossbars  
Butterfly  
 $C^*$   
Cache Affinity Scheduling

Cache Coherence  
Cache-Only Memory Architecture (COMA)  
Caches, NUMA  
Calculus of Mobile Processes  
Carbon Cycle Research  
Car-Parrinello Method  
CDC 6600  
Cedar Multiprocessor  
CELL  
Cell Broadband Engine Processor  
Cell Processor  
Cell/B.E.  
Cellular Automata  
Chaco  
Chapel (Cray Inc. HPCS Language)  
Charm++  
Checkpoint/Restart  
Checkpointing  
Checkpoint-Recovery  
CHiP Architecture  
CHiP Computer  
Cholesky Factorization  
Cilk  
Cilk Plus  
Cilk++  
Cilk-1  
Cilk-5  
Cilkscreen  
Cluster File Systems  
Cluster of Workstations  
Clusters  
CM Fortran  
Cm\* - The First Non-Uniform Memory Access  
Architecture  
CML  
CM-Lisp  
CnC  
Coarray Fortran  
Code Generation  
Collect  
Collective Communication  
Collective Communication, Network Support For  
COMA (Cache-Only Memory Architecture)  
Combinatorial Search  
Commodity Clusters  
Communicating Sequential Processes (CSP)  
Community Atmosphere Model (CAM)  
Community Climate Model (CCM)  
Community Climate System Model  
Community Climate System Model (CCSM)  
Community Earth System Model (CESM)  
Community Ice Code (CICE)  
Community Land Model (CLM)  
Compiler Optimizations for Array Languages  
Compilers  
Complete Exchange  
Complex Event Processing  
Computational Biology  
Computational Chemistry  
Computational Models  
Computational Sciences  
Computer Graphics  
Computing Surface  
Concatenation  
Concurrency Control  
Concurrent Collections Programming Model  
Concurrent Logic Languages  
Concurrent ML  
Concurrent Prolog  
Configurable, Highly Parallel Computer  
Congestion Control  
Congestion Management  
Connected Components Algorithm  
Connection Machine  
Connection Machine Fortran  
Connection Machine Lisp  
Consistent Hashing  
Control Data 6600  
Coordination  
Copy  
Core2-Duo / Core2-Quad Processors  
COW  
Crash Simulation  
Cray MTA  
Cray Red Storm  
Cray SeaStar Interconnect  
CRAY T3E  
Cray Vector Computers  
Cray XMT  
Cray XT Series  
Cray XT3  
Cray XT3 and Cray XT Series of Supercomputers  
Cray XT4  
Cray XT4 and Seastar 3-D Torus Interconnect

|                                          |                                                         |
|------------------------------------------|---------------------------------------------------------|
| Cray XT5                                 | Distributed Logic Languages                             |
| Cray XT6                                 | Distributed Memory Computers                            |
| Critical Race                            | Distributed Process Management                          |
| Critical Sections                        | Distributed Switched Networks                           |
| Crossbar                                 | Distributed-Memory Multiprocessor                       |
| CS-2                                     | Ditonic Sorting                                         |
| CSP (Communicating Sequential Processes) | DLPAR                                                   |
| Cyclops                                  | Doall Loops                                             |
| Cyclops-64                               | Domain Decomposition                                    |
| Cydra 5                                  | DPJ                                                     |
| DAG Scheduling                           | DR                                                      |
| Data Analytics                           | Dynamic Logical Partitioning for POWER Systems          |
| Data Centers                             | Dynamic LPAR                                            |
| Data Distribution                        | Dynamic Reconfiguration                                 |
| Data Flow Computer Architecture          | Earth Simulator                                         |
| Data Flow Graphs                         | Eden                                                    |
| Data Mining                              | Eigenvalue and Singular-Value Problems                  |
| Data Race Detection                      | EPIC Processors                                         |
| Data Starvation Crisis                   | Erlangen General Purpose Array (EGPA)                   |
| Dataflow Supercomputer                   | ES                                                      |
| Data-Parallel Execution Extensions       | Ethernet                                                |
| Deadlock Detection                       | Event Stream Processing                                 |
| Deadlocks                                | Eventual Values                                         |
| Debugging                                | Exact Dependence                                        |
| DEC Alpha                                | Exaflop Computing                                       |
| Decentralization                         | Exaop Computing                                         |
| Decomposition                            | Exascale Computing                                      |
| Deep Analytics                           | Execution Ordering                                      |
| Denelcor HEP                             | Experimental Parallel Algorithmics                      |
| Dense Linear System Solvers              | Extensional Equivalences                                |
| Dependence Abstractions                  | Fast Fourier Transform (FFT)                            |
| Dependence Accuracy                      | Fast Multipole Method (FMM)                             |
| Dependence Analysis                      | Fast Poisson Solvers                                    |
| Dependence Approximation                 | Fat Tree                                                |
| Dependence Cone                          | Fault Tolerance                                         |
| Dependence Direction Vector              | Fences                                                  |
| Dependence Level                         | FFT (Fast Fourier Transform)                            |
| Dependence Polyhedron                    | Fast Algorithm for the Discrete Fourier Transform (DFT) |
| Dependences                              | FFTW                                                    |
| Detection of DOALL Loops                 | File Systems                                            |
| Determinacy                              | Fill-Reducing Orderings                                 |
| Determinacy Race                         | First-Principles Molecular Dynamics                     |
| Determinism                              | Fixed-Size Speedup                                      |
| Deterministic Parallel Java              | FLAME                                                   |
| Direct Schemes                           | Floating Point Systems FPS-120B and Derivatives         |
| Distributed Computer                     | Flow Control                                            |
| Distributed Hash Table (DHT)             |                                                         |

|                                                                 |                                                    |
|-----------------------------------------------------------------|----------------------------------------------------|
| Flynn's Taxonomy                                                | Hazard (in Hardware)                               |
| Forall Loops                                                    | HDF5                                               |
| FORGE                                                           | HEP, Denelcor                                      |
| Formal Methods-Based Tools for Race, Deadlock, and Other Errors | Heterogeneous Element Processor                    |
| Fortran 90 and Its Successors                                   | Hierarchical Data Format                           |
| Fortran, Connection Machine                                     | High Performance Fortran (HPF)                     |
| Fortress (Sun HPCS Language)                                    | High-Level I/O Library                             |
| Forwarding                                                      | High-Performance I/O                               |
| Fujitsu Vector Computers                                        | Homology to Sequence Alignment, From               |
| Fujitsu Vector Processors                                       | Horizon                                            |
| Fujitsu VPP Systems                                             | HPC Challenge Benchmark                            |
| Functional Decomposition                                        | HPF (High Performance Fortran)                     |
| Functional Languages                                            | HPS Microarchitecture                              |
| Futures                                                         | HT                                                 |
| GA                                                              | HT3.10                                             |
| Gather                                                          | Hybrid Programming With SIMPLE                     |
| Gather-to-All                                                   | Hypercube                                          |
| Gaussian Elimination                                            | Hypercubes and Meshes                              |
| GCD Test                                                        | Hypergraph Partitioning                            |
| Gene Networks Reconstruction                                    | Hyperplane Partitioning                            |
| Gene Networks Reverse-Engineering                               | HyperTransport                                     |
| Generalized Meshes and Tori                                     | IBM Blue Gene Supercomputer                        |
| Genome Assembly                                                 | IBM Power                                          |
| Genome Sequencing                                               | IBM Power Architecture                             |
| 3GIO                                                            | IBM PowerPC                                        |
| Glasgow Parallel Haskell (GpH)                                  | IBM RS/6000 SP                                     |
| Global Arrays                                                   | IBM SP                                             |
| Global Arrays Parallel Programming Toolkit                      | IBM SP1                                            |
| Gossiping                                                       | IBM SP2                                            |
| GpH (Glasgow Parallel Haskell)                                  | IBM SP3                                            |
| GRAPE                                                           | IBM System/360 Model 91                            |
| Graph Algorithms                                                | IEEE 802.3                                         |
| Graph Analysis Software                                         | Illegal Memory Access                              |
| Graph Partitioning                                              | Illiac IV                                          |
| Graph Partitioning Software                                     | ILUPACK                                            |
| Graphics Processing Unit                                        | Impass                                             |
| Green Flash: Climate Machine (LBNL)                             | Implementations of Shared Memory in Software       |
| Grid Partitioning                                               | Index                                              |
| Gridlock                                                        | InfiniBand                                         |
| Group Communication                                             | Instant Replay                                     |
| Gustafson's Law                                                 | Instruction-Level Parallelism                      |
| Gustafson–Barsis Law                                            | Instruction Systolic Arrays                        |
| Half Vector Length                                              | Intel Celeron                                      |
| Hang                                                            | Intel Core Microarchitecture, x86 Processor Family |
| Harmful Shared-Memory Access                                    | Intel® Parallel Inspector                          |
| Haskell                                                         | Intel® Parallel Studio                             |
|                                                                 | Intel® Thread Profiler                             |

|                                                   |                                                     |
|---------------------------------------------------|-----------------------------------------------------|
| Intel® Threading Building Blocks (TBB)            | Locality of Reference and Parallel Processing       |
| Interactive Parallelization                       | Lock-Free Algorithms                                |
| Interconnection Network                           | Locks                                               |
| Interconnection Networks                          | Logarithmic-Depth Sorting Network                   |
| Internet Data Centers                             | Logic Languages                                     |
| Inter-Process Communication                       | LogP Bandwidth-Latency Model                        |
| I/O                                               | Loop Blocking                                       |
| iPSC                                              | Loop Nest Parallelization                           |
| Isoefficiency                                     | Loop Tiling                                         |
| JANUS FPGA-Based Machine                          | Loops, Parallel                                     |
| Java                                              | LU Factorization                                    |
| JavaParty                                         | Manycore                                            |
| Job Scheduling                                    | MapReduce                                           |
| k-ary n-cube                                      | MasPar                                              |
| k-ary n-fly                                       | Massively Parallel Processor (MPP)                  |
| k-ary n-tree                                      | Massive-Scale Analytics                             |
| Knowledge Discovery                               | Matrix Computations                                 |
| KSR                                               | Maude                                               |
| LANai                                             | Media Extensions                                    |
| Languages                                         | Meiko                                               |
| LAPACK                                            | Memory Consistency Models                           |
| Large-Scale Analytics                             | Memory Models                                       |
| Latency Hiding                                    | Memory Ordering                                     |
| Law of Diminishing Returns                        | Memory Wall                                         |
| Laws                                              | MEMSY                                               |
| Layout, Array                                     | Mesh                                                |
| LBNL Climate Computer                             | Mesh Partitioning                                   |
| libflame                                          | Message Passing                                     |
| Libraries, Numerical                              | Message Passing Interface (MPI)                     |
| Linda                                             | Message-Passing Performance Models                  |
| Linear Algebra Software                           | METIS and ParMETIS                                  |
| Linear Algebra, Numerical                         | Metrics                                             |
| Linear Equations Solvers                          | Microprocessors                                     |
| Linear Least Squares and Orthogonal Factorization | MILC                                                |
| Linear Regression                                 | MIMD (Multiple Instruction, Multiple Data) Machines |
| LINPACK Benchmark                                 | MIMD Lattice Computation                            |
| Linux Clusters                                    | MIN                                                 |
| *Lisp                                             | ML                                                  |
| Lisp, Connection Machine                          | MMX                                                 |
| Little's Law                                      | Model Coupling Toolkit (MCT)                        |
| Little's Lemma                                    | Models for Algorithm Design and Analysis            |
| Little's Principle                                | Models of Computation, Theoretical                  |
| Little's Result                                   | Modulo Scheduling and Loop Pipelining               |
| Little's Theorem                                  | Molecular Evolution                                 |
| Livermore Loops                                   | Monitors                                            |
| Load Balancing                                    | Monitors, Axiomatic Verification of                 |
| Load Balancing, Distributed Memory                | Moore's Law                                         |

MPI (Message Passing Interface)  
MPI-2 I/O  
MPI-IO  
MPP  
Mul-T  
Multicomputers  
Multicore Networks  
Multiflow Computer  
Multifrontal Method  
Multi-Level Transactions  
Multilisp  
Multimedia Extensions  
Multiple-Instruction Issue  
Multiprocessor Networks  
Multiprocessor Synchronization  
Multiprocessors  
Multiprocessors, Symmetric  
MultiScheme  
Multistage Interconnection Networks  
Multi-Streamed Processors  
Multi-Threaded Processors  
Mumps  
MUMPS  
Mutual Exclusion  
Myri-10G  
Myricom  
Myrinet  
NAMD (NA noscale Molecular Dynamics)  
NA noscale Molecular Dynamics (NAMD)  
NAS Parallel Benchmarks  
N-Body Computational Methods  
nCUBE  
NEC SX Series Vector Computers  
NESL  
Nested Loops Scheduling  
Nested Spheres of Control  
NetCDF I/O Library, Parallel  
Network Adapter  
Network Architecture  
Network Interfaces  
Network Obliviousness  
Network of Workstations  
Network offload  
Networks, Direct  
Networks, Fault-Tolerant  
Networks, Multistage  
NI (Network Interface)  
NIC (Network Interface Controller or Network Interface Card)  
Node Allocation  
Non-Blocking Algorithms  
Nondeterminator  
Nonuniform Memory Access (NUMA) Machines  
NOW  
NUMA Caches  
Numerical Algorithms  
Numerical Libraries  
Numerical Linear Algebra  
NVIDIA GPU  
NWChem  
Omega Calculator  
Omega Library  
Omega Project  
Omega Test  
One-to-All Broadcast  
Open Distributed Systems  
OpenMP  
OpenMP Profiling with OmpP  
OpenSHMEM - Toward a Unified RMA Model  
Operating System Strategies  
Optimistic Loop Parallelization  
Orthogonal Factorization  
OS Jitter  
OS, Light-Weight  
Out-of-Order Execution Processors  
Overdetermined Systems  
Overlay Network  
Owicki-Gries Method of Axiomatic Verification  
Parafrase  
Parallel Communication Models  
Parallel Computing  
Parallel I/O Library (PIO)  
Parallel Ocean Program (POP)  
Parallel Operating System  
Parallel Prefix Algorithms  
Parallel Prefix Sums  
Parallel Random Access Machines (PRAM)  
Parallel Skeletons  
Parallel Tools Platform  
Parallelism Detection in Nested Loops, Optimal  
Parallelization  
Parallelization, Automatic  
Parallelization, Basic Block  
Parallelization, Loop Nest

|                                                                 |                                              |
|-----------------------------------------------------------------|----------------------------------------------|
| ParaMETIS                                                       | Power Wall                                   |
| PARDISO                                                         | PRAM (Parallel Random Access Machines)       |
| PARSEC Benchmarks                                               | Preconditioners for Sparse Iterative Methods |
| Partial Computation                                             | Prefix                                       |
| Particle Dynamics                                               | Prefix Reduction                             |
| Particle Methods                                                | Problem Architectures                        |
| Partitioned Global Address Space (PGAS) Languages               | Process Algebras                             |
| PASM Parallel Processing System                                 | Process Calculi                              |
| Path Expressions                                                | Process Description Languages                |
| PaToH (Partitioning Tool for Hypergraphs)                       | Process Synchronization                      |
| Partitioning Tool for Hypergraphs (PaToH)                       | Processes, Tasks, and Threads                |
| PC Clusters                                                     | Processor Allocation                         |
| PCI Express                                                     | Processor Arrays                             |
| PCIe                                                            | Processors-in-Memory                         |
| PCI-E                                                           | Profiling                                    |
| PCI-Express                                                     | Profiling with OmpP, OpenMP                  |
| Peer-to-Peer                                                    | Program Graphs                               |
| Pentium                                                         | Programmable Interconnect Computer           |
| PERCS System Architecture                                       | Programming Languages                        |
| Perfect Benchmarks                                              | Programming Models                           |
| Performance Analysis Tools                                      | Prolog                                       |
| Performance Measurement                                         | Prolog Machines                              |
| Performance Metrics                                             | Promises                                     |
| Periscope                                                       | Protein Docking                              |
| Personalized All-to-All Exchange                                | Pthreads (POSIX Threads)                     |
| Petaflop Barrier                                                | PVM (Parallel Virtual Machine)               |
| Petascale Computer                                              | QCD apeNEXT Machines                         |
| Petri Nets                                                      | QCD (Quantum Chromodynamics) Computations    |
| PETSc (Portable, Extensible Toolkit for Scientific Computation) | QCD Machines                                 |
| PGAS (Partitioned Global Address Space) Languages               | QCDSP and QCDOC Computers                    |
| Phylogenetic Inference                                          | QsNet                                        |
| Phylogenetics                                                   | Quadratics                                   |
| Pi-Calculus                                                     | Quantum Chemistry                            |
| Pipelining                                                      | Quantum Chromodynamics (QCD) Computations    |
| Place-Transition Nets                                           | Quicksort                                    |
| PLAPACK                                                         | Race                                         |
| PLASMA                                                          | Race Conditions                              |
| PMPI Tools                                                      | Race Detection Techniques                    |
| Pnetcdf                                                         | Race Detectors for Cilk and Cilk++ Programs  |
| Point-to-Point Switch                                           | Race Hazard                                  |
| Polaris                                                         | Radix Sort                                   |
| Polyhedra Scanning                                              | Rapid Elliptic Solvers                       |
| Polyhedron Model                                                | Reconfigurable Computer                      |
| Polytope Model                                                  | Reconfigurable Computers                     |
| Position Tree                                                   | Reconstruction of Evolutionary Trees         |
| POSIX Threads (Pthreads)                                        | Reduce and Scan                              |
|                                                                 | Relaxed Memory Consistency Models            |

|                                                                   |                                                                     |
|-------------------------------------------------------------------|---------------------------------------------------------------------|
| Reliable Networks                                                 | SoC (System on Chip)                                                |
| Rendezvous                                                        | Social Networks                                                     |
| Reordering                                                        | Software Autotuning                                                 |
| Resource Affinity Scheduling                                      | Software Distributed Shared Memory                                  |
| Resource Management for Parallel<br>Computers                     | Sorting                                                             |
| Rewriting Logic                                                   | Space-Filling Curves                                                |
| Ring                                                              | SPAI (SParse Approximate Inverse)                                   |
| Roadrunner Project, Los Alamos                                    | Spanning Tree, Minimum Weight                                       |
| Router Architecture                                               | Sparse Approximate Inverse Matrix                                   |
| Router-Based Networks                                             | Sparse Direct Methods                                               |
| Routing (Including Deadlock Avoidance)                            | Sparse Gaussian Elimination                                         |
| R-Stream Compiler                                                 | Sparse Iterative Methods, Preconditioners for<br>SPEC Benchmarks    |
| Run Time Parallelization                                          | SPEC HPC2002                                                        |
| Runtime System                                                    | SPEC HPC96                                                          |
| Scalability                                                       | SPEC MPI2007                                                        |
| Scalable Coherent Interface (SCI)                                 | SPEC OMP2001                                                        |
| ScaLAPACK                                                         | Special-Purpose Machines                                            |
| Scalasca                                                          | Speculation                                                         |
| Scaled Speedup                                                    | Speculation, Thread-Level                                           |
| Scan for Distributed Memory, Message-Passing<br>Systems           | Speculative Multithreading (SM)                                     |
| Scan, Reduce and                                                  | Speculative Parallelization                                         |
| Scatter                                                           | Speculative Parallelization of Loops                                |
| Scheduling                                                        | Speculative Run-Time Parallelization                                |
| Scheduling Algorithms                                             | Speculative Threading                                               |
| SCI (Scalable Coherent Interface)                                 | Speculative Thread-Level Parallelization                            |
| Semantic Independence                                             | Speedup                                                             |
| Semaphores                                                        | SPIKE                                                               |
| Sequential Consistency                                            | Spiral                                                              |
| Server Farm                                                       | SPMD Computational Model                                            |
| Shared Interconnect                                               | SSE                                                                 |
| Shared Virtual Memory                                             | Stalemate                                                           |
| Shared-Medium Network                                             | State Space Search                                                  |
| Shared-Memory Multiprocessors                                     | Stream Processing                                                   |
| SHMEM                                                             | Stream Programming Languages                                        |
| SIGMA-1                                                           | Strong Scaling                                                      |
| SIMD (Single Instruction, Multiple Data) Machines                 | Suffix Trees                                                        |
| SIMD Extensions                                                   | Superlinear Speedup                                                 |
| SIMD ISA                                                          | SuperLU                                                             |
| Single System Image                                               | Supernode Partitioning                                              |
| Singular-Value Decomposition (SVD)                                | Superscalar Processors                                              |
| Sisal                                                             | SWARM: A Parallel Programming Framework for<br>Multicore Processors |
| Small-World Network Analysis and Partitioning<br>(SNAP) Framework | Switch Architecture                                                 |
| SNAP (Small-World Network Analysis and<br>Partitioning) Framework | Switched-Medium Network                                             |
|                                                                   | Switching Techniques                                                |
|                                                                   | Symmetric Multiprocessors                                           |

|                                       |                                                                                      |
|---------------------------------------|--------------------------------------------------------------------------------------|
| Synchronization                       | Transactions, Nested                                                                 |
| System Integration                    | Transpose                                                                            |
| System on Chip (SoC)                  | TStreams                                                                             |
| Systems Biology, Network Inference in | Tuning and Analysis Utilities                                                        |
| Systolic Architecture                 | Ultracomputer, NYU                                                                   |
| Systolic Arrays                       | Uncertainty Quantification                                                           |
| Task Graph Scheduling                 | Underdetermined Systems                                                              |
| Task Mapping, Topology Aware          | Unified Parallel C                                                                   |
| Tasks                                 | Unimodular Transformations                                                           |
| TAU                                   | Universal VLSI Circuits                                                              |
| TAU Performance System®               | Universality in VLSI Computation                                                     |
| TBB (Intel Threading Building Blocks) | UPC                                                                                  |
| Tensilica                             | Use-Def Chains                                                                       |
| Tera MTA                              | Vampir                                                                               |
| Terrestrial Ecosystem Carbon Modeling | Vampir 7                                                                             |
| Terrestrial Ecosystem Modeling        | Vampir NG                                                                            |
| The High Performance Substrate        | VampirServer                                                                         |
| Theory of Mazurkiewicz-Traces         | VampirTrace                                                                          |
| Thick Ethernet                        | Vector Extensions, Instruction-Set Architecture (ISA)                                |
| Thin Ethernet                         | Vectorization                                                                        |
| Thread Level Speculation (TLS)        | Verification of Parallel Shared-Memory Programs,<br>Owicki-Gries Method of Axiomatic |
| Parallelization                       |                                                                                      |
| Thread-Level Data Speculation (TLDS)  | View from Berkeley                                                                   |
| Thread-Level Speculation              | Virtual Shared Memory                                                                |
| Threads                               | VLIW Processors                                                                      |
| Tiling                                | VLSI Algorithmics                                                                    |
| Titanium                              | VLSI Computation                                                                     |
| TLS                                   | VNG                                                                                  |
| TOP500                                | Warp and iWarp                                                                       |
| Topology Aware Task Mapping           | Wavefront Arrays                                                                     |
| Torus                                 | Weak Scaling                                                                         |
| Total Exchange                        | Whole Program Analysis                                                               |
| Trace Scheduling                      | Work-Depth Model                                                                     |
| Trace Theory                          | Workflow Scheduling                                                                  |
| Tracing                               | Z-Level Programming Language                                                         |
| Transactional Memories                | ZPL                                                                                  |

